STUDENT COMPUTING IN MATHEMATICS: INTERFACE DESIGN

FRANK QUINN

Abstract. This is the first in a series of articles on a computing environment designed to support learning in mathematics and other technical areas. They draw on many years experience with students working with computers and in computer environments, discovering unexpected learning problems and trying to fix them.

The main point is that human learning is quite complex and as we move away from the tightly–bundled package of hand calculation in traditional classrooms the full complexity is coming into play. There are more ways for learning to fail than most of us imagined; many are different from the things educators traditionally look for and are hard to recognize; and underlying causes are obscure.

This article concerns basic student–computer interactions. Among many other things we see that standard cut-and-paste can undercut some learning objectives and has to be modified. The sequel (Student Computing in Mathematics: Functionality) concerns computational functionality. Careful limitations are needed to avoid turning the subject into keystroke sequences.

These articles are very speculative and intended as starting points for further investigation, not a fixed prescription for a final product.

Contents

1. Introduction

I taught my first computer math course in 1975 and was convinced that it was the wave of the future. However it was atypical—a very small class at Yale—and

Date: January 2009.

later attempts were unsatisfactory: either too labor–intensive, or weak outcomes, or (usually) both. I have helped develop computer–*based* and computer–*tested* courses, but, ironically, the students still use by–hand computation.

Others have had similar experiences. Even after much tinkering, few college courses use more than calculators.

Calculators are widely used in K–12 math but many college teachers now associate calculator training with deficiencies in symbolic skills, number sense, and geometric understanding. Reasons calculators might undermine higher learning emerge from the analysis in this essay. Current calculators may be nearly the worst possible learning environment. In any case this cannot be considered successful.

Why has student computation been so problematic? The first problem seems to be a lack of understanding of how people learn. The second is a lack of new learning goals that computation should make accessible.

1.1. **Learning and the Interface.** We understand *teaching* very well, but teaching evolved to produce learning in a traditional classroom and, in math, using traditional techniques. It has turned out that many features of traditional good teaching are artifacts of these environments, *not* features of learning. In particular simple substitution of technology for hand work in traditional lesson plans has turned out to be a poor strategy.

§2 (Guiding Principles) gives a list of odd features of human learning and the contortions needed to fit mathematics into it. The math material is relatively general (support symbolic and abstract thinking, etc.) intended to guide design of a learning–friendly interface. More content–specific material is discussed in the sequel, Student Computing in Mathematics: Functionality.

The basis for this analysis is experience with computer–tested and computer–based courses. These are a useful context for the study of learning because students are the primary actors in formulating and implementing their learning strategies. They can be guided but not as rigidly channelled as in traditional classes. I have spent a lot of time watching students learn in this context and their approaches are very different from what I tried to make them do in classrooms for thirty years.

§3 (Interface Design) applies these principles to find input methods, formats, and interactivity designed to maximally support learning. Such an interface should, it seems, be quite different from ones now in common use. Among the unexpected conclusions is that ordinary copy-and-paste is counterproductive: it does not leave a record that can be diagnosed for errors; and as a purely kinetic activity it undercuts the use of symbols to represent other expressions. Learning–friendly alternatives are proposed in §3.1.3 and §3.1.4.

2. Guiding Principles

This section gives some painfully–acquired insights about technology–enhanced mathematical learning. The focus here is on human learning and generalities about mathematics needed for interface design in the next section. Principles used to guide functionality of the computational environment are discussed in the sequel.

The point of view is as important as specific insights: there is much more to be learned and as we gain experience we must be alert to new insights that further shape design.

2.1. **Learning, not Technology.** Our objective is to help human beings learn. Humans are the bottleneck: technology design should be completely driven by the needs of human learning, *not* by availability, limitations, or capability of technology. Examples:

- Calculators are cheap and effective; shouldn't we exploit their availability? No. Calculator design is constrained by low cost and small size, and they are intended for calculation, not learning. They can be effective in meeting modest short–range goals (e.g. in K–12) but are counterproductive in the long run.
- Full–featured computer algebra systems have amazing capabilities; shouldn't we use them to "empower" students? No. They are designed for high–end professional use, not learning. Students can easily get lost in full–featured interfaces, and learning to mechanically use powerful black–box functions usually fails to develop understanding that transcends the particular interface or enables flexible general use.

