Recognizing Symbolic Errors in Student Responses

Eric Bell
Triton College

Web pages and computer programs are becoming more interactive with every year. An intranet-based tutoring system currently in use at Triton College requires that students enter a numeric expression as part of their on-line assignment. The purpose of the system is to identify errors made by the student and to provide assistance to those who may be having difficulty with details in the problem solution.

The challenge with numeric expressions is that while a student may enter an expression that evaluates to a correct answer, what is to be done if the student's response is not correct? A trivial response is to merely tell the student that their entry is incorrect and ask them to try again. A better solution is to identify the type of error made by the student. For instance, it is not uncommon for a student to misapply a trigonometric function. The goal for the system is to recognize that the source of the error was the trig function, and not some other cause, and then provide either explanation or remediation.

The previous version used a system that was very labor intensive and required a great deal of creativity on the part of the tutorial designer. The current system builds on previous efforts and is able to identify more creative, and multiple, errors on the part of the student with substantially less work on the part by the designer and is a significant extension of previous efforts. This paper will describe the system currently in use that is being tested on students.

Background

Computer aided instruction has come a long way since the early days of drill and practice, however, there is still much to do. Current research in computer–based tutoring systems have yielded us few programs suitable for field use, but have provided us a wealth of didactic functions. Among these are problem solving strategies and more tactical discourse elements. Both of these are intended to help students bridge their understanding for the material – an engineering course in this case – to new knowledge. The fundamental tenet of the process lies in Vygotsky’s Zone of Proximal Development, which is that a student’s understanding is incomplete, but he/she is capable of greater achievement with some assistance by a “more knowledgeable other” in those topics of incomplete knowledge.

By definition, a student’s knowledge is incomplete during the learning process. The only determination of correct behavior comes in the form of a test, which may be the solution for a problem. The delay may not be desirable, though, because by the time a problem solution has been presented, many steps, and opportunities for errors, have passed, and any one of the steps may be at fault. Thus, it behooves the system designer to work with students at as low a level as possible, tracking and guiding the student through every step.

In conventional tutoring/tutorial software, the student can be asked for a solution to a particular step. The design issue has always been that if the student is wrong, the system is not capable of determining exactly what went wrong in the same way that a human tutor might be able to. This system element provides exactly that type of guidance to the student, providing hints and suggestion rather than simple right/wrong responses. In a real sense, the system is attempting to establish a dialog with the student, with the computer system acting as the “more knowledgeable other”.

The functional goal is a set of computer programs and routines that provide a pedagogically significant dialog within a problem solving setting for identifying errors in mathematical expression, and then helping the student to correct those errors.

Situation

In a previous version of this program/system, the problem designer was responsible for entering a set of solutions that were known to be likely candidates for a student's incorrect solution. In this way, the student who entered in an incorrect solution whose numeric solution matched the designer's was guaranteed to get a useful response. Unfortunately, this was very time consuming and required an unusual amount of creativity on the part of the problem designer. Entry was laborious, at best, and often overlooked simple student errors. The single greatest challenge was compensating for multiple errors. The process quickly became one where every conceivable incorrect solution was entered, which I believe is not taking proper advantage of the designer.

Correct Solution	Some Incorrect Solutions
2*COS(25+30)	2 * COS(25-30)
	COS(25+30)
	2 * (COS(25) + COS(30))
	2 * COS(25) + COS(30)
	SIN(25+30)
	SIN(25) + SIN(30)
	TAN(25+30)

Note: All trigonometry functions assume degrees.

Figure 1

The solution considered here was to get the computer to process the correct expression and apply the most likely kinds of errors. Those errors include using the wrong arithmetic operator (eg. adding instead of multiplying), using the incorrect trig function, and just plain forgetting to include a term or variable. The challenge for the program was that more than one error could be present. This implied a good recursive algorithm.

A program was developed that accepts a valid numeric expression (a set of utilities were developed in the previous version that validated student entered expressions) and re-parses it adding parentheses about sub-expressions with higher precedence.

D / C + A * B
Becomes ((D / C) + (A * B))

This allows the program to create new tokens that can be 'turned on' and 'turned off' during a recursive decent based on their grouping by parentheses. Thus, even in this simple example, the sub-expression (D/C) can be treated as a single token.

(D/C)	(D/C)	D	D
+	+	/	/
(A*B)	A	C	C
	*	+	+
	B	(A*B)	A
			*
			B

Figure 2.

This provides a copious set of combinations to explore. Each token set is passed to a processor which recursively applies error-generating modifications to token. The numeric value is then determined and compared to the student's entered expression.

