Improving Feedback in Elementary Mathematics

Autograders

by

Stephanie Denise Carter Greene

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY September 2014 [February 2015]

© Massachusetts Institute of Technology 2014. All rights reserved.

Signature redacted

Author .................

Department of Electrical Engineering and Computer Science September 25, 2014

Signature redacted

Certified by ....

Patrick H. Winston
Ford Professor of Artificial Intelligence and Computer Science Thesis Supervisor

Signature redacted

Accepted by ....

Albert R. Meyer
Chairman, Department Committee on Graduate Theses
Improving Feedback in Elementary Mathematics Autograders

by

Stephanie Denise Carter Greene

Submitted to the Department of Electrical Engineering and Computer Science
on September 25, 2014, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

As schools grow more crowded and required testing outcomes become more stringent, teachers experience increasing demands on their time. To ease this load, I designed a program that is able to give students thorough, automated feedback on their mathematics assignments. This will allow teachers to spend less time grading (and consequently more time on other activities that might better help their students) without losing any of the feedback and error correcting a human would be able to provide.

Based on a number of different test cases, using a wide variety of elementary algebra problems, the program can correctly identify the lines in which errors are introduced. The program is also adept at finding the precise error as long as the student has made minimal changes per step. If multiple changes have been made, the program is forced to make its best guess at the most likely error without resorting to testing hundreds of possible combinations.

Thesis Supervisor: Patrick H. Winston
Title: Ford Professor of Engineering
Acknowledgments

This work would not have been possible without the help of the following:

To Professor Winston, thank you for being a wonderful teacher and mentor. I greatly appreciate you agreeing to supervise my thesis and for giving me excellent comments to consider.

To all my teachers and professors throughout my twenty years of school, thank you for your dedication and drive to help others learn. You have all inspired me to work towards helping make education more accessible to all.

To Projects Abroad - Ghana, The Lighthouse Program, Phillips Academy, and the staff of 6.00 (Spring '13), 6.01 (Fall '11), 6.042 (Fall '11), and 6.005 (Fall '13 and Spring '14), thank you for letting me teach and assist in a variety of different classroom settings. This experience has been invaluable in helping me learn what is needed in the field of education.

To my family and friends, thank you for supporting me all my life and particularly during the last year as I worked on this project.

Finally, I dedicate my work on this project to my uncle, Major General Harold Greene. Uncle Harry, you always valued education and helping others. I hope this project exemplifies both and someday will help better education for students from all walks of life.
Contents

1 Introduction .................................................. 13
  1.1 Vision ..................................................... 15
  1.2 Motivation ............................................... 16
  1.3 Sample of Results ........................................ 17
  1.4 Background .............................................. 18
    1.4.1 Autogradiers ........................................ 18
    1.4.2 Text Recognition .................................... 19

2 Design ......................................................... 21
  2.1 Program Overview ........................................ 21
  2.2 Supported Mathematics ................................... 22
    2.2.1 Basic Arithmetic ..................................... 22
    2.2.2 Number Types: Decimals, Fractions, and Negatives . 22
    2.2.3 Variables ............................................. 22
    2.2.4 Equations ............................................. 23
    2.2.5 Grouping .............................................. 24
    2.2.6 Exponents ............................................ 24
  2.3 Precise Error Pinpointing ................................ 24

3 Program Outline ................................................. 29
  3.1 Overall Goal ............................................. 29
  3.2 Step 1: Character Finding ................................ 30
  3.3 Step 2: Character Recognition ............................ 31
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4 Step 3: Text Parsing</td>
<td>33</td>
</tr>
<tr>
<td>3.5 Step 4: Answer Comparison</td>
<td>34</td>
</tr>
<tr>
<td>4 Evaluation</td>
<td>37</td>
</tr>
<tr>
<td>4.1 Testing</td>
<td>37</td>
</tr>
<tr>
<td>4.1.1 End to End Testing</td>
<td>37</td>
</tr>
<tr>
<td>4.2 Results and Limitations</td>
<td>39</td>
</tr>
<tr>
<td>5 Discussion</td>
<td>41</td>
</tr>
<tr>
<td>5.1 News</td>
<td>41</td>
</tr>
<tr>
<td>5.2 Contributions</td>
<td>41</td>
</tr>
<tr>
<td>5.3 Future Work</td>
<td>42</td>
</tr>
<tr>
<td>5.3.1 Handwriting Recognition</td>
<td>42</td>
</tr>
<tr>
<td>5.3.2 Usability</td>
<td>43</td>
</tr>
<tr>
<td>5.3.3 Error Finding</td>
<td>43</td>
</tr>
<tr>
<td>5.3.4 Other Mathematics</td>
<td>44</td>
</tr>
<tr>
<td>A Unit Tests</td>
<td>45</td>
</tr>
<tr>
<td>A.1 Mathematics Functions and Terms</td>
<td>45</td>
</tr>
<tr>
<td>A.1.1 Numbers</td>
<td>45</td>
</tr>
<tr>
<td>A.1.2 Variables</td>
<td>45</td>
</tr>
<tr>
<td>A.1.3 Addition</td>
<td>47</td>
</tr>
<tr>
<td>A.1.4 Subtraction</td>
<td>48</td>
</tr>
<tr>
<td>A.1.5 Multiplication</td>
<td>49</td>
</tr>
<tr>
<td>A.1.6 Division</td>
<td>50</td>
</tr>
<tr>
<td>A.1.7 Exponents</td>
<td>50</td>
</tr>
<tr>
<td>A.1.8 Groups</td>
<td>51</td>
</tr>
<tr>
<td>A.2 Equation Testing</td>
<td>52</td>
</tr>
<tr>
<td>A.2.1 Character Parsing</td>
<td>53</td>
</tr>
<tr>
<td>A.2.2 Text Parsing</td>
<td>53</td>
</tr>
<tr>
<td>A.3 GUI Functionality</td>
<td>53</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>B</td>
<td>Single Character Test Files</td>
</tr>
<tr>
<td>C</td>
<td>Solution Test Files</td>
</tr>
</tbody>
</table>
List of Figures

