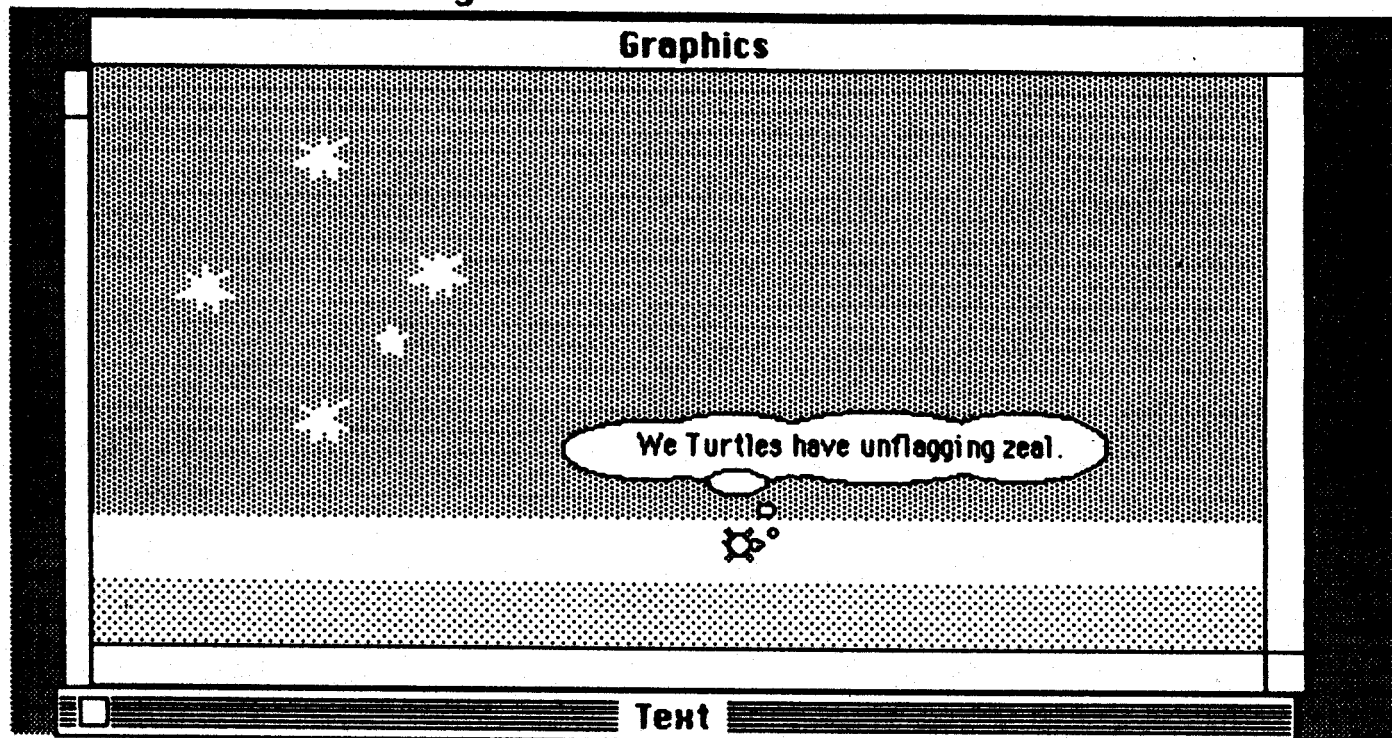


POALL

A Journal for Logo Users

File Edit Debug



Flags

Volume 2 Number 3, July 1987

At last, this issue is ready. To me, it seems to have too much of the computer science flavour; we really need people to write about what they're doing at Primary and Junior Primary level to restore the balance that was intended for POALL at the beginning.

Introducing the theme article is a rendition of Wayne Stokes winning entry in the Austflag 88 competition, a flag we now seem unlikely to see much of in the future. Don't dismiss the article on page 8 simply because it deals with a problem with a specific Logo. If you've ever wondered what .CONTENTS was for, here it is in use, and there are some general problem solving clues along the way.

Briefly reviewed in this issue are two new books from people with Logo almost from the beginning, Sylvia Weir and Cynthia Solomon. Sylvia Weir is to be in Australia during August; hear her if you have the opportunity.

Preparations for ACEC '87 are warming up, and some fine Logo papers have come in. To follow ACEC '87 there's to be a day workshop at SACAE Magill on Oct. 1st. Present planning is for two strands, Logo in Mathematics and Logo in Computer Science, the latter probably with Tony Adams of RMIT. Make a date. Make a date too for BIKIlog: Aug. 10th, Sept. 7th, Nov. 9th, Dec 7th, 7:30 pm in the microcomputing building at SACAE Magill.

Peter

POTS

- | | |
|-----------------------------|-----------------------|
| 2 Turtle Vexillology | 15 Nested Polys |
| 5 Resources | 16 A Game with a name |
| 8 AL // and the ImageWriter | 18 Entropy House |
| 12 STOPping in Style | |

Turtle Vexillology:

Flags as objects of design:

Ever thought of flags as project ideas for Logo? Flags are colourful things, and generally composed of simple elements like blocks of colour, disks, stars and the like. For most flags it's just a matter of working out where the pieces belong and putting them there. Of course, the flags you choose will be dependent on the system you have; its available colours and resolution, and you may have to take a few liberties here and there. The proportions will not always be correct, but the whole screen might as well be used.

There are two design aspects in Logo flags. The first is the flag itself. What do the colours symbolise? What is the significance of things like the fourteen points on the star on the Malaysian flag? What events are commemorated by the flag? What does 'vexillology' mean? What is 'good' vexillology? Enough questions to keep researchers busy for some time. Looked at the other way round, here's some Logo to go with Social Studies for a change.

The other aspect is that of program design. According to the programming books, programs should be designed 'top-down', that is, from the abstract to the particular. As an example, let's consider the Australian flag, and pseudocode part of it:

```
to draw an Australian flag
  take a blue screen
  draw the Union Jack
  draw the Commonwealth Star
  draw the Southern Cross
```

How does one draw the Union Jack? Easy:

```
to draw the Union Jack
  draw St Andrew's Cross
  draw St Patrick's Cross
  draw St George's Cross
```

St Andrew's Cross?

```
to draw St Andrew's Cross
  stand on the corners of the Union Jack and draw diagonal white blocks
```

And so it goes, until finally you have to tell the Turtle to actually do something, to go someplace with its pen and draw lines. If you think about it, to draw the Australian flag you need only two real procedures, one to draw rectangular blocks of colour, the other to draw 5 or 7 point stars. I'll let you draw a structure chart and write the Logo.

Note that I've used the real names of the parts of the flag (yes, I know it should be St. Andrew's saltire, and you *will* have fun with St Patrick's) for the names of procedures to make the program easier to understand. Meaning and understanding are important to programming because programs are really for people to read, not for some dumb machine to churn through. Logo is a problem solving notation for humans, not just a way of writing instructions for computers.