To get good learning we must first understand learning, then we must design technology specifically to support learning. Anything off-the-shelf, or easily adapted from something off-the-shelf, is almost certainly unsatisfactory.

2.2. **Symbolization.** Elementary mathematics divides roughly into conceptual and mechanical (computational) steps. The goal of providing a computational environment is to enable focus on the conceptual activity. We discuss the division, then how it should be organized.

2.2.1. *Conceptual Activity.* Conceptual activities include organizing information and representing it ways suitable for computation. We refer to this as "symbolization" for several reasons:

- Material must be organized as symbolic expressions to be mechanically manipulated. Numbers are considered symbols here since their special properties only come into play in computational steps.
- Representing things as symbols is part of abstraction and a vital part of mathematical thinking.

We expand on this last point. The human mind is quite limited in the number of things that can be tracked or manipulated at one time. Fortunately the individual things can be very complicated. Thinking about complicated material therefore proceeds in two stages. The first stage is to identify good intermediate abstractions to serve as conceptual units; pack as much structure as possible into these units; and then develop enough automatic familiarity so we can actually see and use them as units. The second stage is to think about interactions between a relatively small number of these abstract units.

Representing things as symbols is only a small instance of the construction of conceptual units but it plays a vital role in early learning of the methodology. In particular if it is designed correctly it can serve as a model for later, more elaborate instances.

2.2.2. *Computation.* What counts as mechanical computation—to be farmed out to a machine—depends strongly on level. Why, and what to do about it, is the subject of the sequel, Student Computing in Mathematics: Functionality. However

for the most part computation involves manipulation or evaluation of symbolic expressions. In other words, computation acts on the output of symbolization.

2.2.3. *Organization.* As noted above our objective in providing computational support is to enable focus on symbolization. This can best be done by separating the two activities as much as possible.

Traditional problem–solving usually alternates the activities: the general practice is to compute as soon as a computable chunk of the symbolization is complete. These interruptions fragment symbolization, distract from organization, and conceal mathematical structure. A computational environment should be exploited to consolidate activities: symbolize first, then compute all at once, at least for elementary work. See the essay Beneficial High–Stakes Math Tests: An Example for an illustration of this in a specific simple example. In particular it is not a good strategy to simply use machines for the computational steps in traditional approaches.

Calculator work largely bypasses symbolization and mixes organization and computation so thoroughly that higher–order learning is inhibited. See the reference just above for an illustration. Another can be found in §3.1 of Task–oriented Math Education.

2.3. **Modes of Learning.** The first general point is that human learning is strange and complicated, and while there are commonalities there are also great differences between individuals. A design goal is to support the commonalities while leaving flexibility for individual preferences.

The second point is that over time the complexities of learning become transparent. Character recognition and parsing of mathematical expressions become automatic. The use of a keyboard to obtain characters on a screen becomes second nature. This means the way experienced people do or learn things is not a guide to how neophytes learn, and effective tools for experienced users can be barriers for beginners.

We give examples in four cases: visual/kinetic reenforcement, visual/verbal reenforcement, the role of imitation, and subliminal learning. In each case we see that technically inconvenient things may have to be done to connect with human learning.

2.3.1. *Kinetic Reenforcement.* There are interactions between visual input (reading, seeing) and kinetic input (writing, drawing, copying) that seem to be important for memory and internalization of certain types of structure.

Examples:

- Mathematical notation is precise and delicate: omitting or wrongly locating a symbol, or misreading a statement, can completely change or destroy meaning. Copying a problem by hand seems to improve comprehension and reduce errors, and many traditional students are taught to do this as a matter of good practice. Copying often does not make sense in an electronic environment. We must be alert to problems caused by lack of kinetic feedback, and may have to find a substitute.
- Very young children learn to generate and manipulate symbols kinetically (by drawing them). This gets linked to visual recognition and alternate entry modes (keystrokes) to eventually form a seamless unit not dependent on drawing. Drawing does not translate easily to electronic environments,

but trying to reduce or eliminate it from early learning will probably cause many children a lot of difficulty.

