Massachusetts Institute of Technology
Artificial Intelligence Laboratory

A.I.Memo 652

Logo Memo 60
December 1981

SOME POWERFUL IDEAS

Robert Lawler

ABSTRACT

Here is a set of problem solving ideas (absorbed by and developed through the MIT Logo project over many years) presented in such a way as to be useful to someone with a Logo computer. With the ideas on unbound, single sheets, you can easily pick out those you like and set aside the others. The ideas vary in sophistication and accessibility: no threshhold, no ceiling.

The work reported in this paper was supported in part by the National Science Foundation under grant number 77-19883SED and by the Spenser Foundation. It was conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. The views and conclusions contained in this paper are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the National Science Foundation or the United States Government.

SOME POWERFUL IDEAS

Here is a set of problem solving ideas (absorbed by and developed through the MIT Logo project over many years) presented in such a way as to be useful to someone with a Logo computer. With the ideas on unbound, single sheets, you can easily pick out those you like and set aside the others. The ideas vary in sophistication and accessibility: no threshhold, no ceiling.

CONTENTS

collected by
Bob Lawler

WHAT'S A POWERFUL IDEA ?

Everybody knows what a grapefruit is and how to cut one in half. When you cut a grapefruit perpendicularly to the core, the cut face shows a pattern like a wheel, with axle, spokes and rim. A little energy and perserverance are all you need to dig out and enjoy the juicy meat from between the spokes. There must have been a time when you didn't know which way to cut a grapefruit. Did you find out how the hard way ? What happens if you divide the grapefruit the other way, along the core ? It is nearly impossible to get at the still buried meat, for the tough skin of the sections is an intact obstacle. The grapefruit looks pretty much uniform on the outside of its skin, but when you look deeper you can see there is a very specific and important organization that you must understand if you want to get at what's inside. This very simple, very concrete situation of cutting up a grapefruit provides a useful way to look at many other very troublesome problems. It points up the issue that how you analyze a problem, whether your analysis "goes with the grain" or "against the grain", can make a world of difference in how hard the problem is to solve.

The intrinsic thing that gives such an example power in thought is its simplicity, in the sense that given the perspective of the idea (embodied in the concrete example), the primary conclusions that can be drawn are obvious without long chains of arguments. That is, such ideas are elegant in the mathematician's sense. The extrinsic root of power is the example's fruitfulness, how well it can serve in helping you understand other problem situations by analogy.

The more powerful ideas you have the better. If you have only one possible model of a situation, you are a prisoner of a limited point of view. If you can interpret a situation in terms of several possible models, you can compare the fit of each to judge which is most appropriate. As a situation changes, some alternative model may come to better fit the situation than the one originally best. Could it be that the flexibility of mind we ascribe to "smart" people derives directly from their having a well developed stock of such powerful ideas ?

Descartes introduces his Discourse on Method with the witticism that intelligence in the only thing in the world of which there is no scarcity, for although men may complain that they lack goods or information, no one complains that his judgment is inadequate. Why should this be the case ? People don't know -- or believe -- that ideas can be powerful. The most powerful idea of all is that there are such things as powerful ideas. Those of us who lack this idea may struggle valiantly with problems, combing the world for information and our minds to make connections, without imagining that we need to formulate the problem through a fundamentally different model of the situation.

DO IT YOURSELF

How do you make a Logo circle ? Circles have something to do with centers and a thing called a radius, don't they ? But the Logo turtle doesn't know anything about centers or a radius -- all it knows is how to turn right and left and to move forward and back. What would YOU do if you were the turtle ? You might do it yourself, try to make a shape that's a little bit like a circle by taking one step forward, turning a little, and doing both over again and again. If the path you follow has many turns, the shape of your path will be like the circle the turtle makes in this procedure:

```
TO CIRCLE
FORWARD 1
RIGHT   1
CIRCLE
```

When you do it yourself, you can understand better what the turtle is doing.

Suppose you made a picture of a bird and wanted to make it fly. How would you do it ? Can you put yourself in the scene and write a "people-procedure" for your own flying ? It must have something to do with flapping your arms and going forward. If you do it again and again, as in the circle example, your simplest people-procedure might be like this:

```
TO FLY
RAISE ARMS
LOWER ARMS
FORWARD SOME-AMOUNT
FLY
```

Does it seem childish to do it yourself ? Grown up people never go around imitating turtles or pretending to fly, do they ? Some <u>really</u> smart people do; they know that putting themselves in the scene, being a part of the action, helps them understand what's going on. One of the most famous scientists of modern times, James Clerk Maxwell, once imagined himself a super-ordinary tiny creature (he named it a "demon") who could control the flow of molecules in a gas by opening and closing a tiny window. Maxwell was not being childish by putting himself in the middle of the action. He was raising a fundamental challenge to the basic laws of physics. Give it a try when you have a problem to solve. Do it yourself.

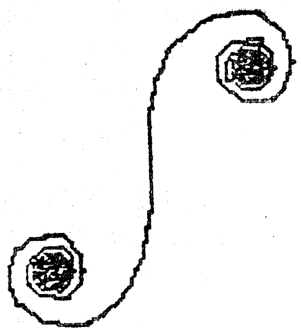
MAKE IT YOUR OWN

A six year old was introduced to the INSPI procedure below. When she saw it executed with these variable values, [INSPI 10 0 1], she declared that it looked like a "seahorse" and asked if she could have a seahorse procedure of her own. (I coded for her the equivalent SEAHORSE procedure.) Her "SEAHORSE" procedure was one she explored with considerable interest and satisfaction for several days, delighting in the pretty designs she discovered:

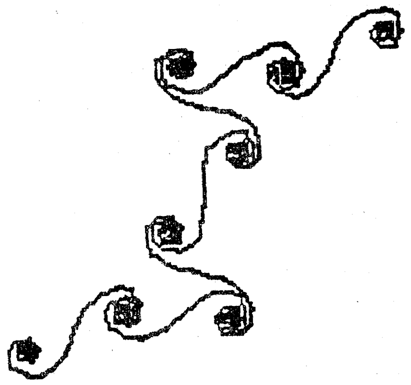
```
TO INSPI  :DISTANCE :ANGLE :CHANGE
1: FORWARD :DISTANCE
2: RIGHT   :ANGLE
3: MAKE  "ANGLE (:ANGLE + :CHANGE)
4: GO  "1
```

```
TO SEAHORSE  :CHANGE
1: MAKE "ANGLE 0
2: FORWARD  10
3: RIGHT   :ANGLE
4: MAKE  "ANGLE (:ANGLE + :CHANGE)
5: GO  "2
```

Seahorse 1



Seahorse 7



It was important to this child to make the procedure her own because it put limits on how much complexity she had to cope with. She didn't have to worry about other people's procedures or how they related to hers. She created her own little world where she was free to explore and develop her own ideas.