'Bottom-up' programming is almost the natural style of Logo; using known procedures to do something interesting. With procedures to draw rectangular blocks, disks, triangles and stars we can mess about. The students who drew the Panamanian and Norwegian flags shown opposite knew how to draw blocks and stars, they simply decided to draw flags using those elements and experimented until they had things where they wanted them. Putting the blue cross in its place took several attempts, but the final set of procedures is quite logical and easy

to follow. Similarly, the girls working on the Czechoslovakian flag found that Apple Logo would not let them draw blue on red. After thinking about it, they solved their problem by drawing a white triangle, then redrawing it blue.

The procedures for drawing the shapes are worth spending some time on and devising concise logic. Unless your Logo has a FILL primitive or is on the BBC with its triangle filling 'nib' you'll have to fill in areas by drawing a line, moving over a bit, drawing another line etc. Will you need an outline? As for the BBC's triangles, beware, or you may be filling unwanted triangles. Here's a way to safely draw seven point BBC stars:

```
TO SAFE
REPEAT 3 [FD 0]
END

TO STAR7 :size
SETPOS SE XPOS YPOS + (:size * .51)
PD SETH 167.2 Safe
SETNIB 80
FD :size RT 154.2
REPEAT 6 [FD :size * .55 FD :size * .45 RT 154.3]
SETNIB 8 Safe
END
```

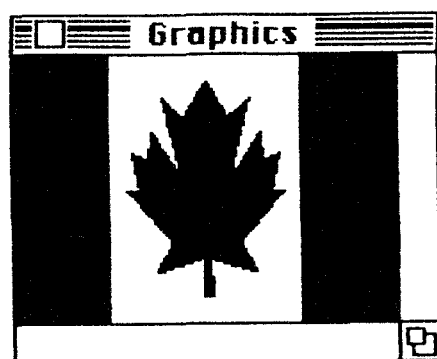
You can write the 5 point version. On other systems, modify the ubiquitous PolySpi procedure with an extra variable so that it can be made to stop at a given size. Finding the right angles will be good exploration.

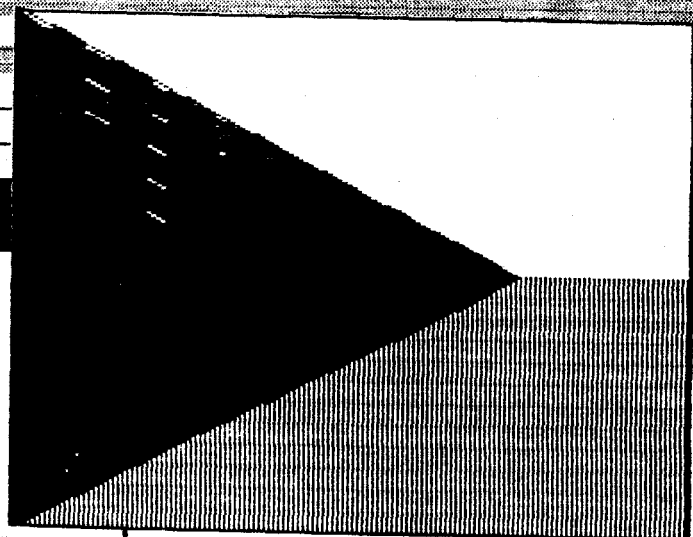
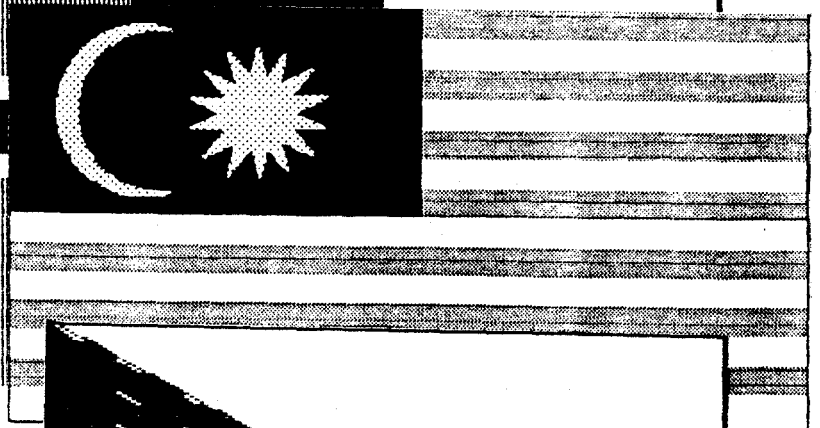
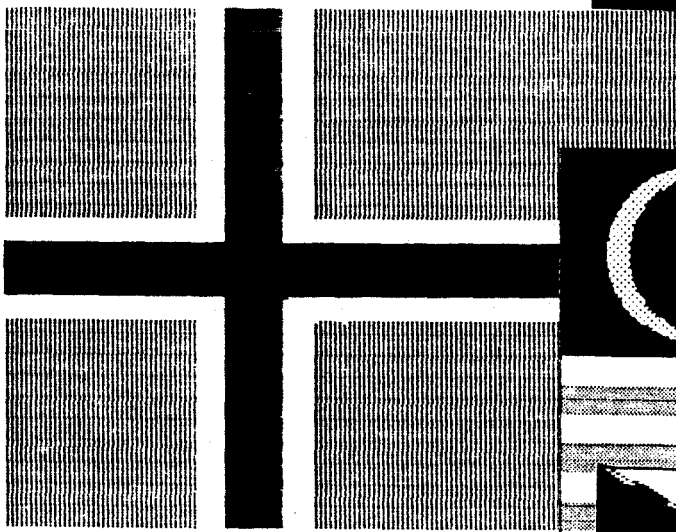
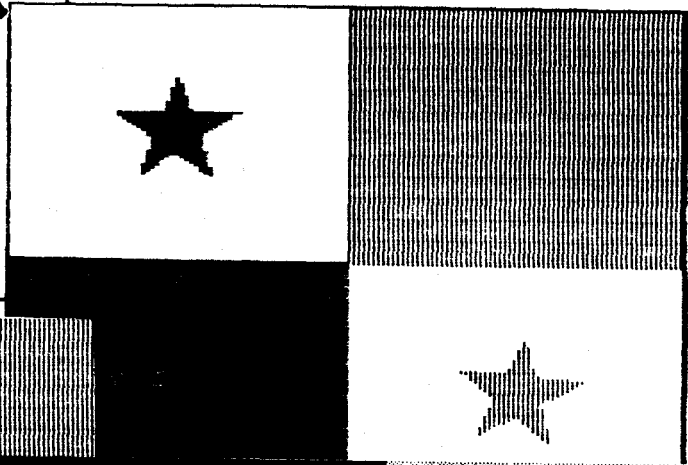
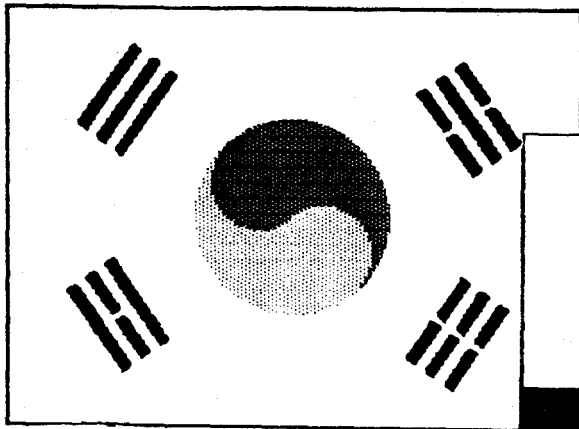
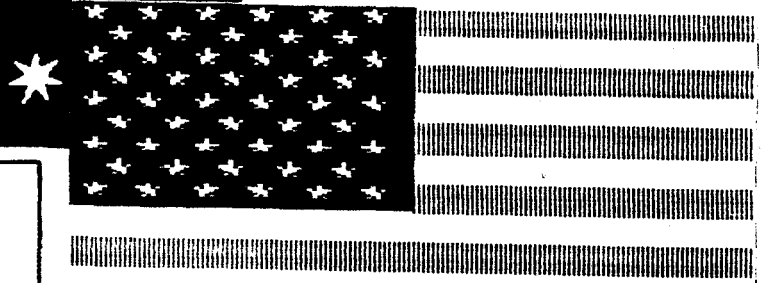
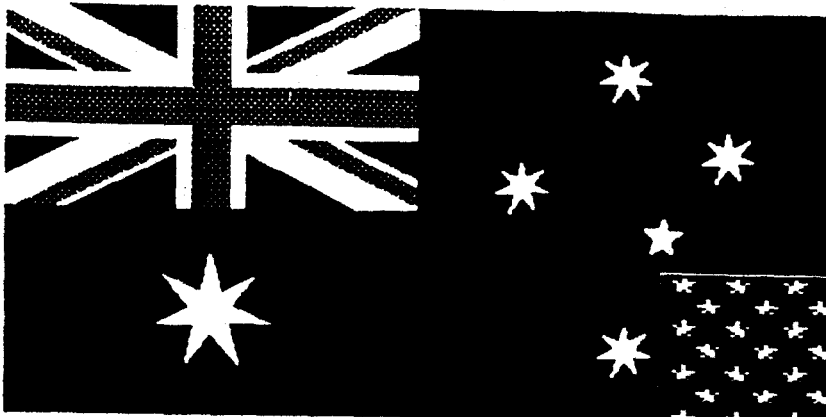
Odd shapes may sometimes be easier to draw 'inside out', by drawing a square and then erasing around the edge to form the required shape. Disks are easy, FORWARD a bit, BACK a bit, turn a bit etc. I'll let you work from there to shapes like the Yin and Yang on the South Korean flag and the crescent moon on the flags of many Moslem countries.