- Kinetic feedback seems to be vital for some students in internalization of geometric structure. Specifically students taught to graph functions by hand usually internalize the qualitative features of graphs of quadratic functions, exponentials, simple rational functions etc. This internalization is used strongly in later work involving plane and three-dimensional shapes, multiple integrals, and the like. Students taught with graphing calculators and tested by identifying a graph among several alternatives have been trained completely visually. They have never, or rarely, picked up a pencil and drawn the curve. And many of them cannot: apparently they have not internalized qualitative features from the purely visual approach. Omitting kinetic reenforcement puts these students at a disadvantage.

There are interesting examples in other subjects. Coloring books are a common adjunct to anatomy texts. Apparently kinetic feedback from coloring in muscles, bones etc. is helpful in fixing these complicated structures in memory.

Kinetic reenforcement might be incorporated in technology by, for instance, requiring students to trace over a computer–generated graph before allowing them to use it or submit it for a grade.

We caution that kinetic involvement alone is not a goal. Calculator arithmetic is intensely kinetic but counterproductive because it replaces rather than reenforces work with structure and symbols.

2.3.2. *Verbal Reenforcement.* Some people have strong verbal orientations. People who move their lips when reading, for example, are translating from visual input to kinetic (movement of lips and tongue). Their best comprehension is through the kinetic channel, and comprehension falls if they have to rely on visual input alone (trying to read while eating, or trying to read something—like mathematics—that they can't pronounce).

Errors that involve confusing or substituting symbols that sound the same indicate verbal involvement. Confusing m and n, or M and m is probably a verbally–based error, while confusing p and q is a (dyslectic) visual error.

Verbal is *not* the same as auditory: there is a profound difference between speaking and hearing. Some students can transcribe lectures but can't read their notes out loud. Children in rural areas can listen to standard English on television for many hours each day but only be able to speak an incomprehensible local dialect. Hearing and speaking tend to be more tightly linked in later life but this is an example of the transparency that can conceal basic features of learning.

A corollary of this point is that audio or audio–visual materials are not a substitute for student verbalization. I do not have any clever ideas on how to incorporate verbalization into technology. For the present this has to remain a job for teachers.

Implications in math education:

- Children should probably be taught to read (out loud) what they have written. They should be encouraged to listen to what they say, and make sure it is what they meant to say. In other words, run stuff through the verbal channel to check it for accuracy.
- Students should be taught *how* to read material out loud. For instance reading expressions involving parentheses can be tricky and this is probably

related to the frequency of parenthesis–related errors. This difficulty is *not* a justification for avoiding parentheses since this leads to serious problems later on.

- Reading is closely related to parsing because reading requires linear organization. I have seen verbally–oriented students completely stumped by notations such as $\Sigma_{i=1}^{n}\frac{i}{2^i}$ that make use of positional information. They usually find it tractable when taught how to parse it so they can read it out loud.

2.3.3. *Imitation.* People learn a lot by watching other people do things *provided* they can see how it is done. A teacher working a problem at a blackboard provides a model that can be imitated. Exactly the same information presented using prepared overhead slides, projected computer output, or PowerPoint cannot be imitated and therefore deprives students of this primitive and innate learning mode.

There is a particular role for imitation in mathematics. Since we do things one-at-a-time the construction of a mathematical expression is a linear dynamic process.

- Mathematics itself is usually not linear so this process involves a linearization, ideally one that follows mathematical structure.
- Parsing an expression, for instance to read it out loud (see the previous section), also involves a linearization.

These two linearizations are often different. If students are not given examples to imitate then they almost always try to use the parsing linearization, and this is less efficient and more prone to errors than structural linearizations.

The following examples are fairly complex so opportunities for confusion will be clear. Students will have similar problems with much simpler expressions:

- The structural linearization used to expand $(a+b+c)10e^{x-5}$ has first step $a(\dots)+b(\dots)+c(\dots)$. We then enter the complicated expression inside each pair of parentheses. The verbal linearization requires switching back and forth between a, b, c and the complicated expression, and offers many more opportunities for error.
- The structural linearization used to construct $\Sigma_{i=1}^{n}\frac{i}{2^i}$ begins with $\Sigma(\dots)$. Filling in the parentheses is usually the next step and the limits come last. In the verbal linearization the summation variable and limits come first, not last. This invites errors like $\Sigma_{i=1}^{n}\frac{n}{2^n}$.