Making things your own limits the problems you face at one time. It also serves to make you more independent. You can use your own procedure whatever way you want. You can become as certain as you need to be about how your procedure behaves. You can change your procedure without doing something unexpected to anybody else. But greater independence has its problems. You may get frustrated when your procedure doesn't work the way you think it should; you may have a harder time explaining your own procedure to somebody when you ask them for help. Even if you begin Logo by copying others' procedures and changing them to be exactly what you want them to be, you will soon go from changing other people's procedures to making your own procedures from scratch.

CAN I MAKE LOGO COMMANDS ?

Any procedure you write is a new Logo command. In this first sense, the answer is "yes" and obviously so. On the other hand, if you write a procedure whose only function is to print "FOO", that would clearly be of a different stature from the Logo primitves, such as RIGHT and FORWARD, mainly because it won't do much for you. Do you need more "Logo commands ?" Do you have enough experience to know what you need ? Hardly anybody can answer "yes" very positively to such difficult questions as these. The way to find out if you need more Logo commands is to look at what you do and the procedures you've coded. Ask yourself: are there sequences of repeated instructions in procedure after procedure ? Are there tests of data conditions applied time after time ? There probably are. Here is an example of a command you might find a valuable addition to Logo:

PURPOSE: when programming with lists, most frequently iteration is terminated by testing a data condition; for example, the list is empty so no more processing can be performed on its members. It would be useful to have a predicate which directly reports whether a list is empty or not. Such a one could be encoded this way:

```
TO EMPTY?  :LIST
IF :LIST = [ ]  OUTPUT "TRUE
OUTPUT "FALSE
END
```

PURPOSE: when programming with lists, very often you want to know if a specific item is a member of a list. The MEMBER? predicate will answer that question.

```
TO MEMBER?  :ITEM  :LIST
IF EMPTY? :LIST OUTPUT "FALSE
IF :ITEM = FIRST :LIST OUTPUT "TRUE
OUTPUT MEMBER?  :ITEM  BUTFIRST :LIST
```

The gain in your programming from using such a new command may appear a small one, and so it is. But small gains add up to significant advances. Easy extensibility is a key feature of the Logo language. Using that feature to simplify your own coding will enable you to code more complex procedures and understand them better.

HOORAY FOR BUGS !

Making a square with the turtle is pretty easy, FORWARD 100, RIGHT 90 and do it again and again and again. The simplest procedure does it this way, using "recursion" (the third line, "SQUARE" means perform the entire procedure again):

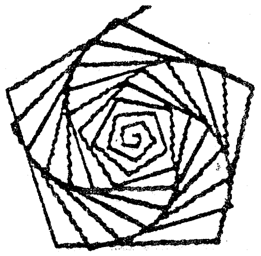
```
TO SQUARE
FORWARD 100
RIGHT   90
SQUARE
```

A procedure to make a little square would start with the turtle going forward some smaller distance, such as ten turtle steps. One little change can make a square maze grow out of the little square. You can figure out just what the turtle will draw with this procedure:

```
TO SQUARE.MAZE   :DISTANCE
FORWARD :DISTANCE
RIGHT  90
SQUARE.MAZE   :DISTANCE + 5
```

When the turtle turns RIGHT 90, the maze is square. How much should the turtle turn to make a six-sided maze ? ("Sixty degrees," you say ? That's right.) How much to make a five-sided maze ? Five is halfway between four and six. Because 75 is halfway batween 60 and 90, that would be one good guess for how many degrees to turn.

```
TO FIVE.MAZE   :DISTANCE
FORWARD :DISTANCE
RIGHT   75
FIVE.MAZE :DISTANCE + 5
```



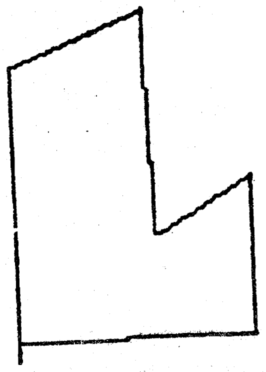
Would you say that using '75' is a mistake because it does not make exactly what you hoped ? If so, you have to be willing to see that mistakes can be good things. When the result of a procedure turns out different from what was expected, programmers say the procedure has a "bug". But sometimes the surprising result is a better one than what you first intended. That's a "new discovery" bug, one of the best kind.

Any bug, something which makes your procedure do the unexpected -- if you bother to fingure it out -- leads to an increase in your knowledge. Although a bug may hinder your objective, the bugs of your procedures will be the best guidance you can get of what to learn to master the Logo programming environment. If bugs can lead you into new discoveries and give good guidance on what to learn, this suggests a new way a teacher or advisor could be helpful to you. Such an advisor could help in exploring and understanding the difference between what you expected and what the computer did. If you, or anybody else, want to know how you're coming along with the computer, the best indication of progress in understanding is a record of the bugs you have encountered, and understood, and those you are still working on.

