3

Some Notes on Friendly Computer Systems

Andy diSessa

- January 6, 1978 -

Logo as a computer language is nearly obsolete. When I think of the things we know now about how to make a better, friendlier computer system and about the cheap computer power coming to implement those things, I am greatly embarassed by what has been done. It seems no one has had the time or money to follow up the many leads that using current Logo has suggested. If we get more money and manpower, perhaps it will happen automatically, but I am not sure of that.

In a larger context, very few people (maybe none except Xerox) have been engaged in developing a broadly integrated friendly system of the sort that can win with child, housewife, student ... me! Not many even think this is an important area, and of those who do, who compares to MIT in technical expertise and sensitivity in the area?

To vent my spleen on this state of affairs and to put down on paper a few ideas I've had, the following annotated list of Logo peeves and suggestions is presented. It is very local and does not even begin to talk about the large scale setting of a general frame for research into friendly systems.

1. Image of the Computer

Logo suffers from lack of a large scale coherent computational image of what is going on in the machine. Small Talk and actor systems in

general are at least an attempt at creating such an image. I don't think it is necessary or even desirable that we "take over" that kind of solution suggestion. This is particularly true since it gives up what I think is one of Logo's real potential strengths -- the integration of text manipulation and general computation power. Logo's list and language oriented basis could be a real plus in trying to achieve what is for me, a high priority for the next round of systems, a coherent text and computation system. (I don't see how one can ignore the fact that for many, computers will and should mean text manipulation.) The integration of graphics, text manipulation and computation power per se deserves a lot of study. (For some specifics on capitalizing on Logo's language roots with a more comprehensive computational image see BACK-TALK below.)

2. Editing - Text and Programs

The day of separate text and program editing should become rapidly history. Real time editing of text and programs should be the rule with at least a minimal set of graphics oriented editing commands available at all times. (Certainly the teletype artifact of only capital letters should fade away). This theme is followed up in the description of AUTO below and also, to some extent, in BACK-TALK.

3. Define time parsing and other "on line" helps.

Some of the most frequent and painful bugs in Logo are caused by parsing problems. Need I mention negative inputs subtracting each other, parsing in predicate situations, and that bane of highschool programmers, parsing of arithmetic expressions. Some of these can simply be "fixed" with a more reasonable parser, but many remain necessary or at least useful

ambiguities which, for all their usefulness, will always invite bugs, frustrating debugging effort, and a clutter of parentheses. Why shouldn't parsing be an interactive component at define time, or even whenever an expression is typed in. Certainly it wouldn't hurt to give graphics indication of how the expression is being (will be?) passed. (Automatic parentheses to open an expression such as an input, with user option to close it if its not obvious where to close it, etc., is probably unusable and cluttered, but in the right direction. Other more graphic and less clumbsy-to-use methods need be explored.)

The role of ":" in Logo is as much a help for the user in visually parsing and understanding program structure as to denote "the value of a variable." Viewed in this way it makes sense that all procedures which output should have ":" or some other signifier associated with them. In a parallel way would it help to have procedures which need inputs marked as, for example, POLY:? Functions would look like :SIN: As most (all?) of the burdon of supplying such signifiers could be left to Logo one might expect a genuine increase in readability without requiring more of the user. (Hal's suggestion following this same observation -- do away with : and " and use, e.g., Pascal format for inputs.) In contrast, my gut feeling is that data type signifiers should be entirely optional.

Would define time parsing speed Logo up? What other "partial compiling" can we offer without sacrificing clearity and flexibility? (After all, what <u>is</u> Logo doing most of the time but waiting for the next line to be typed?)

<u>4. Files</u>

Logo's filing system, beyond its moderately successful tree structure is one of the more blatent, unfriendly, hard to learn to manipulate, unstructured, garbage can method for keeping carefuly thoughtout and well structured programs that I can imagine. (I can't even remember what's on top in my own files - let alone use someone elses.) Having to manipulate with workspace size chunks is the first loss. (I know you can use obscure primitives to do what you like, but who has the time to invent and implement a reasonable system which will only work marginally anyway because it has to wind its way around Logo's implemented fixtures.) Wouldn't it be nice if procedures were stored heirarchically with top level on top so you could see from the organization of a file where to start and who uses whom as subprocedures. Ideally the user should have control of how and where even top level procedures appear in the file. (More on this sort of thing in BACK-TALK section.) To be sure there is some conflict between versatility and ease of use here, but a flexible but richly defaulted system doesn't seem unmanageable.