If the value matches and the other limitations are acceptable, then the 'solution' with the fewest errors is presented to the student. An explanation of each component is presented so that the student may benefit and make a correction. Because student errors can be so creative, not all errors will be caught.

Evaluation of Student Input - Part 1

Students have a knack for using expressions that you or I would find clumsy or awkward. But they can often be correct, so the system attempts to compensate for this kind of variability by only comparing the numeric result. There are several good tools for this that allow use of variables too.

But the more important question is how to evaluate the student response when the numeric result is wrong. This is where the system described above can come to the rescue. First, the student expression is over-parenthesized - meaning that parenthesis pairs are placed about every binary operator and its pair of operands. The program then goes though, removing redundant parentheses, as in ((A)) becoming (A). The next step is to recursively turn on and off each set of parentheses that are nested. When a parenthesis pair is 'turned on', the sub-expression it surrounds acts as a single token, otherwise, each element is evaluated separately.

The next process for the parsed expression is to enter another recursive routine that replaces each token with a zero or one, depending on adjacent binary operators to give the effect of being "forgotten". If the adjacent binary operators are addition or subtraction, the token is replaced with a zero. If the adjacent binary operator is multiplication or division, then the token is replaced with a one. This has the effect of removing the sub-expression represented by the token from the expression, but not necessarily invalidating the expression. (Consider that multiplying by zero would eliminate both sub-expressions in the expression, or the harm attempting to divide by zero could cause.)

This is performed to the student's entry, as well as the correct entry provided by the problem designer. Both expressions are evaluated numerically. When there is a match, the results are stored in a temporary data file. When all of the combinations have been processed, the system then attempts to find the most likely candidate for the error.

In the first round of error checking, the system is only trying to find the terms that must be in error - not necessarily explain the error(s) to the student. Thus, the system scans the temporary file for the shortest error. This is the most likely candidate for the error. The reasoning behind this is that if a student has entered an incorrect solution, then the entire expression is obviously incorrect, but knowing that is trivially useless. The goal it to find the smallest set of errors that produce the same result as the student's mathematical expression.

An example may serve us here. Take a look at the set of expressions in Figure 1. The top expression is provided for reference and is available to the system, but not to the student..

Correct Expression 2 * COS(25 + 30)
Student Expression 2 * COS(25) + COS(30)
Error in these terms… 2 * COS(25) + COS(30)

Figure 3

The student enters the second expression (2), and is presented with the third (3) after the first phase of processing.

If the student enters an expression that is incorrect, and is found to be completely incorrect, then he is prompted to re-examine the problem or refer to the remediation material. By the way, there is no penalty for using the remediation material.

A special note needs to be added at this point. In the case of the example shown above, the system is not looking for a "2" and then a "COS( )" function, et.al. The student could have entered

COS(55) * 6 / 3

and it would have been evaluated as correct, because it numerically evaluates to the correct answer. The system is looking to find components, which *when eliminated*, match the designer's solution with one or more of *its components eliminated*.

An interesting note is that this process takes much less time than I would have feared because newer machines are so much faster (800MHz - 1.8GHz) than machines that were available when this project was started. Thus, the problem of CPU usage has not been a significant factor.

Evaluation of Student Input - Part 2

A second routine examines the problem of identifying the probable error the student actually made. The primary limitation here is that the set of solutions for the student error is merely a good guess, but a reasonable start.

The system proceeds in much the same way that the first process did, but replaces operators and functions until the numeric result matches the student's expression. This is definitely a case of hitting a fly with a sledgehammer in terms of processing time.

An example set of data for the second part of the program looks like that of Figure 4 (again, the top expression is for reference):

Correct Expression 2 * COS(25 + 30)
Student Expression 2 * SIN(30+25)
Most likely error: replaced COS with SIN

Figure 4

Another example where there are multiple errors is shown in Figure 5:

Correct Expression 2 * COS(25 + 30)
Student Expression 2 * SIN(25)
Error in these terms… replaced COS with SIN forgot 30

Figure 5

This kind of error explanation needs to be developed, but can provide students with the clues they need to provide a correct solution. By the time the student is getting this message, it is too late to fix this particular problem[1], but a follow-up problem usually is of a similar type, so the lesson need not be lost. Additionally, a presentation of remediation material is available if the student feels they really don't understand their error.

When examining the expression during the second pass, an answer may be forthcoming, but owing to the nature and creativity of students to mangle a numeric expression, it is possible that no solution may be identified. In this case, the student is simply told that their expression did not provide enough clues to find the error. They are prompted to try again, or examine the remediation material.

An arbitrary limit of three errors was imposed. The feeling here was that if a student generates more than three errors in an expression, then he has made a serious conceptual error and needs more help than a simple diagnostic program can provide. A side benefit of limiting the number of errors is that this also limits the depth of the recursive routine making processing faster.