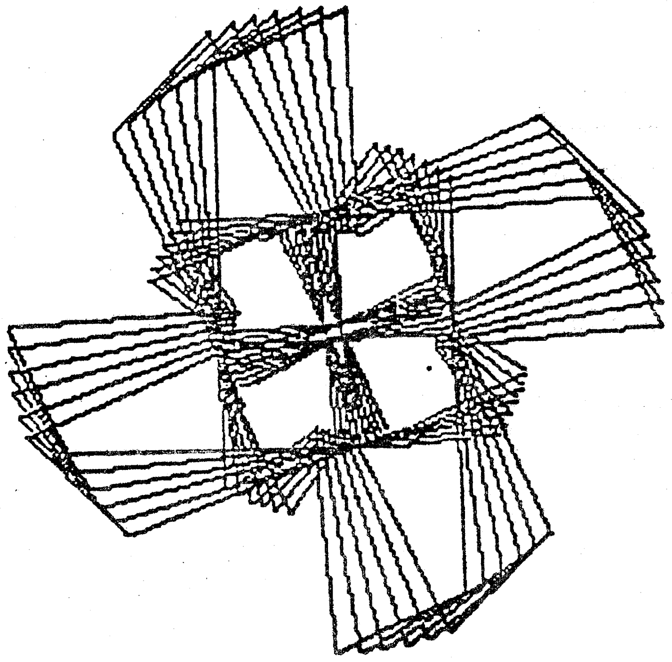
DO IT AGAIN

A ten year old girl met the Logo language for the first time. She didn't have any idea of what to do or what could be done. She started to draw with the turtle, making it go forward different amounts and turn various numbers of degrees. Sometimes she didn't like the latest line the turtle put on the screen and wanted to erase it. The turtle didn't have an eraser, so she used a clever trick: whenever she liked an addition to her drawing, she would code it into a procedure she wrote as the drawing developed; when she didn't like a change, she would clear the screen and execute the procedure, that way re-creating the drawing as it looked before the latest change. She completed here effort with a drawing that looked a bit like a boot, as you see below:

"Boot"



25 "Boots"



The girl had given Logo a fair chance, worked dutifully, had some good ideas, but she was basically bored. "Is that all the turtle can do ?", she asked. For no very good reason, she executed her "BOOT" procedure again. The second boot drawn over the first was rotated through a large angle. She did it again, and again, and again, becoming caught up in the design that was emerging from repeating her "BOOT". Finally, she cleared the screen and executed "BOOT" twenty five times in rapid succession under control of the REPEAT command. She was thrilled with the design above which emerged from her doing it again.

SUMMARY

Repeating a specific list of commands can have interesting results if the executions "add up". The ways to do it again are by re-keying, naming the commands as a procedure and rekeying the name; executing a named procedure under the scope of a repeat command; recursive invocation; and looping.

THINKING ABOUT VARIABLES

Variables are names which have values assigned to them. A good first way to think about variables is as little boxes, say the kind that are used to keep wooden matches in. Computer memory is made of of these little boxes, each of which may or may not have something in it. Naming the boxes is a good way to keep track of specific boxes into which you put things. When you ask what's in a specific box, for example, :BOX2 (read as "dots Box 2"), you don't change what's in the box. When you use the MAKE command, you always CHANGE the contents of the box. You could think of it as emptying the box before packing a new thing in it.

A second important way to think about variables comes from their use as inputs to procedures. You can imagine a procedure as a list of instructions to be performed by a lazy little man. The little man knows how to do what the steps of his procedure specify. He sleeps whenever he is not doing his procedure. He wakes up when somebody calls his name; then he does what he knows how to do and goes back to sleep. The little man never sees other people, but he can get mail in his mail box. This is necessary because he sometimes needs a message to specify exactly how he should execute a command. The message in his mail box when he wakes up is the value of the input variables he needs to perform the steps of his procedure.

A third important way to think about variables comes from their use in controlling the repetition of procedures (whether by looping or by recursion). Here are two ways to draw a square with iteration:

```
TO SQUARE1
1: MAKE "COUNT 4
2: IF :COUNT = 0  STOP
3: FORWARD 25  RIGHT 90
4: MAKE "COUNT (:COUNT - 1)
5: GO "2
END
SQUARE1
```

```
TO SQUARE2  :SIDES.LEFT
IF :SIDES.LEFT = 0  STOP
FORWARD 25  RIGHT 90
SQUARE2  (:SIDES.LEFT - 1)
END
SQUARE2 4
```

Executing SQUARE1 or SQUARE2 4, the turtle passes over each side one time. With the variable some other value, the turtle would trace that many sides of a square. In general, such iteration count variables control execution of the steps within a procedure's boundaries.

SUMMARY

variables - permit indirect reference to values which may change -- as one may refer to the contents of a box by naming the box.

input variables - permit specification of values for use by operations within procedures refering to variables -- one may think of them as messages needed by the little man who executes the procedure steps.

iteration variables - control the number of times procedure steps are executed within their iteration boundary.

DO SOMETHING A LITTLE DIFFERENT

One of the basic procedures most people work out when starting Logo is a procedure to make a square. If you're willing to stop the turtle with "control G", this procedure will do quite nicely:

```
TO SQUARE
FORWARD 100
RIGHT   90
SQUARE
```

If you want to do something a little different, you might pick out a command operand, such as 100 and turn it into a variable. Doing so would permit you to makes squares of any size.

```
TO SQUARE :DISTANCE
FORWARD :DISTANCE
RIGHT   90
SQUARE :DISTANCE
```

If you want to do something a little different, you might consider changing the value of distance in every invocation of SQUARE. You would have a SQUARE.MAZE procedure:

```
TO SQUARE.MAZE :DISTANCE :CHANGE
FORWARD :DISTANCE
RIGHT   90
SQUARE.MAZE (:DISTANCE + :CHANGE)  :CHANGE
```

If you want to do something a little different, you might look at the operand of the second command in the procedure and turn that '90' into a variable. You would then have what's become known as a POLYSPI procedure (can you find some of the many good numbers for angle ?):

```
TO POLYSPI  :DISTANCE  :ANGLE  :CHANGE
FORWARD :DISTANCE
RIGHT  :ANGLE
POLYSPI  (:DISTANCE + :CHANGE)  :ANGLE  :CHANGE
```

If you want to do something a little different, you might think of applying the change value to the variable "ANGLE" instead of to "DISTANCE". You would then have what's been called the INSPI procedure (be certain to try INSPI 5 0 7):

```
TO INSPI  :DISTANCE  :ANGLE  :CHANGE
FORWARD :DISTANCE
RIGHT  :ANGLE
INSPI  :DISTANCE  (:ANGLE + :CHANGE)  :CHANGE
```

If you want to do something a little different, you might ask yourself about symmetrical versions of the POLYSPI and INSPI procedure. Or ask yourself <u>why</u> the procedures generate the attractive designs they make. Or ask if you can apply in other places the idea of isolating some single element of a procedure and changing it to create new things and to understand them.