5. Redifining primatives

One of the most useful user options which Logo flatly denies is the redefining of primatives. Let me give some examples. Suppose a user wants to have his turtle move in real steps with pauses along the way. (Several visitors have asked me if such an option were available.) Similarly, mightn't it be a good idea, at least for beginners, to change RIGHT so that turning is visable and RIGHT 40 and RIGHT 400 are visually distinct? If a student wishes to study scaling properties of designs or programs, shouldn't he have the option of controling the size of the turtle unit.

(The current TURTLESIZE is an add on Kludge to do this one thing, without any parallel ones like scaling turns to use radians.)

An important issue for me is the perceived coherence in the pieces of a Logo experience. Let me use an example near and dear to me. If one wanted to create Logo an extention of turtle geometry to physics, a dynaturtle, the meanings of many primitive funcitons, CS, WC, HOME, SETTURTLE etc. should be different in such a setting, but all in a very natural way. Currently one must replace all those old commands with newly named special commands, giving an unfortunate complication to remembering what to do in the different but similar environments (turtle and dynaturtle). Creating a 3-d turtle geometry or a turtle geometry in any special world (like a CUBE) asks as strongly for system primitives to be user redefinable.

Isn't it a very clean and powerful method to vary programs by replacing one chunck (say FORWARD) by a user defined substitute (say SQUIGGLE)? Isn't the idea of elaborating and developing one time "black boxes" an important computational metaphor deserving recognition in the language?

6. Instantizing

One of the most attractive areas of computer-person interaction is real time control of ongoing process. We should have buttons and joysticks but, beyond that, the keyboard is an obvious facility which, unfortunately, is buried in CTYI, CHAR and TTYP. Suppose a kid had the option of "instantizing" any key -- that means running a procedure whose name is the key marker instantly on pushing the key. Even beginners could write their

own etch-a-sketch type real time drawing systems with a few trivial commands.

```
TO F            TO R            TO L            TO S
10 FD 10        10 RIGHT 30     10 LEFT 30      10 SQUARE

TO E            W               U               D
10 ERASERDOWN   10 WIPECLEAN    10 PENUP        10 PENDOWN
```

How about using the keyboard as a "piano" musical input device?

If instantizing became an interrupt even with another procedure running, many dynamical control areas would open up and I think greatly increase the domain of phenomenon and range of things to be done by elementary students.

There are modularity problems in such a suggestion having to do with utilization of scarce resources (single letter commands) for different purposes in different contexts. Furthermore the idea stands now as an ad hoc (but not, I think, unattractive) "feature." But if some such thing could be integrated in a more natural way into the system of user control over control (other such issues: parallel processing, execution of INIT files etc.) the benefits could be well worthwhile.

For more specifics on this see INSTANT below.

7. Self-Explanation

Logo now doesn't even know it's own set of commands!

There is a distinction to be made here. A system can be self-explanatory in two fundamentally different modes: active (and usually explicit) or passive (and usually implicit). An active mode pretty much requires question and answer format with dedicated structures (like a HELP key and subsequent annotation including maps, explanations etc.). On the

pie in the sky level one can imagine intelligent monitors watching for errors - offering explanations, asking for intent in order to be able to give implementation help. The passive mode avoids dedicated structures but is a carefully organized system whose behavior can be assertained by the user through interrogations on the level of operations which the system is meant to perform. For example, an interpretive system is one of the most important advances in passive self-explanation in that it allows a programmer to ask questions easily just by trying a command sequence out. Passive self-explanation can be very much enhanced by a coherent conceptual model of the machine which allows the user an invariant knowledge base concerning how to pose question. An inbuilt self-explanatory capability with respect to what is actually going on in the machine may well supplant more complicated active debugging aids such as .DEBUG and TRACE. (Again, see BACK-TALK.)