It is easy to see implications for machine–based examples and presentations: they should be dynamic and emphasize the thinking behind each step. It is less clear how this should influence design of a computational environment.

2.3.4. *Subliminal Learning.* Students can potentially learn from anything, and it is often unclear exactly what they are learning. A consequence is that everything should be designed so that if students do learn from it then they will learn correctly.

For instance by–hand arithmetic involves a lot of symbol manipulation and shows mathematical structure in action. It may be that students internalize it and this prepares them for algebra. Calculator arithmetic avoids symbols and hides structure and so does not provide the same opportunities for subliminal learning.

As another example we describe how very young children can be offered an opportunity to absorb a sophisticated mathematical viewpoint. Mathematicians think of "addition" in functional terms: any binary operation that is associative,

commutative, has a neutral element and inverses is entitled to be called "addition" and represented by "+". One point of the abstraction is that work habits appropriate for integers and other elementary examples are equally valid for any other "additive" system.

Now imagine showing a child that pretty much any two expressions that can be entered into the computational environment can be combined using +. In general this is just a property of these things, but you can see what it is good for in special cases: if you combine numbers representing lengths of two sticks using + then you get the total length of the two sticks joined together. In other words, finding lengths of joined sticks is an *application* of + when it is applied to numbers. It is *not* the definition, and + is *not* limited to numbers that can be interpreted as lengths. This is not a point that should be made explicit to children, but the approach presents it in a way that it can be absorbed subliminally.

2.4. **Process, not Answers.** The particular virtue of mathematics is that correct use of correct methods gives correct answers. The focus in learning mathematics should therefore be on methods and their use. The implication for the current context is that a learning environment for mathematics should "show work" in the sense of providing a record of the methods and reasoning used, and this record must be usable for locating and targeted correction of reasoning errors. A wrong answer only indicates that an error was made, not the nature of the error, and without a diagnosable transcript the only recourse is to repeat the work and hope for a better result the next time.

"Correction" here means fixing errors of reasoning or mathematically incorrect methods, not conformity with a standard template. An alternative but mathematically correct approach does not need correction.

2.4.1. *Learning from Process.* In any other subject there is enough imprecision in terms or context, or possibility of unanticipated factors, that careful logical reasoning can fail to give a correct conclusion. It is still worthwhile because it greatly improves the *chances* of getting a correct conclusion, and I believe that experience with careful reasoning in complex logical systems is the greatest general benefit of studying mathematics. In this sense the medium is the message.

2.4.2. *The Role of Answers.* Mathematics also has the virtue that incorrect reasoning usually gives an identifiably incorrect conclusion. This means correct answers can be a useful proxy for correct reasoning. However this is only true if students are using correct methodology. Independent checks on methodology should be possible, and the reasoning itself should be available for diagnosis and correction when the answer is wrong.

2.4.3. *Diagnosis and Error Correction.* Ideally *every* error should be diagnosed and corrected. This would catch misunderstandings immediately, before they can be reenforced by repetition. It would also encourage students to work carefully to avoid triggering the diagnosis process.

Our best goal is for students to learn to detect, diagnose, and correct their own mistakes. Teacher diagnosis and correction should offer models students can imitate. Self–diagnosis can also be promoted by providing diagnostic aids for worked–out problems, see Task–oriented Math Education, and the Teaching Notes on the AMS Technical Careers web site. This is an aspect of problem design rather than

the learning environment, but it can only be effective if there is a record that students could try to diagnose.

3. Interface Design

This section is concerned with structuring the interactions between student and machine. Objectives established in the previous section include focus on organization and construction of mathematical expressions (§2.2 Symbolization); supporting learning modes such as kinetic reenforcement (§2.3.1); and producing a diagnosable record (§2.4).