WHAT GOOD IS PLANNING ?

The first view is that planning breaks a problem up into parts, each of which can be more simply solved than can the whole. For example, if you wanted the turtle to draw a picture of a house, you probably would find it easier first to write a triangle procedure for the roof and a square for the storey then put the two together than you would composing a procedure for drawing the whole thing at once.

A richer view of planning is that the breaking up of a problem is very fruitful -- because you will create partial solutions which can be used in different ways to make other things. Let's extend the "HOUSE" example. The simplest extension of a HOUSE would be to separate the parts from each other and reconnect them a different way. Doing so you could make a "WISHINGWELL", such as the one drawn by this procedure:

```
TO WISHING.WELL
RIGHT 180  STOREY
RIGHT 180  FORWARD 100
ROOF
END
```

This attempt to use the parts of a HOUSE as parts of a WISHING-WELL runs into an immediate problem: the WISHING-WELL is bigger than the house ! If you want both of them in the same picture, it would be necessary to make another triangle and square procedure for a small wishing-well. At such point, it makes sense to generalize the ROOF and STOREY sub-procedures, specifying their size by the use of input variables, as in the procedure below:

```
TO STOREY  :SIDE
LEFT 90
FORWARD :SIDE/2
RIGHT 90  FORWARD :SIDE
RIGHT 90  FORWARD :SIDE
RIGHT 90  FORWARD :SIDE
RIGHT 90  FORWARD :SIDE/2
RIGHT 90
END
```

```
TO ROOF  :SIDE
LEFT 90
FORWARD :SIDE/2
RIGHT 120  FORWARD :SIDE
RIGHT 120  FORWARD :SIDE
RIGHT 120  FORWARD :SIDE/2
RIGHT 90
END
```

Not only are these new procedures more flexible. The way they fit together can now be modified to make a better house than the original !

```
TO HOUSE.WITH.EAVES
RIGHT 180  STOREY 100
RIGHT 180  ROOF 120
END
```

PASSING THE BUCK

Sometimes hard problems can be simplified by doing a small part and "passing the buck". This worked example is to clarify the idea and the Logo techniques for applying it. Suppose d you want to print messages in a code where every word is spelled backwards, e.g. sdrawkcab. How can you write a procedure to switch letters around? You know the procedure begins with a title line and a variable input, such as:

```
TO SWITCHEM :INPUT
```

If the input to SWITCHEM has no letters, nothing should be printed -- maybe a space. If the input is only one letter long (such as "I" or "a"), it should be printed. If the input is longer, you will always want to print the last letter of the input anyway, so you might as well do that and pass the buck (all those other letters except the last one) to another procedure, call it HARDER:

```
TO SWITCHEM :INPUT
IF EMPTY? :INPUT PRINT SPACE STOP
TYPE LAST :INPUT
HARDER BUTLAST :INPUT
END
```

Now, what should the HARDER procedure be like?

If the input to SWITCHEM was one letter long, the input to HARDER will have no letters -- it should stop. If HARDER's input is one letter long, that letter should be printed. If longer, you will want to print the last letter of the input anyway, so you might as well do that and pass the buck (all those other letters of HARDER's input except the last one) to another procedure -- call it EVEN-HARDER.

Doesn't that sound familiar? EVEN-HARDER will have to do the job that HARDER was supposed to do. HARDER does the same thing as SWITCHEM. The good trick in passing the buck is _you never have to write the HARDER procedure_ if SWITCHEM calls itself:

```
TO SWITCHEM :INPUT
IF EMPTY? :INPUT PRINT SPACE STOP
TYPE LAST :INPUT
SWITCHEM BUTLAST :INPUT
END
```

THE CLEVER HACK AND CLEVER TACTICS

Two children played a simple Logo game, SHOOT. In that game, the turtle first draws a circle on the video display then, after lifting the pen, sets the turtle down at a random screen location. The objective of the game is to turn the turtle with RIGHT and LEFT commands until it points at the target then SHOOT the turtle forward into the target. When SHOOT is executed, first it moves the turtle forward the specified number of turtle steps. SHOOT next computes whether or not the turtle has landed within the circumference of the target. If so, a point is scored, the screen is cleared and a new round begins. Otherwise, the turtle is returned to its initial location and orientation.

This is a simple, low pressure game, used to familiarize new Logo people with the commands of the language. But with these two children taking turns at one terminal, the game quickly became competitive. It became important to score every time SHOOT was executed (they counted SHOOT executions as the basis of turn taking). One child noticed that the turtle always drew the target at the center of the screen. He also knew that the HOME command puts the Logo turtle at the center of the screen (and thus at the target center). He proceeded to score every time with the command sequence [HOME SHOOT 0], despite the outraged complaints of cheating from his opponent. This solution to the SHOOT problem is a clever hack. A "hack" is an accidentally effective way of getting around a particular problem.

The child's clever hack was easy enough to render ineffective. Some one else had only to change the game so that the turtle drew the initial target at another location for the clever hack to become worthless in itself. And yet, this clever hack served well as an example of a more general form of solution the child developed. He developed what he called a "clever tactic". Knowing that the SETHEADING command could point the turtle in a specific direction, he used SETH 0 then moved the turtle forward or back as necessary to align it horizontally with the target. A RIGHT or LEFT 90, with more forward and back commands would always then put the turtle within the target and permit SHOOT 0 to bring a certain score.