An Implementation

With the goal being a better learning experience, I designed the initial application of this system to simply get student responses to relatively simple problem components. As you can see from the example outlined below, the purpose in this first system is to get the system up and running, and provide a useful learning tool for the student.

[1] A learning application that implements Mastery Learning would likely allow the student to continue until successfully solved, providing remediation as needed.

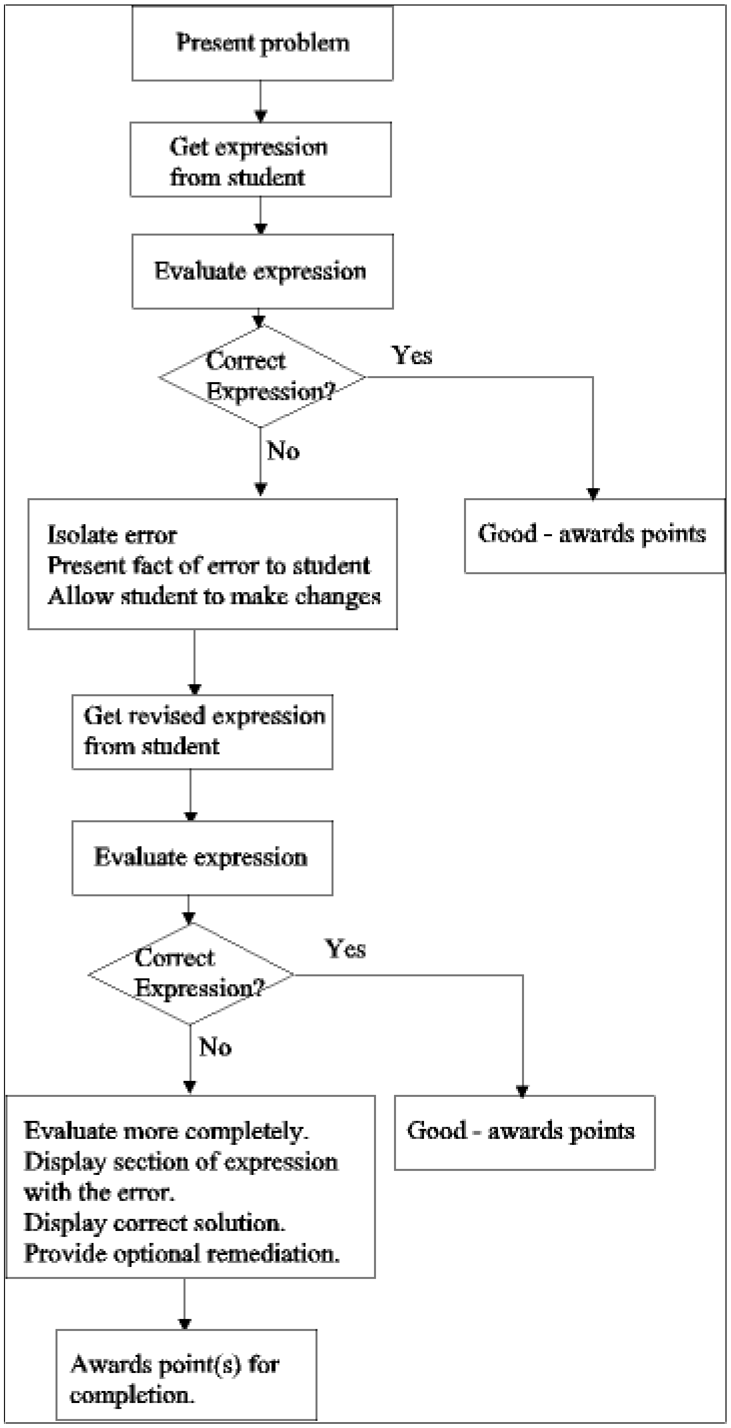
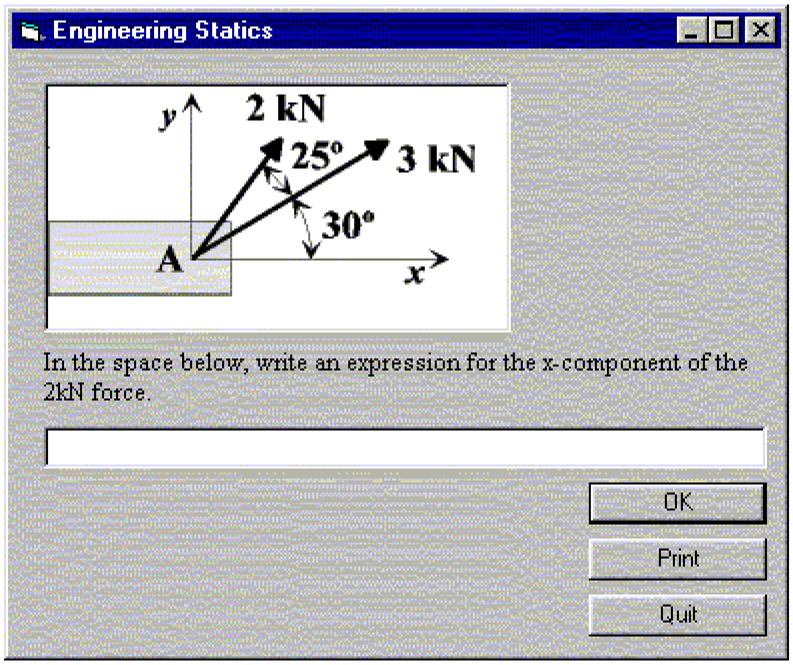