3-1 Character Sorting Example: the numbers indicate the order that the characters would be sorted into. ............................... 31

3-2 An example of an image with the bounding boxes drawn around each character. Note that these bounding boxes are never drawn, this is simply used to illustrate the portions of the original image that are used for character comparison. ............................... 32

3-3 This is an example of the image shown to the student or teacher at the conclusion of the program. There is an error in the arithmetic going from the first step to the second step. To help the teacher pinpoint the problem, both steps are highlighted so the teacher knows immediately that there was some type of error between the two steps. ....... 35
Chapter 1

Introduction

Teachers at all levels of education have many tasks to perform. Depending on their teaching style and the level of their students, they may need to plan lessons, give lectures, hold office hours, meet with parents, create assignments, grade assignments, manage course staff, and a variety of other tasks. This does not even take into consideration the time many teachers donate to their students by coaching sports teams, supervising independent study, etc. or the fact that many schools are currently experiencing record levels of overcrowding [Gonzalez, 2014] [Monahan, 2013], requiring teachers to spend more time to educate each student as thoroughly as in previous years. With a limited number of hours in the day the more dedicated and overworked teachers may find they need to let something go.

If forced to choose, many teachers would opt to delay grading and focus their energies on planning the best possible lessons or assisting a struggling student. Planning better lessons allows the teacher to assist many students at once and assisting those known to be struggling has a much higher return on time invested than helping those who already have a solid grasp on the material. Contrast either of these activities with grading, where the teacher can benefit at most one student at a time and that serves primarily to assess what the student has already learned rather than helping them learn something new. While grading can help students struggling understand what they did wrong, the teacher usually cannot only grade confused students. In order to fairly assess everyone, the teacher must also spend time grading those who
do not need additional assistance. Even if the teacher could decide not to use any of the assignments for assessment purposes and simply choose to grade to help students learn from their mistakes, the teacher would need to spend valuable time attempting to uncover each student's misunderstanding to locate the error. This can be done much more effectively if the teacher could sit down with the student and walk through problems together.

Other teachers might choose to let grading fall by the wayside because it can be one of their least interesting tasks. It is frequently repetitive, particularly when grading assignments that are not open-ended. It can also be incredibly time consuming. The more feedback a teacher attempts to provide, the longer it will take to track down the errors and misconceptions that will allow them to provide that level of detailed feedback. Bored teachers who are not paying sufficient attention to the task at hand may find they need to regrade students later after grading inconsistently.

However, most classes include a graded component to accurately assess students' progress. No matter why a teacher may prefer not to, most find themselves grading anyway. Lately, with the increasing ubiquity of computers, teachers have begun to integrate computerized autograded assignments into their curricula. This allows them to devote the time spent grading elsewhere while simultaneously assessing their students fairly. Unfortunately, this is not yet an ideal solution. Many such tools, such as Mastering Physics, simply compare the student's final answer to the prerecorded correct answer [Kiley, 2011]. This is fine if there is one and only one correct answer, but as many frustrated students discovered, minor discrepancies in formatting can make a correct answer record as an incorrect one.

Nonetheless, this is not the biggest issue to consider. While frustrating, students can learn the formats such tools will accept or the tool can clarify for each question the expected answer format. Much more problematic is the widening disparity in educational quality between affluent and impoverished areas. All the autograding tools that I have researched require computer access for each student. While this is not a problem for schools with primarily well-to-do student bodies, schools in more disadvantaged areas may have difficulty affording up to date textbooks, let alone extra
educational tools.[Horvitz and Shanklin, 2002] The lack of proper school resources, the disproportionate overcrowding among schools in poorer areas, and the social issues that so often follow mean students in such situations already face an uphill battle to obtain the same level of education as some of their more advantaged peers, and the rise of tools requiring extra equipment will serve to widen that gap.