SUMMARY

It is useful to distinguish between specific solutions to a problem in a particular circumstance and general solutions to all problems of a given class. Never despise the particular solution, however, for it can show the way to a more general and more powerful solution.

ADVICE TO A TEACHER

I write here about my own experience and out of that experience, but my situation has been different from yours. You've had to worry about instructing twenty or thirty children. I have merely had to play with two children -- and those children were my own and I knew them well. I write here also with the conviction that your work in the future will be more like my experiences than it has been. Computers will permit the construction of intellectual worlds where children will be able to spend much time learning effectively on their own. This will give you more time to know individual children and to intervene in their learning as the advisor you, their parents, and the children themselves hope you will be.

Geometry has been an important central theme of instruction in our laboratory because its founder invented a kind of geometry for children. We've called it "turtle geometry". It is distinguished from other geometries because it is a geometry of action. The leading actor, the agent of this action, is "the turtle". Either a mechanical robot or a triangular cursor on a video-display screen, the turtle goes forward some distance or turns through some angle on command. When its pen is down, the turtle draws a line. At their ages of six and eight, I introduced my children to SHOOT, a simple turtle geometry game. A setup procedure drew a target and placed the turtle at some random screen location. To score, the children had to turn the turtle right or left some angle to point it at the target then SHOOT forward some distance into the target. The game was easy for them to play and they enjoyed it. (They even played the game without the computer; setting a hula hoop on the floor for a target, the children took turns playing turtle and keyboard commander.)

Robby, the older child, came to want a more complicated game. He was fascinated by the air battles of World War II and asked me to make a game where the targets would be airplanes. READY-AIM-FIRE (we called it R.A.F.) satisfied him; even more, it engaged him. Robby spent the better part of an entire day trying to score more kills than von Richthofen, the famous Red Baron of World War I. This game permitted him to do what he wanted -- play in his own fantasy world. It permitted me to introduce him to absolute coordinate geometry. The AIM procedure required specification of the airplane's location through naming its X and Y coordinates. (Axes provided a scale from which these values could be read off.) After the plane's location was specified, the AIM procedure moved the "gunsight" to the location. AIM could be executed as many times as necessary to get the gunsight on target, where FIRE would destroy the plane and increase the score. When Robby later wanted a similar game for sinking ships, I showed him how to modify the R.A.F. procedures so that he could replace the gunsight with a SUB and the airplane with a CARRIER, both simple drawing procedures he made.

The style of introduction presented in this story is opportunistic in the extreme. In depends on three things: the initiative of the child to connect his computer activities with what he knows about other things that concern him; the flexibility of computer systems to enable the building of simple models; the knowledge and values of a teacher in shaping particular procedures through which the child's objectives are achieved in such a way that the child is introduced to important ways of looking at and describing the world.

COMPUTERS AND PEOPLE

Is playing with computers good for children ? Couldn't it be bad ? Might they not begin to think of themselves and others as machines ? Here is a story about how my daughter, Miriam, pretended to be a machine -- the Logo turtle -- and what she made of that.

This night is the last night of summer, so defined by the children's having to begin school on the morrow. Over the summer they have gradually become accustomed to going to bed late, and now, in order to rise early, they should go to bed early. No one found this argument convincing. We negotiated a compromise that the children get into pyjamas, return for dessert (delayed by conversation with dinner guests, Jose and Fernando), and then go off to bed. Robby lived up to the agreement; Miriam would not.

When given a direct order to go to bed, she went to my bed instead of hers. I had mentioned during dinner the children's inclination to play turtle. Fernando tried to help. "Miriam, FORWARD." She did nothing. I advised him that he had omitted the carriage return. Upon his "carriage return" Miriam complained, "You haven't told me how far to go", chuckled, and popped back onto my bed. Gretchen attempted "FORWARD 100, carriage return." With the gripe "You haven't told me how to FD100" still in the air, I described this bug as the well known space omission between command and input. Fernando was then precise: "Miriam: FORWARD, space, 100, carriage return." Miriam played fair and proceeded stepwise (counting each step) down the length of the loft. At first, we expected 100 steps to be too few. Miriam counted "70" in the kitchen and at "88" gleefully announced "Out of bounds" as she walked into the wall in the hallway. While so close to her bed room, she picked up her 'security blanket' (the air was a little chilly) and came skipping back into the living area.

The game wore on (hide turtle under the blanket, and so forth), after a while became wearing, and I directed her to bed with the threat of physical force. Miriam replied, as she has for some months now, with the counter-threat "I'm quitting your research, Daddy, I really am." Having thus preserved her dignity, she acquiesced to the demand that she go to bed.

In this incident and many others, Miriam showed that this robot-role which she was willing to adopt was one she subjected utterly to her ends as a person. Playing turtle was an enrichment of her repertoire, not a constraint upon it. As we paraphrase William Blake:

Tools were made; born were hands.
Every child understands.

<u>SINGLE-KEY INTERFACES</u>

Young children, especially those who have never used a typewriter -- and even more so those who have not yet learned to read -- can have a lot of fun with turtle geometry if using the keyboard is made simple for them. One obvious and simple way to do so would be to make an "interface" for their use. An INTERFACE translates what someone keys into Logo commands and then executes those commands. Many such have been made in the past, and surely more will come. Typically, the simplifications are in reduction of the child's keying burden to a single character for any action desired. For example, when the child keys the single character F, the interface translates this into a command "FORWARD 20" and executes it.

Many computer based games depend upon the speed of reaction of the person playing the game. Consider a real-time game where you have to fire a rocket to change the trajectory of a space ship. Keying "FIRE" and "return" could take so long the result of the force would be different from what you wanted. In contrast, keying a single letter whose value was encoded to mean "FIRE" could be effective at once in changing the space ship's speed.

Have you ever wondered why the letters on the typewriter keyboard or the computer terminal are where they are ? Alvin Toffler relates the history of keyboard development, pointing out that the keyboard arrangement was made difficult on purpose, to slow people down so that their high speed keying would not jam the originally slow-moving mechanical linkages of early machines. Would you like to make your own keyboard arrangement ? You can. All you need is an interface which will substitute your characters for those wired at the keyboard, and a set of sticky labels to show how the interface will assign meaning to the keys struck.