Present problem
Get expression from student
Evaluate expression
Correct Expression?
Yes
No
Isolate error
Present fact of error to student
Allow student to make changes
Good - awards points
Get revised expression from student
Evaluate expression
Correct Expression?
Yes
No
Evaluate more completely.
Display section of expression with the error.
Display correct solution.
Provide optional remediation.
Good - awards points
Awards point(s) for completion.

Figure 6

The program interface is very simple, as can seen below, in Figure 7.



In the space below, write an expression for the x-component of the 2kN force.

Figure 7

When the student has entered the expression for the question, he then presses the OK button, which causes the system to evaluate the student's work. The first thing is to simply evaluate the expression numerically. If the solution is correct, then the student is given full credit and allowed to continue to the next problem. The results are printed so that students can turn in their work for credit.

Interesting Alternatives

The analysis programs do not have to be fed a strict diet of correct solutions in order to perform a useful task for the student. One alternative is to apply the expression modification process to the student's incorrect solution and determine what needs to be done to get it to match the correct solution. This approach will be investigated to see if response resources can be reduced, as web-based processing is usually more expensive and time-consuming.

Another alternative is to allow the problem designer to feed the system known incorrect solutions. Keeping in mind that student solutions can be very ingenious, provide a set of carefully chosen incorrect solutions for further processing and matching. Similarly, allowing multiple correct expressions would provide for indirect student solutions. Remember that what seems obvious and optimal to you is anything but that to the budding engineer.

Evaluation

To date, there has not been a formal evaluation of this program. Informal discussions with students indicate that they generally like the system, and are especially fond of being allowed to correct their mistake before finally submitting it for a grade. Most identified that they were encouraged to enter their solution as abstractly as possible. Those who tried to get a solution on their calculators first were usually disappointed until they "relaxed" and let the machine do what it wanted to do. The reason for this was that an incorrect solution entered numerically resulted in being told they were just wrong with no helpful explanation. A few expressed a desire to play with the system to see what else it could do.

When asked if they would rather not have the additional work, they (generally) expressed the desire to keep the system because they get homework credit for it, and the feedback is fairly specific, allowing them to fix their incorrect notions immediately rather than wait until the homework gets returned. One student even preferred it because the system never got tired of him making mistakes like the tutor did. A few of the students expressed a desire for non-credit problems so that they could become more familiar with the system before using the system for a grade.

Discussion

The system as described works and is definitely useful to students. The purpose of this system has been purely educational. The focus has been to isolate the source of the student error. As explained above, students don't seem to mind the explanation of their error(s), but seem to like them, taking them as suggestions to re-examine their entered expression rather than a strict correction. This is a significant improvement over mathematical software that is able to solve for a particular variable, but unable to compare two different equations and identify the meaningful differences. Outside of an educational setting, “significance” is difficult to define, and even then, software has to be built-up around the response to take advantage of the educational opportunity.

Conclusion

The application as presented here is barely useful, but the expression-parsing engine is extremely useful and has great promise. Its proper place is in a system of much greater complexity and interaction. The focus of this project is an application of a purely educational system, serving the needs of students rather than instructors. In this way, the system is more student-centered than most and deserves further development.

References

Bell, Eric. (2001), "*Using an Expert System to Recognize and Remediate Student Errors*", Proceedings of the ASEE Annual Conference, Washington, D.C.

Vygotsky, L. S. (1978), "Mind in Society: The Development of Higher Psychological Processes", Harvard University Press, Cambridge, MA.

ERIC BELL
Eric Bell is currently a full time instructor at Triton College. He received his B.S. and M.S. in Mechanical Engineering from the University of Illinois of Chicago.