Both active and passive explanatory capability should find their place, but let's not be so literal minded that we forget the importance and advantages of the passive approach in terms of its person to person invariance, freedom from supplementary constructs (understanding explanatory maps and jargonized explanations) etc. It is a hard problem to know at what level and in what terms an explicit explanation should be. The job is easier if we leave a bit of the work to the user in terms of formulating questions on a operational level (providing we can give him the opportunity to ask and an ease of interpretation of the response.) A key question: Does the user know enough to pose a useful question when the question is likely to come up?

8. Nits to Pick:

a) Arrays should clearly be cleaned up. I don't think any array creation or type specification should be necessary (optional, of course, for tight packing). Assignment statements should be the same as variables, as should be retrieval.

MAKE "A_{ij} 5 (i,j, some numbers)

PRINT :A_{ij}

As a physicist I like subscripts but I'm open to suggestions. If subscripts are used one should allow index expressions such as

$+_i\ V_i * W_i$

for summation etc. On the other hand one should not need to specify subscripts to identify the thing as an array. One should be able to deal with it in a chunk.

OUTPUT :A

b) Infix operators - we should have them user definable. Consider vector operations. If you need a dot product you should be able to make one. We should have the option of redefining + so that it can handle vectors, etc. I don't think these issues are trivial "because they only concern advanced students." If done well without a lot of muck I don't see why elementary school students can't handle vectors.

c) Improving List Processing. The current way of handling lists with FIRST, LAST etc. to unpackage them and prefix WORD, SENTENCE etc. to package seems a bit abstract and overtly counter-visual. Wouldn't gluing together words with a hyphen or sentences with a semi-colon be more direct. Wouldn't even active brackets (so that the insides gets evaluated) be

visually preferable:

[:FOO :BAR :REST]

to

(SENTENCE :FOO :BAR :REST)

Isn't

(BUTFIRST :LIST) ; (FIRST :LIST)

more attractive than

SENTENCE BUTFIRST:LIST FIRST:LIST

My reasons for preferring these suggested forms are 1) visual clarity, 2) it establishes a hierarchy with markers and operators ; - [ ] on one level and entities like :WORD and :LIST on another.

I know this is a problematic area where tastes enter strongly, but it deserves thought.

d) Repetition is at least as useful and powerful notion as recursion. I personally feel that Logo should recognize this with a language feature like a collection of REPEAT, REPEAT UNTIL, REPEAT WHILE commands.

Below are some more specific suggestions. The first two are things I've tried out and still think are useful ideas.

INSTANT

Suggestion 6 above is not hard to implement in pieces with the current system. Here's what I did for a try.

The INSTANT system consists of knowing two commands and a few

stereotypical format uses. The command GETLETTER sets a variable "LETTER" to a character just grabbed from the TTY buffer. If nothing was typed, it sets "LETTER to " . (You don't want the procedure to sit and wait for a letter since there are undoubtedly other things to be done.)

The procedure DO is a slightly smarter RUN whose stereotypical use is to cause a single letter procedure to be run.

```
GETLETTER
DO :LETTER
```

Currently, GETLETTER collects and stores up a number to be used as an optimal prefix REPEAT which is enacted with DO once a non-numerical character has been typed. For example

```
INSTANT
10 GETLETTER
20 DO :LETTER
30 GO 10
```

implements an instant mode for the whole keyboard. IF 5F is typed, F is run 5 times. If 537G is typed G is run 537 times. This repeat in instant mode I consider a fairly useful frill. DO is error trapped not to stop when the letter happens not to have been defined as a procedure and for a few other frequent glitches.

GETLETTER and DO :LETTER can of course be used in various other formats. One may want to TYPE:LETTER or set up a dispatch table.

```
TO GETCOMMAND
10 GETLETTER
20 IF :LETTER ="S STOP
30 IF :LETTER ="L MAKE "NUMBER :NUMBER - 1
40 IF :LETTER ="M MAKE "NUMBER :NUMBER + 1
```

A procedure like GETCOMMAND will usually sit in some dynamic Loop.

```
TO WALK :NUMBER
10 FORWARD 1
```

```
20 RIGHT :NUMBER
30 GETCOMMAND GO 10
```

One may wish to combine a dispatch with a DO, adding a DO :LETTER for line 50 in GET COMMAND.