Much of the student–interface interaction looks like elementary programming. This is implicit (or subliminal) rather than explicit, and is enforced by the structure of the interface. This is not an accident: programming requires extensive symbolization and explicit use of structure and so is a good model for mathematical learning. In fact it is completely compatible with the primary learning objectives to take development of fluent use of high–level programming languages as an additional long–term objective.

The section is divided into Input Modes, concerned with immediate interactions between student and machine; Windows and Sessions, describing structure of interactions, and Input Formats.

3.1. **Input Modes.**

The primary input mode should be writing or drawing directly on the screen with a stylus. Reasons include:

- For young children this avoids indirect input and provides kinetic reenforcement for number and symbol formation;
- for all students it provides kinetic reenforcement of graphic work (§2.3.1); and
- it enables easy and intuitive addition of commentary and reference tags.

3.1.1. *Character Recognition.*

The interface has to provide character recognition, but it should probably should not learn the user's style. Reasons include:

- It is appropriate to expect reasonably clear character formation, and for mathematical material it may be better for the interface to be a bit picky about characters than to have to override inappropriate guesses. "Guess" could be provided as a button.
- An adaptive system leads to non–portable input habits: they won't work on other machines, and in particular not on secure systems used for testing. Some individual calibration will be necessary, for instance for left– and right–handed differences, but this should be kept to a minimum to avoid becoming a stumbling block.

It is not so important that text entry be portable, and careful math mode would be available as a backup, so these considerations do not apply to text.

Anyone concerned about asking students to adapt to an interface should reflect on how well they have adapted to a really terrible text interface: entering text on a numeric keypad with their thumbs!

3.1.2. *Keyboard.*

A standard keyboard should be provided for fast entry of text. However there should be *no function keys*:

- Functions may be disabled.

- Functions should be considered parts of mathematical expressions, internal to the symbolization process and recorded in the transcript, not as external objects living on a keypad. Logarithms should be obtained by writing "`Log`" and evaluating, not by pushing a button.
- In order to separate conceptual and computational steps we want students to construct expressions, possibly including functions, and then evaluate them. Immediate evaluation (via a function key) defeats this.

For similar reasons the interface should generally *not* depend on palettes for insertion of symbols, patterns, or functions. It might provide lists or browsers from which material can be copied in appropriate ways.

3.1.3. *Copy-and-Paste via Symbols.* Standard copy-and-paste has the same invisibility and symbolization–defeating problems as function keys and so should be strictly limited. We suggest substitutes for some of the functionality.

The first replacement for copy-and-paste is symbolic assignment. The student can select an expression in a static window and assign it a name, for instance by writing "`A=`" in the selection area. The expression can be used in an input window by entering the name, "`A`". When the name is referenced the selection and name assignment should be frozen to preserve a record.

Example: A company has 47 employees with an average salary of $37,867. What is the total payroll of the company?

The student can select 47 and write "`emp=`" in front of it, then select 37,867 and write "`sal=`" in front of it. He then can enter "`emp*sal`" in the input window and evaluate to get the answer. Alternately he could enter "`payroll = emp*sal`" to have the output accessible for later use. Note this scheme subliminally supports symbolic thinking, see §2.3.4, and provides a record of the work.

3.1.4. *Copy-and-Paste via Tracing.* The second replacement for copy-and-paste is provided by tracing "templates". The student selects something in a static window and drags it to a work window. A dimmed copy appears. This cannot be evaluated or further selected, but the student can trace over it to get a functional copy.

- For young students this provides kinetic reenforcement of character and symbol formation, and construction of expressions.
- This provides kinetic reenforcement of graphic material, see §2.3.1.
- Expressions or graphic material can be modified rather than traced exactly. Symbols could be changed to change the input into the expression, for instance.

Again the selection area should be frozen and linked to the template to provide a record.

3.1.5. *Copy-and-Paste in an Input cell.* Input cells in a Work window (§3.2.3) should be an exception to these restrictions. These cells serve as "scratchpads" and there currently seems to be no reason to disable full copy-and-paste *within* such a cell.

3.2. **Windows and Sessions.** There should be three standard window types: Data, Work, and Preview.

3.2.1. *Data Windows.* These are static in the sense that they cannot be edited, but annotations and selections can be made in an overlay.