REPEAT loops will often be the way to arrange a series of elements like the stripes on the United States or Malaysian flags, with plain sequential code the way to go for many other things. The stars on the US flag shown (go on, count them) were drawn by three procedures, one to draw a row of 5, another a row of 6, and the third which kept going recursively until it sensed that it had finished by keeping track of its YCOR. Tasks like that could also be tackled with lists of coordinates, giving you an excuse to introduce list processing.

Whichever way, top-down, bottom-up or middle-out, there are lots of good design opportunities, and if you should ever run out of national flags, some of the flags of Japanese prefectures are quite striking. Then there are the international code flags, and you can design your own flags for special occasions too, or use the emblems of special events: last year it was the Jubilee 150 logo, next year the bicentennial. Something for everyone, Year 5 to Year 12 and beyond.

There's a page of the next issue waiting for your contributions.





Resources

Two books this time, by people who have had long experience with, and influence on, Logo. The first is *Cultivating Minds: A Logo Casebook* by Sylvia Weir, the second, *Computer Environments for Children* by Cynthia Solomon. Neither is readily available off the shelf yet, in fact Solomon's book is direct from MIT Press in Boston, but both deserve to be widely read.

'In my vision...' wrote Papert in *Mindstorms*, describing what ought to be as much as what had been done. Now, in *Cultivating Minds*, Sylvia Weir shows that what many people tended to dismiss as mere idealism was in fact, very real. Logo is a system with a promise. It is a system which can change education; can serve as the medium for children, including those with deep intellectual, physical and emotional problems, to communicate and explore. This is not another collection of anecdotes, but the result of several meticulously documented research projects.

Originally a medical practitioner, Sylvia Weir found it natural to be drawn into working with children with problems, and has been associated with Logo at both MIT and Edinburgh. Her book, as its subtitle implies, contains case histories of a number of children whose lives have been profoundly changed by their contact with the computer, and Logo in particular. One is 'Michael Murphy', with quadriplegic cerebral palsy, barely able to move and to speak, yet with an intelligence waiting to be liberated by Logo, an intelligence that was to enable him to go on to university studies. Another is the autistic 'Donald' who spontaneously and excitedly came to shout 'See how it works!' as he directed a hardware Turtle. Physical Turtles turn out to be an important influence in helping children with problems, especially autism.

While much of the book is taken up with description of the painstaking and patient work with children with problems, there is a great deal of useful insight for all teachers. There is sound discussion of psychological theory and teaching in general. There is also excellent discourse on the subject of metaknowledge, at once both Logo's strength and its weakness. Logo is one of the few educational systems which allow, almost force, teachers to think about their own knowledge and thinking, and how to help children think about their own thinking. The weakness lies in teachers and students being unwilling, afraid even, to expose themselves to this sort of thing and either abandoning Logo or using it in a strictly teacher directed manner. A small portion from some extensive discussion:

A good way to learn something is to work with an expert as an apprentice and engage in "guided messing about" (Hawkins, *The Informed Vision*), rather than receive a pure diet of rules, principles and concepts. Young children tend to learn by "messing about" naturally. Unfortunately they are gradually brainwashed out of using this method as they become more sophisticated and subject to social beliefs about "needing to be taught." Computer-based systems such as Logo have the potential for doing something about this, but the potential is not realised automatically. It is necessary to contrive the "messing about" to make it appropriate to kinds of experience and so to invite the desired learning.

But more is needed than the opportunity to mess about. As well as doing, we can think about what we are doing. As well as perceiving, we can reflect upon our experience. Piaget called this "reflective abstraction." Conscious self-reflection is slow to develop, and we mature as individuals to the extent that we can look at our own functioning. It must be of benefit to a growing child to be encouraged by the very nature of the learning environment to "look at her own thinking." (pp 79..80)

As Weir points out elsewhere, some students have difficulties because their thinking is unconscious, they cannot readily gain access to it to help in any debugging process.