SUMMARY

Writing an interface which reads one character at a time is a primary way of shaping your computer environment to be what <u>you</u> want it to be like. You can make is simpler, more responsive, you can even change the meanings of individual keys.

MICROWORLDS AND LEARNING

The central problem of humane education is how to instruct while respecting the self-constructive character of mind. Teachers face a terrible dilemma in motivating children to do schoolwork that is not intrinsically interesting. Either the child must be induced to undertake the work by promise of some reward or must be compelled to do the work under threat of punishment. In neither case does the child focus his attention on the material to be learned. The work is seen as a bad thing because either it is an obstacle blocking the way to a reward or it is a cause of the threatened punishment.

Psychologists know that -- however much insights do occur -- much learning is a gradual process, one of familiarization, of stumbling into puzzlements and resolving them by proposing simple hypotheses in which a new problem is seen as like others already understood, and performing very simple experiments to test the latest "theory".

Microworlds can be seen as worlds designed for virtual, streamlined experiences, worlds with agents and processes one can get to know and understand. Properly designed microworlds embody a lucid representation of the major entitites and relations of some domain of experience -- geometry and music and two examples -- as understood by experts in the domains. This is where the knowledge of the culture is made available, in the very terms in which the microworld is defined.

The child's appropriation of that knowledge is made possible by the microworld not being focussed on problems to be done, but on "neat phenomena" - i.e., the primary manifestation of the power made available by knoweldge about the domain. If there are neat phenomena, then the challenge to the knowledgeable expert is to formulate so crisp a presentation of the elements of the domain that even a child can grasp its essence. The value of the computer is in building the simplest model which an expert can imagine as an acceptable entry point to his own richer knowledge.

If there are no neat phenomena that a child can appreciate, there is no function that knowledge of the domain can serve for him. He should not be expected to learn about it until he is personally engaged with other tasks which will make the specific knowledge tolerable as a supporting prerequisite to something desirable to know.

THE IDEA OF A FORMALISM

A formalism is a set of symbolic objects that are related by the operations or manipulations that can be performed on them. Everyday arithmetic is an example of a formalism: the numbers are related to one another by addition, subtraction and so forth. We often use formalisms, such as arithmetic, without asking what there is about them that really makes them useful in thinking. The mathematician-philosopher Whitehead raised this question about the calculus, another formalism, and proposed an answer of the following sort. A formalism is useful because it gives you one less thing to worry about. You learn a set of rules of almost mechanical manipulation -- then you can concentrate on how to apply them to a specific situation you want to know more about. You judge the applicability of a mathematical formalism by whether or not its predictions correspond to what happens in the problem domain.

A programming language such as Logo is also a formalism -- but one whose focus is more on its concrete use than on its symbolic prediction. In this sense, the Logo language is a kind of empirical mathematics, one whose value does not depend upon immediately mastering perfectly a set of rules. One can begin with a faulty procedure and perfect it by debugging -- retrying the execution until it produces the intended result or some better one discovered along the way. Eventually one may become sufficiently expert to compose perfect code, but it is not necessary that one ever do so.

A relaxed requirement for perfection is one major way that Logo programming contrasts with the child's other experienced formalism, arithmetic. This is important because in the world of turtle geometry, the domain of design is so rich that unintended results can often be more attractive than what the programmer first intended. This is a direct contrast with arithmetic -- where errors are of positive value only to psychologists. There is a second sense, however, in which Logo programming requires perfection as much as any other formalism. When one is committed to a specific result, specific operations must be performed in the correct order to achieve that result. Because of the relative richness of the error paths in turtle geometry, Logo may be a more accessible formalism -- because a more attractive one -- than children commonly met before the advent of computers.

DIFFERENT KINDS OF VARIABLES

If a variable is defined outside of a Logo procedure, its value can be changed by keyed commands or by executing any procedure which refers to it. Such a variable is called a GLOBAL variable. Now, if you store something in a box, generally you would like it to remain there until you change the contents of the box. You can't count on the contents of a global variable unless you take special care to guard against unexpected references. One way it to give your variables unusual names, e.g. [MAKE "GRANDMOTHER'S.SHIN.BONE 3]. The reason not to use UNIQUE variables is the difficulty of remembering what name you assigned. A second technique is to "initialize" every variable in every procedure before you refer to it. Doing this becomes a little tedious when you write lots of procedures. A third method is to use local variables.

LOCAL variables are defined only within the context of the procedure which references them -- so no procedure or keyboard entry can alter the value of another procedure's local variables. Further, local variables exist only within a specific execution (or "instantiation") of a defined procedure. This convention of the Logo language (and a number of others as well) is central to the use of input variables (and others) in recursion. Consider the procedure below:

```
TO SQUARE :SIDES.LEFT
IF :SIDES.LEFT EQUAL 0 STOP
FORWARD 25  RIGHT 90
SQUARE :SIDES.LEFT - 1
end
```

When you key SQUARE 4, the Logo interpreter creates an instantiation or copy of the SQUARE procedure for execution. Let's refer to it as 1-SQUARE. The value of the corresponding variable 1-:SIDES-TO-GO is 4. When the third line of 1-SQUARE executes, the Logo interpreter creates a second copy of SQUARE; call it 2-SQUARE. What is the value of the corresponding variable 2-:SIDES-TO-GO ? The answer is three. Executing 1-SQUARE, the Logo interpreter evalates 1-:SIDES-TO-GO as 4 and subtracts one from it, then assigns 3 as the value for the variables 2-:SIDES-TO-GO. In successive recursions of SQUARE 4, this is what happens:

COPY	:SIDES.LEFT	ACTION
1-SQUARE	4	draw and turn
2-SQUARE	3	draw and turn
3-SQUARE	2	draw and turn
4-SQUARE	1	draw and turn
5-SQUARE	0	stop

The theoretician DIJKSTRA, inventor of the language ALGOL and one of the pioneers in the development of programming, said that once you understood how variables are used in programming, you understand the essence of programming. We believe he was refering to local variables as used in recursion when he said that. Understanding local variables has become more important in the world of systems and commercial programming as well with the use of "re-entrant" code in operating systems. Many such systems have extensive subroutine libraries. When these subroutines use local variables and observe other coding restrictions, they are re-entrant --which means they can be used simultaneously by several programs.