Data windows can have form boxes in which material can be entered, for example answers when the data window displays a test. Form boxes should be assigned names so material can be entered either directly (by stylus or keyboard) or by assignment. For instance in the payroll example in §3.1.3 the answer box might be assigned the name "`answer5`" and the answer could be recorded by entering "`answer5 =  emp*sal`" in the Work window.

Note this design makes symbol use natural and helpful so it encourages symbolization.

3.2.2. *Preview Window.* The preview window nicely formats expressions but does not manipulate them. Expressions to be previewed are selected (in the usual way) and a Preview button is activated.

- Error messages are issued, for instance when parentheses are unbalanced.
- The formatting displays how expressions will be interpreted. For instance the expression `2^5 x` will be previewed as 2^5x. If 2^{5x} was intended then the mistake will be evident and appropriate parentheses added.
- Complex expressions should routinely be proofread using Preview. For instance the TeX expression `\Sigma^n_{i = 1} \frac{i}{5^i}` is previewed as $\Sigma_{i=1}^{n} \frac{i}{5^i}$. If this is not what is intended then the input expression can be edited.
- Some incomplete expressions should be previewed as expressions with boxes for missing material rather than giving an error message. For instance the incomplete Mathematica expression `Integrate[  , {y,  }]` should be previewed as $\int_{\square}^{\square} \square\, dy$.

Preview material cannot be edited directly, nor can it be copied. If the source window is static then the original selection can be assigned a name or can be used to form a template. If the source is in an active input window then it can be edited.

3.2.3. *Work Windows.* Work windows are divided into alternating Input and Output cells.

The bottommost Input cell is active, and can be edited, previewed, and evaluated. Results of evaluation appear in the Output cell immediately below. The Output cell cannot be edited. The active Input and Output cells are dynamic so cannot be annotated and the material in them cannot be selected for copying.

Input and Output cells other than the bottommost are static (have been frozen) and cannot be edited. They can be annotated in the overlay, parts selected and copied, etc.

The active Input cell can be repeatedly edited, previewed and evaluated. It becomes inactive (is frozen) when a new Input cell is opened at the bottom of the window or an End Session button is activated.

Input and Output cells can be deleted. Links and symbolic–copy material from a deleted cell disappear with it. If the bottommost remaining cell is an Input cell then it becomes active.

3.2.4. *Sessions.* A session consists of working in an Input cell, freezing it by opening a new Input cell, and repeating as needed until the session is closed.

When a session is closed the Data and Working windows are linked and saved together as a Data window. This preserves the work record because it can no longer be edited. This record can be diagnosed for errors and annotations can be added to record the diagnosis. It can also be used as a source to rework problems using a new Work window. Correct fragments from the previous session can be spliced in to reduce repetition and focus on corrections. All this provides support for the diagnosis and targeted learning described in §2.4.

Closing the session may activate additional features. For instance if the Data window contains a test then closing the session should activate scoring functions to grade the test, and a section containing answers and diagnostic hints should become accessible.

3.3. **Input Formats.** We have discussed how symbols and other material should be entered in the interface. The topic here is how these objects should be arranged to be accepted for processing. We will mostly be concerned with symbolic material. There seems to be a role for an input format for graphics but it is far from clear what should be involved. There should also be formats for data entry, for instance tables of numbers, but these will be special–purpose and infrequently used. Entering a lot of numbers is not a useful or real–life activity and data will usually be accessible in electronic format.

The design is driven by concern for learning and based on watching and working with students on computer projects in calculus and vector geometry. A particular conclusion is that *writing* and *reading* need to be separated. A symbolic input format must be easy to write and edit; it need not be easy to read. The beautiful two–dimensional formats of typeset mathematics are easy to read but not easy to write (for machine use) and not easy to edit. There is not going to be a satisfactory single format. Instead we optimize formats for specific uses and use Preview and other tools to negotiate between them.

3.3.1. *Linear, Primitive, Explicit, Verbose.* These are characteristics needed to make the format easy for inexperienced users to write and edit. Sophisticated or experienced users with other needs should use a different system.

"Linear" means in particular that the format should not incorporate positional structure ($a \wedge 5$, not a^5). Positional data entry invites mistakes and frustrating misinterpretations. Positional representations are harder to edit. Finally, copying part of a positional representation can fragment formatting instructions and lead to bizarre results and obscure crashes.