What is learning?

How is it that a mind can come to understand anything new? During the usual course of events, if the new experience is sufficiently close to an old one, it can be seen as a version of that old situation. Interpreting the mismatch between the two and updating the stored schema accordingly constitute learning. Learning will happen more readily when new and old do not match exactly, but *nearly* match. Instead of recording the description of the new as an isolated event, the learner records the difference between it and the already recorded old event, and, perhaps, the significance of that difference. The central process is making connections between experiences and between things; making judgements: about what is the same and what is different; and noting the meaning of the difference. (p 167)

Cynthia Solomon holds a PhD in education from Harvard University, in fact her book is a revised and expanded version of her thesis. She worked for many years with Seymour Papert's group at MIT, and also with the Atari Research Laboratory. *Computer Environments for Children* is a comparative study of four different styles of working in schools with computers, and more importantly, of the educational theories behind them. As a confirmed Logo user, she has an obvious leaning, but she treats all four with sympathy and reason, and knows the subjects of her comparisons personally.

The first is the drill and practice and rote learning environment of Patrick Suppes, a behaviourist. Suppes and his team developed an elementary mathematics curriculum which was used in a number of areas in the US through the 1970s, at first on time sharing systems and now for microcomputers.

Suppes' approach has wide support in education, particularly among those who hope that the computer can teach things with which teachers have not succeeded, and to students for whom the school system has failed. Both industry and government tend to favour rote learning ideas and the notion of the computer as an intelligent and sympathetic teacher. Among the attractive features, as Solomon sees them, is the fact that the system seems to make good on its promises. The materials are being used mainly with disadvantaged and low income students, who, in graded tests, show improvement.

People like the Suppes approach for different reasons:

"Theorists" like it because it has a clear intellectual structure and scientific pedigree. "Hard-nosed empiricists" like it because its effects can be measured. "Administrators" like it because its cost structure is clear. "Teachers" like it because they are free to do other activities. (p 22)

As she points out, however, although the underlying psychology may be statistically satisfying, it is of extremely limited scope. Even Suppes admits that there are complexities unaccounted for.

Robert Davis, subject of the second section, is a mathematics educator who sees the subject pragmatically rather than as an expression of logic. He sees the mechanism of learning as discovery rather than repetitive reinforcement, and has been influenced by Piaget, Papert and others. Davis sought to base his teaching on the everyday things about students, using such examples as sharing sweets to introduce aspects of mathematics.

Davis's system, Plato, was intended to be an environment in which students, through illustrated games and simulations, would discover mathematical concepts for themselves, and several of the games are described in detail by Solomon. Plato, unfortunately, has never lived up to its expectations, and what now passes as Plato on microcomputers is but a shadow of the original intention. Not only is the quality of presentation much reduced, but the old spectre of rote learning, so criticised by Davis, who has now left the project, has reappeared.

I felt a distinct sense of *deja vu* with the discussion of Thomas Dwyer. His book, *BASIC and the Personal Computer*, was where I started in the days of 16k, tape drives and Level II BASIC, and Solomon uses a number of the programs from the book in her discussion. Solomon sees much to be complimented in Dwyer's approach, which she describes several times as 'eclectic'.

Dwyer, along with Seymour Papert, Alan Kay, and many others, believes that the best computer learning experiences consist of learning to master the computer. Instead of waiting for new systems to be developed, or, as Papert and Kay did, develop new systems, Dwyer used the language already available, BASIC. His programs and books, along with those by David Ahl, Arthur Luehrmann and others, reflect an exploratory approach, using the computer to simulate a wide variety of situations. But of course BASIC is also Dwyer's undoing, and Solomon devotes several pages to a criticism of the language and the culture around it, concluding

'that his work is blighted by a *basic* inconsistency. There is an inconsistency in Dwyer's vision between the environment he wants and the tools with which he chooses to construct this environment. (p 100, and the emphasis (pun?) is Solomon's)

Subject of the fourth section is Seymour Papert. Solomon contrasts the views of mathematics of Papert and the others: Suppes, Davis and Dwyer see the subject as a body of material to be learned, although Davis is concerned with process. Papert proposes new content that emphasises computational processes rather than arithmetic skills. He sets out to provide an intellectual environment in which children can discover and construct new ideas, learning from their own experiences. Solomon proposes the 'Papert principle': 'If you want to teach arithmetic to children, arithmetic might not be the best route into these areas for an easy understanding of the topic. What is needed is a way of *mathematizing* the child; thereafter particular mathematical topics become easy.' (p 114)

Much of this section could be used as a tutorial for teachers setting out to use Logo as Solomon explains the philosophy and psychology behind the language, as well as the language itself. The final chapter is clearly designed for teachers and would-be teachers as she describes her views of learning and learning environments. Before ending with questions, Solomon describes her ideal teacher:

Out of experiences in this culture a new breed of teacher emerges: This teacher is thoroughly imbued with a coherent computer culture and its language. She knows how to use this language to talk interestingly about things people from outside the culture know and care about. This teacher has a fluent mastery of certain powerful ideas. She is thoroughly familiar with project terrains through which she will guide those who come for "instruction" (but will be given something better!). She has been there often! She knows how to observe people engaged in thinking, learning, puzzling, agonising, rejoicing She knows (and can only know this through experience) when to intervene and when to let the learner struggle. She believes that the key goal for any learner is to improve his image of himself as a learner, as an active intellectual agent. (p 160)

Both books are well written and are clearly the result of a great deal of careful study and research. Both have extensive bibliographies and are valuable and important contributions to the educational literature. One could almost say that in these books Logo has come of age.

Bibliographic Details:

Solomon, Cynthia *Computer Environments for Children: A Reflection on Theories of Learning and Education* MIT Press, 1986 ISBN 0-262-19249-7

Weir, Sylvia *Cultivating Minds: A Logo Casebook* Harper & Row, 1987
ISBN 0-06-046991-9

Apple Logo // and the ImageWriter

How to solve a problem the *long* way:

Like most printers, Apple's *ImageWriter* and its interface, the Super Serial Card, use sequences of control characters^{*} to control functions like type size etc. Certain functions can also be set with the DIP switches in the printer itself and on the interface card. One of those functions is whether or not the printer does a line feed after a carriage return when 80 characters have been printed on a line, and with the exception of one item of software there is no problem.