VARIABLES AND ABSTRACTION

The Logo turtle can't deal with abstractions. It must go forward some specific amount or turn through some specific number of degrees. When you key "FD :some-distance", the LOGO interpreter evaluates the symbolic name "some-distance" (looks in the box or storage cell to determine its contents and substitutes that contents for the expression ":some-distance").

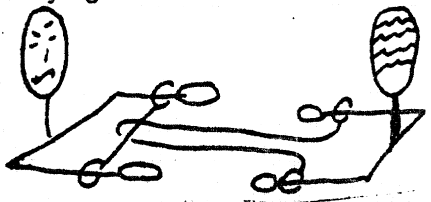
People apparently can deal with abstractions, but find problem solving easier when they don't have to do so. Most often when a new procedure is being written, people use specific operand values, e.g. FD 100, which they later change to variable form, such as FD :some-distance. The nature of the abstraction involved is common to some other examples of mathematics as well. The famous mathematician Bourbaki describes the creation of an axiomatic system as proceding from the mathematician's working out a series of theorems with very concrete examples in mind and subsequently examining the inferences of his theorems to define precisely which characteristics of his examples were used by the theorems. In a third step, he redefines the set of objects to which his axioms apply as that most general class of objects having all those characteristics used in the theorems. That is, he bases his generalization on the operations he performed and not on a list of the characteristics of the example he began with. We stress that the process through which a child generalizes a procedure after creating a concrete product with a concrete precursor, this child's play, is a particular kind of abstraction of value in the most intellectual endeavors as well.

This mathematical form of abstraction is called reflexive abstraction by Piaget, where he sees the child creating his own mind through processes of thought that are like those of Bourbaki's mathematician. This points to the most significant potential impact of computer experience on children developing their minds. Reflexive abstraction may become more "natural" to them than what Piaget calls "aristotelian abstraction" (abstraction by feature selection and classification) with which Piaget contrasts it. That is, more children of the future may more often think like mathematicians than do children of today.

GETTING OFF THE GARDEN PATH

Some problems are terribly difficult because they tempt you to set up your description in an unproductive way -- and lead you that way down a dead end path to useless fretting. Here's a good example of such a problem, one that you might run into at a party:

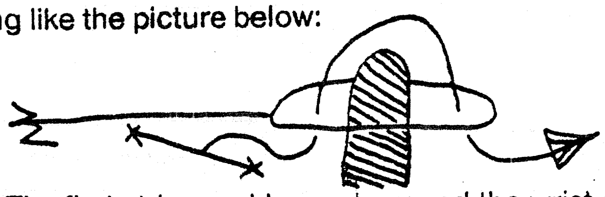
> You need people who are willing to work at the problem as couples. You need string and a little ability to tie knots. Here's what you do. Take one string and tie it loosely around the wrist of one "victim". (Leave about two feet of string between the wrists.) The circle of string, arms and body form the first loop. Pass the second string through the first victim's loop and tie each end loosely around the wrist of the partner. Passing the second string through the first victim's loop made the loops interlocking. The puzzle is how these two victims can separate without cutting the string or untying the knots.



Your victims might get angry if you don't help them solve the problem. Maybe you should try it yourself before imposing on anyone else.

Most everybody sees the string, arms and body as forming a loop. This is what puts them on the dead end of the garden path. After they have been told it's illegal to slip the string loop off the end of their arms from around the wrists, they frequently try all sorts of contortions to get free, then give up. Have you given up yet ? Do you see how to solve the problem ?

A critical question to ask here is "what can I really count on ?". Note that if the arms, body and string really do form a loop the problem can not be solved. That whole way of looking at the problem must be wrong. Next notice that the places where there might be a break in the loop can't be between the body and arms; it has to be at the wrists. There are four wrists, but if you can get the string past one of them the problem is solved. Focus on one wrist and try to think of a different way of seeing the problem. I think of it as being like the picture below:



That "pole" is supposed to be a wrist. The first string and loop go around the wrist at one end and then off somewhere else. The problem is now to get the second string out from under the first. It's easy, isn't it: through the loop, over the pole and down on the outside. This sort of problem can only be solved after you get off the garden path.

SUMMARY

1. when you have a hard problem, it can be very important to ask yourself, "what can I really count on in the way the way I am describing the problem ?"
2. a second good question, when you are looking for a new way to describe a problem is "What's the point where there is something unusual or still unclear ?" Focus your attention on that point.

RE-SOLVING PROBLEMS

Some problems you want to put behind you -- like having to do what you don't want to do, and not being able to do what you do want. Such problems should be resolved. Other kinds of problems have a friendlier face, and certain of them are worth solving and re-solving. Think about making a circle. Doing so is a classic Logo problem for beginners. Novice learners are typically asked to "do-it-yourself", to walk through the problem by simulating the turtle. Their typical explanation of what they are doing as they walk in a circle is that they go forward a little and turn a little and do it again. This explanation translate directly into the Logo circle:

```
TO CIRCLE
FORWARD 1
RIGHT  1
CIRCLE
END
```

The Logo circle is very easy to make with a Logo computer, but it would be difficult to make such a circle by drawing on a piece of paper. The Logo circle is very perimeter-focussed because the turtle knows nothing at all about "centers". (This leads to some interesting bugs and problems in turtle geometry procedures.) The Logo circle is natural in the sense that it is no more than the path of an activity as familiar as walking is.