"Primitive" essentially means limited to text. A good rule of thumb is that it should be possible to send an input expression by email as text. We might accept Unicode rather than ASCII so some symbols could be considered text. However most mathematical functions should be spelled out in some way.

"Explicit" means everything in the expression must be visible. Invisible material, for instance formatting instructions, is dangerous and not worth the trouble. "Unambiguous" should be part of this. For instance multiplication is better denoted by $*$ than $\times$ or a dot since the latter two are easily misinterpreted.

"Verbose" means that instructions and function names should be spelled out in ways that are easy to guess and remember. For instance to get an integral one should write out "`Integral`", and then provide arguments. Abbreviated function

names can be entered faster by experienced users but add a coding/decoding layer that is difficult and distracting for students.

To repeat, the objective is a format that students can easily learn to write and edit. The features listed above seem to help with this but do not guarantee success. In particular, students learn most easily and naturally from examples and hints, not explicit instruction. If students have trouble learning basic use of the format from examples then the format needs improvement.

3.3.2. *Mathematically Correct.* It may seem odd that this needs explicit mention but traditional notations and ways of thinking are sometimes imprecise, rely on context, or are heuristic rather than really correct. In such cases correctness requires a break with tradition.

For instance "=" is traditionally used in several different ways. The expression

$$y = ax^2 + bx + c$$

may indicate an *assignment*: the symbol y is given the value of the expression on the right side. Or it may indicate a *test*: a relation that is either true or false, as in "Intersection points of the two curves are points (x, y) that satisfy ...". Further, an assignment can be *immediate*, in which case y is given the current value and not effected by later changes in $a, b, \ldots$; or *delayed*, in which case y is a shorthand for the right–side expression and it's value at any particular time reflects current values of $a, b, \ldots$. It could even be an *implicit* assignment intended to specify a or x, etc.

This notational ambiguity means a traditional expression containing "=" is incomplete and must be accompanied by text indicating the meaning. Confusion results when, as is too often the case, the text is omitted[1]. This is bad enough in common practice and unacceptable in a machine environment.

A mathematically correct format must have different notations for these different meanings, or support them in other ways (e.g. implicit assignment might be done with a "solve" function rather than a variation on "="). This will conflict with ambiguity in traditional notation and language, but we should see this as a feature (who needs self–inflicted notational confusion anyway?) rather than a flaw.

3.3.3. *Graphics Input.* An important objective in studying functions is to develop a feel for qualitative features of their graphs. What does re^{st} look like as a function of t, independent of the values of r, s? How about $r+(t-s)^n$? Beautiful computer–generated graphs in specific cases are not a substitute for qualitative understanding.

It would be nice to have a full syntax and computational context for qualitative graphic information, but for starters it would be useful just to have an input format. Suppose we ask a student to draw a graph with the qualitative features of an exponential function. How can we extract (mechanically) the qualitative features of graphic input to determine if the drawing is reasonably correct?

Note that we really do want students to draw the graph by hand. Kinetic reenforcement seems to be essential for some students (§2.3.1), and is probably important for all, so this is another case where technical difficulty or convenience cannot be allowed to override educational concerns.

[1]The confusion can even include a failure to recognize this as a notation problem. W. Byers in *How mathematicians think: using ambiguity, contradition, and paradox to create mathematics*, (Princeton U. Press 2007) argues that this reflects a basic ambiguity in mathematics itself!

4. Summary

The long–term goal is to better prepare K–14 students for advanced learning in mathematics, science, engineering, and other technical disciplines. It seems obvious that this should include systematic use of machine computation, but most attempts have been counterproductive and none have been fully satisfactory.

The core problem seems to be that current computational environments do not support the complex oddities of human learning. This essay describes a rough draft for an interface design driven by this complexity and the structure of mathematics. The final form will no doubt be different from this draft but it should also be clear that it will be profoundly different from any current interface. It is also clear that development of such an interface is a delicate and sophisticated undertaking.

The sequel, Student Computing in Mathematics: Functionality concerns the need to carefully limit functionality of the computational environment.