The exception is Apple Logo //. According to the manual, AL // 'treats the printer interface in the same way that Apple II Pascal version 1.1 does.' (p 301) Interesting, because Pascal 1.1 and 1.2 have never caused any problems. AL // does:

```
TO Demo
PR$UPER$S$E$1$U$S$A$D$1$A$N$D$T$H$E$I$M$A$G$E$I$W$R$I$T$E$R$. $how what happens with App
END
```

The manual says 'If text is being over-printed, set the printer to generate a line feed character after each line. If text is always double-spaced, reset the printer to *not* generate a line feed after a carriage return.' (p 304) So we set switch 1-8 in the printer and get:

```
TO Demo
PR$UPER$S$E$1$U$S$A$D$1$A$N$D$T$H$E$I$M$A$G$E$I$W$R$I$T$E$R$. $how what happens with App
END
```

Very helpful. The problem is with the interface, and to overcome it one normally sends a <CTRL>I C, which 'causes the SSC to generate a carriage return character automatically any time the column count exceeds the printer line width.' (*SSC Installation and Operating Manual*, p 17) Alternatively, one can POKE 1401,1. Well, one can do that from BASIC but AL // will have none of it. Control characters can get to the printer to control type size etc., but anything to the SSC is simply lost: PR WORD CHAR 9 "C or .DEPOSIT 1401 128. After trying all this, we gave up, and printed AL // files, from disk with a small BASIC program which first set up the printer with PRINT CHR\$(9);"C"

Some time ago there was reason to write to LCSI about a MacLogo problem and they were also asked about this one, but the letter became lost somewhere. Recently it surfaced, and Alain Tougas of the Technical Support section replied with this set of procedures:

```
to prettyprint :file
open 1
setwrite 1
local "limit
make "limit 78
open :file
setread :file
loop rv
setwrite []
close 1
close :file
end
```

^{*} 'Control' characters are those whose ASCII value is less than 32. They are normally typed with the <CONTROL> key, and control some aspect of the machine or program. <CTRL> C, <CTRL> G and the like are familiar Logo examples.


```

to loop :line
if equalp :line [] [stop]
if (count :line) > :limit [printlimited :line][(type :line char 13)]
loop rw
end

to printlimited :rest
repeat :limit [type first :rest make "rest bf :rest]
(type " " char 13)
if (count :rest) > :limit [printlimited :rest stop]
repeat count :rest [type first :rest make "rest bf :rest]
type char 13
end

```

So, it reads a file line by line, counts the characters, and prints <Return>s as required, although to TYPE CHAR 13 seems a bit odd. But we wanted to be able to print from workspace, not from a file. In short, that means that we have to write our own POTS, POPS and PONS, and the essential logic boils down to this:

```

assemble a list of the names in the workspace
work through the list
  if the name is the name of a procedure
    take its TEXT, add TO, its name, : on any inputs, and END
    change each line into a word
    print each line, adding <Return>s where necessary
  if the name is the name of a variable
    turn it into a word, with its name, the word 'is' etc
    print it, adding <Returns> where necessary

```

To assemble the list of names we look at the system object list, which in AL // contains all the primitives as well as our own words. The first system name is FIRST, so we stop there but add STARTUP because we will want its value. Along the way we have to reject the names of these procedures and their variables (:buried), and the initial input is the output from .CONTENTS:

```

TO CheckContents :contents
IF "FALSE = FIRST :contents [OP "STARTUP]
IF MEMBERP FIRST :contents :buried [OP CheckContents BF :contents]
OP SE FIRST :contents CheckContents BF :contents
END

```

Now we must find the words that are the names of procedures...

```

TO xPOTS :contents
IF EMPTY? :contents [OP []]
IF DEFINEDP FIRST :contents [OP SE FIRST :contents xPOTS BF :contents]
OP xPOTS BF :contents
END

```

...and think about printing them:

```

TO xPOPS :procedures
IF EMPTY? :procedures [STOP]
xPO FIRST :procedures
xPOPS BF :procedures
END

```

Life becomes interesting here. Given the name of a procedure, we want its TEXT, but we want more than that, we need its name, dots on any input, and END on the end⁸:

```

TO xPO :procedure
  xPOLoop xPOAux :procedure TEXT :procedure
  END

TO xPOAux :name :procedure
  IF NOT EMPTY FIRST :procedure
    [OP LPUT [END] FPUT SE "TO :name Dots FIRST :procedure BF :procedure]
  OP LPUT [END] FPUT SE "TO :name BF :procedure
  END

TO Dots :inputs
  IF EMPTY :inputs [OP "]
  OP SE WORD " : FIRST :inputs Dots BF :inputs
  END

```

The list of lists that is a procedure must now be printed, but as we do that we have to count characters so that the <Return>s can be put in place. To do that we'll need to change each line, each sublist, into a word, with square brackets (CHAR 91 and 93) where they belong:

```

TO MakeWord :object
  IF EMPTY "object [OP "]
  IF LIST FIRST :object
    [OP (WORD CHAR 91 MakeWord FIRST :object CHAR 93 MakeWord BF :object)]
  OP (WORD FIRST :object CHAR 32 MakeWord BF :object)
  END

```

It does put an extra space before]s, but never mind. xPOLoop is...

```

TO xPOLoop :procedure
  IF EMPTY :procedure [PRINT " STOP]
  PrintLine MakeWord FIRST :procedure
  xPOLoop BF :procedure
  END

```

...and we finally get to print, the whole line if it's short enough, otherwise one character at a time:

```

TO PrintLine :line
  IF EMPTY :line [STOP]
  IF (COUNT :line) > 75 [PrintLine PrintChars 1 :line STOP]
  PRINT :line
  END

TO PrintChars :column :line
  IF EMPTY :line [OP "]
  IF :column > 75 [(PRINT CHAR 32 "!) OP :line]
  TYPE FIRST :line
  OP PrintChars :column + 1 BF :line
  END

```

⁸ Although this is Apple Logo //, it has been formatted like the generic Logo to make it easier to read. Remember to type the IF..THEN lines as one line.

We can now turn our attention to free variables. The problem here, as with procedures, is with []s:

```
TO xPONS :names
IF EMPTY? :names [STOP]
LOCAL "name
MAKE "name FIRST :names
TEST NAMEP :name
IFTRUE [IF LISTP THING :name
        [PrintLine MakeWord (SE :name "is CHAR 32 CHAR 91 THING :name CHAR 93)
        [PrintLine MakeWord (SE :name "is CHAR 32 THING :name)]]
xPONS BF :names
END
```

At last, the main procedure, and the RECYCLE is there to clear extraneous words off the system object list, otherwise Fred, fred and FRED, all the same thing, would be printed:

```
TO xPOALL
LOCAL "names
RECYCLE
OPEN 1 SETWRITE 1
MAKE "names CheckContents .CONTENTS
xPOPS xPOTS :names
PR "
xPONS :names
CLOSE 1 SETWRITE []
END
```

We need a list of what mustn't be printed...

```
MAKE "buried [buried Dots Inputs PrintChars column line xPOAux name procedure
              xPOLoop xPO MakeWord PrintLine xPONS names xPOTS contents
              xPOPS procedures xPOALL CheckContents]
```

...and a :STARTUP to put it out of view (before you load or type other things into workspace):

```
MAKE "STARTUP [BURYALL]
```

It works (with the 75 in PrintLine and PrintChars changed to 65 for the occasion):

```
TO Demo
PR [This is just a long line of text to show what happens with !
   th Apple Logo //, the Super Serial Card and the Imagewriter. !
   ]
END
```

```
STARTUP is [ BURYALL ]
```

We haven't bothered with property lists, but that could be done if necessary. But it would have been *much* easier with PRINT WORD CHAR 9 "C.