I have played with this little two command system specifically in the content of making an easily approachable but user dominable mini-dynaturtle. It makes interactive game writing very easy as well as performing its basic funtion in this context, allowing real time dynamical control. (Notes: I suspect a more transparent GETLETTER which outputs its find like TYPEIN of REQUEST is really the way to do this. Furthermore, the number prefix might be made available to be used as an input rather than (in addition to?) as a repeat.

AUTO

The following is implemented "new front end" for LOGO. It attempts to be friendly and coherent in a number of ways, particularly looking forward to unified text-procedure systems. It has a philosophy:

1) What is typed on a terminal should be an active artifact of what you have done. You should be able to go back in history, see what's there and do it again, edit it, or collect the successful parts of it into a procedure at will and with ease.

2) In conjuntion with 1) it aims at making your interaction with programming a more fluent affair by avoiding sequencing and avoiding collecting or separating acts of experimentation, playing, defining or editing. It does not like modes such as edit or define mode as they tend to enforce a non-modularity of decision making. This particularly is an

outgrowth of my observations of the difficulties beginners have with define mode and its change of meaning for the do-it key etc, and was reinforced by Bob Lawler's ideas on non-planning. In addition while writing this section I ran across an old "protocol" by Laurie Miller which I've attached to this paper. It certainly reinforced for me the observational basis of the design decisions made in AUTO.

Here's how it works:

The functions involved with the LOGO CR are separated out and inparticular made mode independent. Now CR means new line, DO-IT means do it and REMEMBER means store this line away always. A new "place" (Small Talk window) is reserved for the procedure as defined to date. [Now, AUTO uses the part of the TV screen to the left of the TURTLE field which is normally left unused.] TO still gives a title to a procedure but can be used at any time; it does not need to start a defining sequence. END terminates a definition in so far as it clears the define window.

I think it important to visually separate off what a student has created so far in a procedure from what he types or tries in the act of defining.

Editing is a all-the-time-available feature. The ↑, ↓, ←, → keys move the cursor up a line, down a line, left or right a character. "Rubout forward" and "rubout back" work, and the cursor always stands unambiguously between letters. SUPER prefix causes →, ← and rubout keys to refer to words. Your history can be retrieved and reused by climbing up the scroll with ↑. SUPER ↓ goes to the bottom of the scroll. A DO-IT will run the line the cursor is on no matter where in the line it's placed.

Line numbers are assigned automatically and sequentially when a REMEMBER key is pushed. Of course you have the option of specifying a line number for inserting intermediate steps etc. A line number on a blank line

when REMEMBERed erases that line from the procedure window. Even when you are specifiying line numbers, a DO-IT still executes that line (ignoring the number). Incidentally you can change your title at any time by remembering a new (or an edited old) TO line.

Repetition is easy in AUTO - a bunch of DO-IT keystrokes. One can just as well insert a line a bunch of times into a procedure with REMEMBER strokes.

A mistake in a line typed for direct execution can be easily edited rather than retyped. Studying a procedure's behavior dependence on an input involves just changing the input on the line and doesn't need any other retyping or other than obvious editing commands. One can easily type out a "menu" and select from it with ↑, ↓, DO-IT.

BACK-TALK

(Name corrupted from Radia Perlman's Double talk)

Logo lacks a coherent, comprehensive, computational image. Here is a preliminary proposal to help that. It aims at making the computational process comprehensible. It wants the process, including flow of control, visable and user controlable.

BACK-TALK is text and spacially oriented. The fundamental chunk is called a box which may be a variable, a procedure, an entire environment or a paper text. The surface structure organization within a box is a two dimensional array of words. A third dimension is reachable since any word can be the label on another box which may be "entered" by moving the cursor to the word the using ENTER command. (This much of the arrangement is

really nearly identical to the Architecture Machine Group's Spacial Data Management System - the existance of which has encouraged me to think more seriously about BACKTALK). Thus the entire hierarchical structure of a procedure can be maintained, with subprocedures actually living inside (underneath) a procedure. Transporting a procedure as for filing could transport all necessary subprocedures. A box, since it looks like a text on the surface, can serve as a repository for text. Or it can be a list... or an array. In terms of connectivity structure I don't think one needs any more for a filing system or set of environments. Wandering through files, examining variable, looking at a program all would involve the same small set of curser control commands. (The local movement set would need to be augmented with global commands to FIND, PICKUP and PUTDOWN boxes in various ways.) I think the unified coherence of such a system is worth some effort to make it work, particularly since such mysterious operations as assigning a variable become simply interpreted as writing <u>something someplace</u>. Filing can become picking something up and putting it somewhere, that "where" importantly being a place you can walk around in and rearrange things there.