In plane geometry if you ask, "What's a circle?" the object, "the locus of all points in a plane equidistant from another point", is easy to construct with a compass and not even hard to construct without one. The euclidean circle is as "natural" as the Logo circle in the following sense: imagine a person sitting; the figure traced by the farthest reach of his arms is as circular as the path followed by any person imitating the Logo turtle. The euclidean circle is center-focussed, and the circle is the boundary of the center's territory. Can you get a computer to draw a euclidean circle? There are several ways. If your computer speaks "polar", you can specify the definition of a circle with the simplest of equations, radius = constant. Descriptions of circles in polar coordinates are simple, but they get complicated quickly if located away from the coordinate system origin.

While the description of a circle in polar coordinates still keeps in mind the relation of the circle to its center, and to a process a person could use unaided to make a circle, the description of a circle in a system of cartesian coordinates becomes remote from the process of generating a circle:

$$X^2 + Y^2 = C^2$$

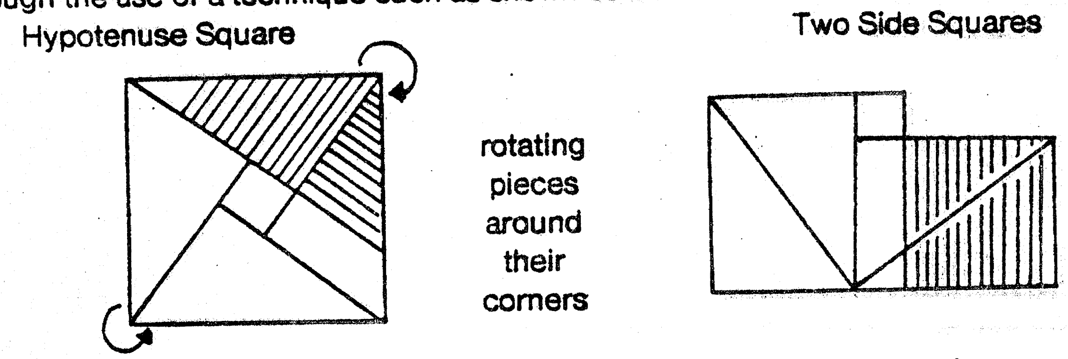
This algebraic equation for an origin centered circle (of radius 'C') specifies that the circle is the set of all point pairs (X,Y) in a cartesian coordinate system which satisfy the equation. The primary relationship between the circle and "something else" is here between the circle and the cartesian reference frame. This contrasts with the Logo circle (where the primary relation was between the circle and its process of creation) and the euclidean circle (where the primary relation was between the circle and its center). The cartesian description of the circle and other curved lines, although central to the development of modern mathematics and science, seems relatively un-natural as compared to the Logo and euclidean circles because of the extent to which the person is removed from the description of the circle. (over)

SUMMARY

Scientists have recommended re-solving problems through the ages. Descartes recommends that whenever you encounter a new idea, you bring it into comparison with all the other ideas you hold as valuable and try to appreciate their interrelations. Feynman, a famous physicist of our time, relates that his practice as a student was typically one of solving a problem whatever way he could, then, with a worked out solution to guide him, to re-solve that same problem in as many different other formalisms or frames of reference as he could.

SOMETIMES YOU NEED ANOTHER IDEA

One of the most famous problems in the history of ideas puzzled the mathematicians of ancient Greece. They knew how to count very well (even though they used letters of their alphabet to represent numbers). They even knew about fractions, and this is where the puzzle came up. They knew about numbers like 1, and 1/2, and 1/4, but they wondered if there were any numbers that couldn't be represented by whole numbers or fractions made from whole numbers. The puzzle became a hot issue for them after the discovery of the pythagorean theorem. They could prove that the areas of two squares constructed on the edges of a right triangle was equal to the area of a square constructed on the longer line (the hypotenuse), through the use of a technique such as shown below:



This proof helped make the problem more critical because it raised a specific question. If you start with a square, one unit long on the side, and make a triangle by drawing the diagonal of the square, the sum of the areas of the two squares constructed on the side will be two units of area; but how long must be the hypotenuse, H, of the triangle made from half a unit square ? "H" must be greater than one and less than two. It must be more than five-fourths and less than three-halves. Greek mathematicians suspected no fraction of whole numbers would result in the number two when multiplied by itself, and they began the attempt to prove there was no fraction of whole numbers equal to H. They tried to represent the number H a fraction of two wholes numbers, T (the TOP number) and B (the BOTTOM number). They knew that H times H had to equal 2 and developed these equations:

$$\text{first,}\quad H \times H = 2$$

$$\text{then,}\quad \frac{T}{B} \times \frac{T}{B} = 2$$

$$\text{or}\quad \frac{T^2}{B^2} = 2$$

$$\text{finally,}\quad T^2 = 2\,B^2$$

Having reduced their relation of the possible whole numbers T and B to this simplest form, they were <u>stuck</u>. What else is there to do ? Where can you go from here with this one idea ? Think about it for a while. Can you go on from here ? (over)

Another idea is needed, another whole different way of looking at what "T" and "B" might be. The trick is to look "inside" T^2 and B^2. What must they be made of? No square can be a prime number (squares are made by multiplying at least two other numbers together). The factors of a square must be two (in number) if the roots are prime or some multiple of two if the roots are not prime, as in the example below:

SQUARE	25	36	64	100
PRIME FACTORS	5x5	(2x3)x(2x3)	(2x2x2)x(2x2x2)	(5x2)x(5x2)
COUNT OF FACTORS	2	4	6	4

Any number is either a prime number or can be decomposed into prime factors. Therefore every square <u>must</u> have an even number factors. But think back about the equation: $T^2 = 2B^2$. Doesn't that imply there is at least one square, T^2, which must have an <u>odd</u> number of factors? It surely does, and therefore it must be wrong. Consequently, there must exist numbers, like the square root of two, which can not be expressed as the ratio of two whole numbers. That is, irrational numbers exist.

This mathematical proof was a difficult one for men to discover. Then someone realized that a new idea was needed, a new way of looking at the problem. Once a second way of describing the problem was brought to bear, its solution was relatively straight-forward, almost obvious. When you have a real hard problem, maybe you ought to think about whether some other description of the problem could help you with it. Finding the right description isn't always easy; it may, however, be necessary.