STOPping in Style

Or: Why Logo procedures are arranged the way they are:

There is a body of opinion that considers a Logo procedure like this to be poor style:

```
TO PolySpi :size :angle :inc
  IF :size > 200 [STOP]
  FORWARD :size RIGHT :angle
  PolySpi :size + :inc :angle :inc
END
```

Why poor style, even though Logo interpreters are designed to efficiently implement tail recursion? Because we eventually make a call to the procedure which does nothing. Better, some argue, is to have the procedure call the new copy of itself only if the value of :size is less than the limiting value:

```
TO PolySpi2 :size :angle :inc
  FORWARD :size RIGHT :angle
  IF :size < 200 [PolySpi2 :size + :inc :angle :inc]
END
```

Well, one could argue that that is better and more conducive to a clearer understanding of recursion. In its possible favour is that it more closely approaches the style adopted in other programming notations, and the people suggesting a change are usually experienced in BASIC, Pascal, etc.

While that is true, that style is not used by the writers of any of the Logo texts or references: Abelson, diSessa, Lawler, Ross, McDougall *et al*, Harvey, Thornburg, Goldberg, Bitter, Marin *et al*, Hurley, Watt, etc., etc., neither is it used by the professional programmers who write the utilities and demonstrations supplied with Logo systems. The only one of those writers to discuss STOP in this way is Harvey (*Computer Science Logo Style: Intermediate Programming*, pp 64..65), and he invariably uses the normal Logo style. Writers such as Watt (*Learning with Logo*) emphasise the importance of devising compact and effective STOP rules. Abelson is on record elsewhere (*Structure and Interpretation of Computer Programs*) as being vehemently opposed to unnecessary logic and syntax.

I believe it only confuses the issue to insist on a style not used elsewhere in the Logo culture. Natural languages are embedded in their cultures and this is true also of computer languages, including Logo. Logo has a very rich culture, which we neglect at our peril.

We ought to be applying the razor of William of Occam, and minimising logic and code, not adding to it, but for the moment let's see where the avoidance of STOP leads us. In the case of a procedure like PolySpi2 there is only a little added, but with a procedure stepping through a list of data the final line becomes...

```
IF NOT EMPTY? BUTFIRST :data [Thingo BUTFIRST :data]
```

...as we have to test for emptiness before the object is empty. Alternatively, we could write:

```
IF (COUNT :data) > 1 [Thingo BUTFIRST :data]
```

and add yet other, less obvious, logic. We don't seem to have gained anything.

What of the fractal procedures so elegant in Logo?

```

TO Koch :size :level
  IF :level = 0 [FORWARD :size STOP]
  Koch :size / 3 :level - 1
  LEFT 60
  Koch :size / 3 :level - 1
  RIGHT 120
  Koch :size / 3 :level - 1
  LEFT 60
  Koch :size / 3 :level - 1
END

TO Koch2 :size :level
  IF :level > 1 [Koch2 :size / 3 :level - 1][FORWARD :size]
  LEFT 60
  IF :level > 1 [Koch2 :size / 3 :level - 1][FORWARD :size]
  RIGHT 120
  IF :level > 1 [Koch2 :size / 3 :level - 1][FORWARD :size]
  LEFT 60
  IF :level > 1 [Koch2 :size / 3 :level - 1][FORWARD :size]
END

```

Again, we don't seem to have gained anything, and it's unlikely that it really helps the understanding of recursion. In fact, the simple substitution of a recursive call for a FORWARD :size has been obscured by the IF..THEN..ELSE. As well, the level of recursion has been confused. Level 0 of a fractal/Peano curve, is the initiator (a straight line) and level 1 the generator (the actual shape), but that has been lost in the IF... test.

To be fully consistent, as we should, we will have to rearrange list manipulating procedures so that they do not make calls with empty inputs. First, the normal version...

```

TO Remove :item :object
  IF EMPTY? :object [OUTPUT []]
  IF :item = FIRST :object [OUTPUT Remove :item BUTFIRST :object]
  OUTPUT SE FIRST :object Remove :item BUTFIRST :object
END

```

...then the revised (which took 5 attempts make to work):

```

TO Remove5 :item :object
  IF :item = FIRST :object
    [IF NOT EMPTY? BUTFIRST :object
      [OUTPUT Remove5 :item BUTFIRST :object]
      [OUTPUT []]]
  IF NOT EMPTY? BUTFIRST :object
    [OUTPUT SE FIRST :object Remove5 :item BUTFIRST :object]
    [OUTPUT :object]
  END

```

Noone would argue that that is clearer and more meaningful than the normal version. Let's pseudocode the original:

```

to remove an item from an object
  if the object's empty, return an empty list
  if the item matches the first element of the object
    then return the result of removing it from the rest of the object
  otherwise, hang on to the first element, and combine it with the result
    of working on the rest

```

As they say in the classics, pseudocoding Remove5 is left as an exercise for the reader. One final look at Remove, in its real original form:

```
(defun Remove (item object)
  (cond ((null object) nil)
        ((eq item (car object))(Remove item (cdr object)))
        (t (cons (car object)(Remove item (cdr object))))))
```

LISP programmers have always used the most concise forms of logic (LISP is crisp, almost totally devoid of 'syntactic sugar'), and placed the most critical test *first*. That is not only a good, but a necessary model to follow.

There's something else. The style shown in PolySpi2 causes stack overflows in some Logo systems. Suggesting a scheme that may cause unexpected errors is hardly to be recommended.

There is no need to emulate the style of other languages in Logo. Rather, we might ask why the other languages don't have the power and conciseness we have in Logo.