Running a program (text) is carried out by some entity, let me call him Chief Honcho (CH) whose chore it is to successively "evaluate" the words in a text. Evaluating a primitive amounts to doing it and having the name of the primative disappear. If the primitive outputs, the output merely replaces the primitive in the text. Evaluating a box which is a program means the CH enters the box (replaces the name with the box named) and evaluates what's there. The command which serves to output is BECOME

(or BE) which causes a whole box to become a new word or another box. A variable is just a named box which says, for example, BE 5.

A box can input words from the left or right of it in the text by grabbing them right out of the text. Note again this is intended to be visual so that programmer can watch his program run. In parallel to grabbing from the left and right, one should be able to spit words out to the right or left.

Consider the following programs and the pictures of their operation.

(EAT and POLY on next page)

Note use of box to parse, collecting the input to BE

One might make inputting cleaner and more automatic in various ways. For example, INPUTS, S A (from the right is the default) in place of the first two lines. Notice the tail recursion is effected with a BE.

(Programs following next page)

A trully recursive program would build up a visual stack of environments which the user could inspect at his pleasure.

I have not specified the rules of this substitutional grammar, who

EAT

```
IF INPUTRIGHT = FOOD
    BE YUM!
    ELSE BE YUCHT!
```

① EAT FOOD

② FOOD

```
IF INPUTRIGHT=FOOD
    BE YUM!
    ELSE BE YUCHT!
```

③

```
IF FOOD=FOOD
    BE YUM!
    ELSE BE YUCHT!
```

④

```
IF TRUE
    BE YUM!
    ELSE BE YUCHT!
```

⑤ YUM!

---

① EAT BALONEY . . . . ⑤ YUCHT!

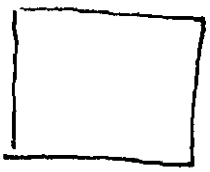
POLY

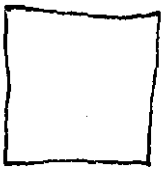
```
S← INPUTRIGHT
A← INPUTRIGHT
FD S
RT A
BE [POLY SA]
```

SEARCH

```
INPUT, WORD  <~ IF EMPTY WORD
                BE <NOTHING>
IF WORD = BAR
    BE BARN
    ELSE OUTPUT LEFT WORD BE SEARCH
```

① SEARCH FOO BAR

②  FOO BAR

③  BAR

...

④ FOO SEARCH BAR

⑤ FOO  BAR ... FOO BARN

FACT

```
INPUT N
IF N=1
    BE 1
    ELSE BE N * FACT N-1
```

FACT 5 ... 5 * FACT 4 ... 5 * 4 * FACT 3

... ... 5 * 4 * 3 * 2 * 1 ... 20

gets evaluated when, in what order. The simplicity of BACKTALK depends on finding an "obvious" and useful default set of rules and a sufficiently small set of user commands to modify the rules. (Presumably one would need a literal mark - "don't evaluate now" - and an "evaluate now" mark at the least. These resemble : and " except : is more general now meaning more like RUN. Default rules might look like "evaluate everything immediately if possible and when done with a line return to beginning and evaluate again until no more evaluations are possible.") The set of rules and modifying commands can be somewhat baroque provided the default is obvious enough that beginners need only a vague sense of the rules. Since the action of the default is quite visable, correcting for a mistake should be straight forward. (Watch the thing run... "Oh, the CH wants to evaluate that now and I want it to wait; I need a literal mark.")

In order to make a box useful as an environment with local procedures and variables, one needs some additional structure. I think some form of a "local library" should live in a corner of each box, inspectable and writable if the user cares to. The local library contains the list of locally known procedures, variables etc. with their definitions if they are not given explicitly in the text of the box. This could allow procedures to keep local data even while they are not running.