Yea or Nay

Sometimes it's convenient to have a program stop for a yes or no answer:

Are you happy with that? (Y/N): _

Of course, there are several ways of doing it, some easier than others. Often, it saves time for the user if <Return> can be pressed alone instead of having to hunt for the Y. Here's a procedure that does it:

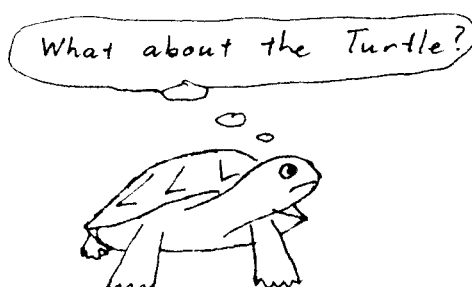
```
TO GetTruth
  OUTPUT IF MEMBER? READWORD (LIST " Y "y) ("TRUE)("FALSE)
END
```

It's a bit sneaky in its use of LIST, but you can't have a list like this [" Y y] by simply typing it that way. To use the procedure (The Δ represents a space to be printed.):

```
TO Happy?
  TYPE (Are you happy with that? \<Y\N\):Δ)
  OUTPUT GetTruth
END
```

You may have good reason not to take No for an answer, and this time we can use READCHAR instead of READWORD:

```
TO Yes!
  OUTPUT IF MEMBER? READCHAR (Y y) ("TRUE)(Yes!)
END
```



Turtle falls on man

HONG KONG — A man has been hit on the head by a fresh water turtle, which fell or was thrown from a high-rise residential building in the Wong Tai Sin district. He was not badly hurt.

Nested Polys

Adapted from the pages of *Turtle Geometry*

A commonly used example of a recursive procedure is this one:

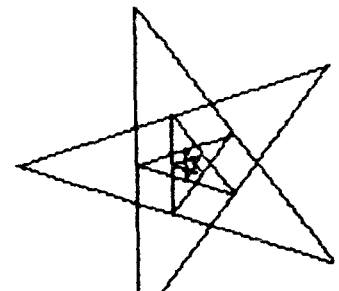
```
TO NestedTriangle :size
  IF :size < 10 [STOP]
  REPEAT 3 [NestedTriangle :size / 2 FORWARD :size RIGHT 120]
  END
```

With a couple of fairly large steps we finish up with these procedures. They are generalised so that they can draw any polygon besides triangles, but more than that, they add one polygon per level of recursion:

```
TO PolyNest :size :angle :level :totalTurn
  IF :level = 0 [STOP]
  FORWARD :size / 2
  SubPolyNest :size :angle :level
  FORWARD :size / 2
  RIGHT :angle
  PolyNestLoop :size :angle :angle "FALSE
  END
```

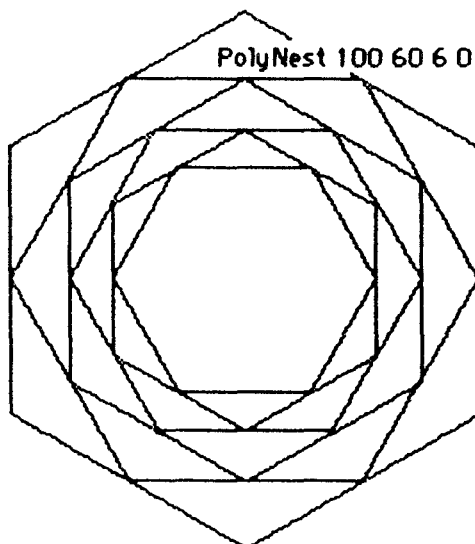
```
TO SubPolyNest :size :angle :level
  RIGHT :angle / 2
  PolyNest :size * COS :angle / 2 :angle :level - 1 0
  LEFT :angle / 2
  END
```

```
TO PolyNestLoop :size :angle :totalTurn :doneOne
  IF AND :doneOne 0 = ( REMAINDER :totalTurn 360 ) [STOP]
  FORWARD :size
  RIGHT :angle
  PolyNestLoop :size :angle :totalTurn + :angle "TRUE
  END
```

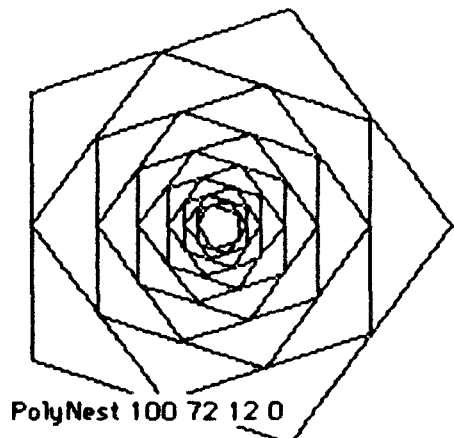


PolyNest 120 144 3 0

You can think about the need for :doneOne, and about the recursion involved. A couple of examples to start you off:



PolyNest 100 60 6 0



PolyNest 100 72 12 0

A Game with a name:

This program is just a bit of nonsense, but on the way we take a look at some interesting list processing. We read a name from the keyboard, but instead of printing it in its normal form print it reversed and then with the letters sorted into alphabetical order.

The toplevel procedure is easy enough although there are several things to be explained later. The `Δ` represents a space to be printed:

```

TO NameGame
  CLEARTEXT
  LOCAL "name
  PRINT [I'm the Logo Genie. Who are you?]
  TYPE [Please type your name:\Δ]
  MAKE "name READLIST
  PRINT "
  PRINT SE [Please to meet you,] Reverse :name
  PRINT [Oops! I think I had that backwards.]
  PRINT SE [Let me try again. You are] Sort MakeWord :name "
  PRINT "
  PRINT [Hmm, that isn't it either. I'm all]
  PRINT [confused. Perhaps you might see me]
  PRINT SE [again later,] :name
  WAIT 600
  NameGame
END

```

Now let's deal with the details. Totally reversing a list can be done several ways, but this is the usual one:

```

TO Reverse :object
  IF EMPTY? :object [OUTPUT []]
  OUTPUT SE (ReverseWord LAST :object)(Reverse BUTLAST :object)
END

TO ReverseWord :word
  IF EMPTY? :word [OUTPUT " ]
  OUTPUT WORD LAST :word ReverseWord BUTLAST :word
END

```

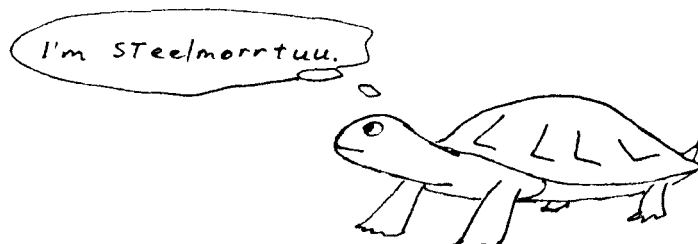
Reverse puts the words in the list into reverse order, while ReverseWord handles the characters within words. Use your system's TRACE facility to follow the recursion and the passing of values.

Before we sort the letters we must change :name from a list into a word, and you can think about the reason:

```

TO MakeWord :list
  IF EMPTY? :list [OUTPUT " ]
  OUTPUT WORD FIRST :list MakeWord BUTFIRST :list
END

```



Now we get to the interesting bit. The sort we're using is an Insertion sort, and it works by taking a letter at a time, working its way along the already sorted letters until it arrives at the right spot and inserting. As with Reverse and ReverseWord there are two procedures, but note that this time there is an input for what will become the final output. (Could Reverse and ReverseWord be written this way? Try it and see, or try rewriting Sort and Insert. You might also look up other sorting methods in a programming book.) In pseudocode to start:

to sort characters into an ordered word
if there's no character, output the word
otherwise insert the first character into the word and then deal with the rest

to insert a character into an ordered word
if there's no word left, output the character
if the character to insert comes in the alphabet before the first one in the word
then output the character and the rest of the word
otherwise, hang on to the first character of the word and work on the rest

In Logo:

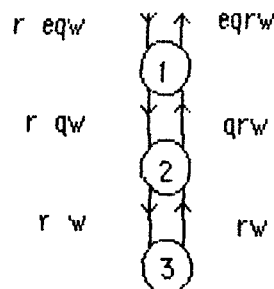
```
TO Sort :in :out
  IF EMPTY? :in [OUTPUT :out]
  OUTPUT Sort BUTFIRST :in Insert FIRST :in :out
END

TO Insert :in :out
  IF EMPTY? :out [OUTPUT :in]
  IF BEFORE? :in FIRST :out [OUTPUT WORD :in :out]
  OUTPUT WORD FIRST :out Insert :in BUTFIRST :out
END
```

If your system doesn't have BEFORE? as a primitive:

```
TO Before? :first :second
  IF (ASCII :first) < (ASCII :second) [OUTPUT "TRUE][OUTPUT "FALSE]
END
```

Again, you should TRACE how these work. Sort should be straightforward enough. Insert is perhaps more interesting, and here's a diagram of it inserting r into eqw, part of the process of sorting qwerty:¹



Now you can intrigue your friends.

¹ In diagrams like this the numbered circles represent calls to the procedure. Inputs are shown down the left hand side, outputs upwards on the right. As for QWERTY, that's a sad story...

Computing at Entropy House

Loose ends...

Dr Alan Kay is the person who coined the term *personal computer*, was the leading light behind machines that led to the Macintosh and its visually oriented operating system, the Smalltalk language, and numerous other ideas. He was recently the speaker at a meeting of A.P.P.L.E., the worldwide Apple user group, and discussed three types of learning: symbolic, visualisation, and reflex/motor. No prizes for guessing on which one schools place the greatest emphasis, even though it has been shown to be the weakest mode of learning.

One of Dr Kay's examples was a study:

'...done with the Logo turtle and three children; one age five, one age ten, and one age 15. Here each child was in a particular stage of learning development.

The youngest child learned best by touching and doing, the middle child learned by watching other's examples, and the oldest child was well into reading and studying. Each child was able to play with a mechanical turtle which could draw on a piece of paper. They were all given the same assignment, Make the turtle draw a circle.

The first boy told the turtle to move a little and turn a little, the way he would if he had walked out a circle with his body. The second boy noticed that if the turtle started at the centre of a circle, which he visualized on the paper, and then moved out to draw a series of points around the centre, the circle would be drawn. The third boy was unable to get the turtle to draw a circle. The controls would not accept $x^2 + y^2 = r$.

(K. Nemitz, 'Dr Alan Kay at the Pacific Science Centre', in *Call A.P.P.L.E.* January 1987, p 54)

Interesting. (Care to write the procedures?)

Coral Logo has appeared in one Australian supplier's catalogue, although it is not quite here yet and no price is listed (US price is \$79.95). As mentioned in the last issue, it is an 'object oriented' implementation. More on that when we actually have a copy for review. In the meantime, we know already that it is good for number crunching, has ready access to the Mac's Toolbox and is lexically, rather than dynamically scoped. What does that mean? Key in the following...

```
TO ScopeDemo
LOCAL "word
MAKE "word "dynamic
PrintIt
END
```

```
TO PrintIt
PRINT :word
END
```

```
MAKE "word "lexical
```

... and call ScopeDemo. PrintIt is called by ScopeDemo, and therefore has access to its locally defined variables. If Logo were lexically scoped, it would not have such access, and would print the value of the globally defined variable. Logo inherited its scoping rules from LISP, but there is now at least one lexically scoped LISP, the version named Scheme, which is also block structured like Pascal, with functions within functions. For more details of scoping, see Brian Harvey's books, or any LISP or Pascal text. It's always safest to call procedures with inputs. That way, there's never any confusion.

Two other new Logos are on the way. One is a version by Terrapin for the Mac. It supports multiple turtles, access to Mac QuickDraw routines, multidimensional arrays, property lists, strings, stream I/O etc. etc. The turtles can STAMP their shapes and can apparently be redefined. Price? \$US79.95.

The other is from LCSI for Apples, and to quote Alain Tougas: 'Apple Computers discontinued Apple Logo II (which was done by us and produced by Apple). This gave us the right to re-do it and we did so. It will be called *LCSI Logo II*. This version will look like Apple Logo II and will be sold by LCSI. It will have 30 Turtle shapes, a good shape editor and some bugs fixed. Finally, it will be available around mid-June...and will run on Apple //c, //e, and //gs'. Price in the LCSI catalogue is \$CDN139 for the single user version, and multiple copies and site licences are available. Wonder what it does in combination with SSC and ImageWriter, but we'll have to wait until a copy arrives.

Rumours about a version 2 of MacLogo are incorrect, although there is now a French version.

Latest to join the control Logo scene is Milton-Bradley, with their *Robotix* kit. It looks like something from long ago in a galaxy far away, but incorporates some very ingenious engineering. The parts fit together with octagonal studs, which give a useful choice of angles, are easy to put together and are quite strong. Fitted one way, the wheels are fixed to their axles, the other way about, they freewheel. There are four motors, but only one gear, and a hand controller. M-B make an interface for the Beeb (through user and printer ports), another is also available, and the whole thing can be driven by Logotron Logo.

The documentation includes ideas for class use and an account, with case studies, of research done with the system as part of MEP work in Welsh schools. It's not as flexible in some ways as the Lego or Fischertechnik kits, but would be an attractive and interesting way into control technology. Price? £50 for the kit itself, £80 for M-B's own interface, and no, it's not available in Australia yet, but Ralph Leonard is working on it. If you want to see it, give Ralph a call at Angle Park.

Advertisements:

Thinking Logo

An Introduction to [the Universe through] Programming

by Peter J. Carter

Ostensibly a text for the SSABSA Year 12 Computing Studies Course, *Thinking Logo* has information, techniques and ideas for anyone who wants to learn and use Logo.

Reviewers say: 'A delightful stroll through Logo.' (R. Green)
'...a clear, logical educational treatise... encyclopaedic...' (H. Hyde)

\$9 (\$10 posted) from the author, 28 Rowells Rd Lockleys 5032

BiKiLog

An informal group for Logo users to discuss, play with, argue about, and rave over Logo. BiKiLog is a SIG of the Computers in Education Group of South Australia Inc.

Venue: Microcomputer centre, Magill Campus SACAE, Lorne Ave Magill.

Next Meeting: Monday August 10 th, 7:30 pm