Rules for the use of the Local Library (L.L.) could be something like this:

1. If a word is to be evaluated and its contents are not written explicitly under the word, the CH checks with the L.L. for the evaluation.

2. Commands changing the contents of any box (such as setting a

variable) must do that somewhere. The local library is the storage place for those changing boxes. Normally the user would be sitting in some box (environment) whose L.L. contains all "global" (for him) variables.

3. The local library must specify a default recourse in case a given box cannot be found there - no default means no evaluation. The standard default should be LOOKUP, meaning look up to the Local Library of the box containing this one (dynamic binding).

The local library is an obvious place for annotations of various kinds to explain the box with which it is associated. For example the default Logo box in which a first time user finds himself might have list of primatives and some explanations in its local library.

The role of CH is a bit unclear at this stage except as a place marker, the place where computation is taking place. Presumably his insides would be the place to make changes in grammatical rules and other changes in control structure, such as instantizing. My original plans had the CH being the repository for all global information. For example he would contain the knowledge about all primitives and global variables. Locality of variables etc. would be established by having reserved space in each box (like the local library) to change the CH's contents upon entering or leaving the box. I now prefer the Local Library version, saving the insides of CH for user control over control

An episode in Learning Logo

I had a little trouble logging in. First I pressed the "Do It" button correctly. Then I typed in BOB.LAWLER. The third instruction was to read in in "LAURIE. So I typed in Laurie and the computer answer that it didn't know how to Laurie. I don't remember exactly how I solved that problem but I think I eventually asked somebody for help. I found the entire opening procedure very confusing. Now I think I have memorised it but I don't understand very well why I have to go through so many ste ps to call a program out of memory. I also find the mnemonics a little confusing. To get the names of the programs in my workspace, I have to type POTS. I imagine PO means Print Out but I don't understand TS. I looked it up in the manual but the explanation in the manual is incomprehensible unless you already know what things like titles and names and files are, and how they are different.

The program I had written last time is the following.

```
CORNER :DEGREES
10 FORWARD 100
20 RIGHT :DEGREES
30 CORNER :DEGREES
```

Bob's first suggestion was that I try out different values for DEGREES and see what happens. I did that for a while but it was not very challenging. I figured out pretty quickly that CORNER produces a closed figure for any value for DEGREES. Furthermore, DEGREES/360 =x/y where y is the number times Corner runs before the figure closes and x is the number of complete 360° rotations the turtle makes before the figure closes. I didn't find that first episode very interesting.

Bob's second suggestion was that I try to edit CORNER to accept an input for length. I knew to do that I would have to change the title and line 10. To do so, I remembered from my first lesson that I would have to be in EDIT mode (although I don't know what a mode is). So I typed EDIT. I received an error message that I hadn't given enough input. I guessed that it might want the name of the program so I typed EDIT CORNER :DEGREES. It accepted that. I had to guess what to do next. I looked up the section on edits in the manual. I understood (not clearly) from the manual that I could edit a line by typing EDL and something after. So I typed EDL, returned the carriage, and typed the old version of line 10. I got an error message. Then I think I typed EDL 10 FORWARD :LENGTH which is what I wanted the new line to be. I think I got another error message. Finally, I tried EDL 10, carriage return, 10 FD :STEPS. (I changed the name of my variable from LENGTH to STEPS so that it would represent more what I wanted. LENGTH would be right if an input could be 100 cm. but steps is better if it is only 100). It worked. Later Bob explained that the EDL 10 line was superfluous.

Editting the title was more difficult. I understood from the manual that I had to type EDT and then something. So I tried EDT and then the name I wanted to assign. I got an error message. I tried inserting a carriage return in various place but that did no good. Finally I asked Bob for help and showed me what to do. After Bob had gone, I practiced editting the title and I succeeding twice. I noticed during the editting procedure that when ever I typed in EDIT CORNER :STEPS :DEGREEES or PO CORNER :STEPS :DEGREES, I received an error message saying that I had given no value for :STEPS although the computer did do what I had wanted. Bob explained that I didn't have to type :STEPS :DEGREES after the word CORNER.