To help close the gap, those interested in education must make an effort to investigate new educational tools and resources that rely as little as possible on specialized equipment and instead make use of basic educational materials in new and inventive ways.

1.1 Vision

My goal was to design a program that allowed teachers to automate the grading of mathematics assignments without requiring any substantial revisions to the format of their pre-existing assignments. In other words, the goal was to allow teachers to immediately spend less time grading so that they could spend more time on more effective teaching activities.

While I wanted the program to be versatile, I particularly wanted to devise a program that could used by a teacher with minimal extra tools to foster adoption in lower income areas. As it stands, the program can be used by a teacher with nothing more than paper and pencils (for student completion of the assignments) and temporary teacher access to a camera and computer (to perform the actual grading). While many low-income areas do not have computers in schools, many communities have at least basic access to free or inexpensive communal computers. While I would ideally like to create something that does not require any additional tools at all, in this scenario only the teacher needs access to additional tools. It is far more feasible that a teacher would be able to find temporary computer access once a week for grading than all students locate a computer to use every night to complete assignments.

However, this program can also be used as an online tool, by simply not using the handwriting recognition step and instead pulling the characters directly from the
user's input. This would allow students who desire more practice to try as many
practice problems as they would like while receiving instant feedback about their
grasp of the material. While this usage clearly would not be practical for everyone in
poverty-stricken areas, students with families fortunate enough to own a computer or
motivated students with the means to get to a public library would be able to access
the tool on occasion and supplement their classroom time. While such online tools do
allow the educational gap to widen, particularly when used as part of high-performing
schools' curricula, they also allow motivated students from lower-performing schools
to close their own personal gap.

1.2 Motivation

Educational reform is a cause that has always been important to me. I have always
enjoyed helping others learn. This has prompted me over the years to volunteer or
seek teaching jobs in a variety of school districts. Having worked or volunteered
everywhere from a low-performing inner city school to an elite prep school in the
suburbs to an open air village school halfway across the world, I have observed a
wide variety of learning environments and thought long and hard about how best to
improve the expected outcomes for those worst off. While there are many social issues
that need addressing and many teaching techniques that can be used, ultimately I
believe I can make the biggest impact by creating educational tools. Tools can be
deployed quickly and can allow a small number of creators to make an impact on
hundreds of others. I decided on a grading tool primarily because it is not where a
teacher's attention is needed most. A grading tool can pinpoint a student's mistake
and tell them what they did wrong, but it takes a teacher to engage the students'
attention and reach out to each one.
1.3 Sample of Results

The program I have created allows users to automatically grade their students’ or their own solutions to elementary mathematics problems. One example follows. The user uploads a solution:

\[
2^3 + 4 \\
6 + 4 \\
10
\]

The program highlights the incorrect lines:

\[
\text{Highlighted: } 2^3 + 4, 6 + 4, 10
\]

For additional detail, the user can ask the program to guess the error. Based on the incorrect lines flagged, \(2^3 + 4\) and \(6 + 4\), the program guesses the user incorrectly reduced \(2^3\) to 4.
1.4 Background

1.4.1 Autogriders

As the internet became more ubiquitous and internet usage in the classroom became the norm, use of autogriders has increased. A wide variety of academic fields have found autograding applicable to their assignments. As a field requiring automated assignments, it should hardly be surprising that many classes have adopted automated grading as well.[Jean Mandernach and Wise, 2013] In MIT’s 6.005 (Elements of Software Construction), students are able to push their code to a repository where a series of tests are run against it automatically. Students are able to get a general sense of how well they are doing in real time. 6.005 also employs hidden tests whose results are only visible once the deadline for the assignment has passed.[Staff, 2014] This combination allows students to ensure they are on the right track while still requiring them to think critically about the assignment.

Other MIT classes, such as 8.01 (Physics I) also use autogriders. Mastering Physics, 8.01’s grader of choice, is on the less technical end of the autograding spectrum. Although it has been extended to accept slightly more input variations, it operates primarily by matching the student’s answer directly against the recorded answer. Unfortunately, in physics classes where evaluation is required, students may get a variety of answers that are equivalent to each other. Many students have been frustrated by Mastering Physics strict adherence to specific and sometimes vague format. The other downside to this method of simply entering a final answer is that it is fairly unforgiving with small mistakes.[Kiley, 2011] Imagine for example that a student accidentally dropped a negative while solving the problem. They could have done everything else perfectly and on a test they would probably have lost less than 20% of the points for the problem. However, Mastering Physics doesn’t have access to the student’s work and has no definitive way of knowing whether the student was on the right track or way off, so small errors cost them far more.
1.4.2 Text Recognition

While I don’t know of any other autograders that attempt to read and process scanned text, there are a number of applications that perform text recognition. Rather than attempting offline recognition as my program does, most of these applications do online text recognition. Here, offline text recognition refers to a process that uses previously created text. For example, it might try to process a typed document or a scanned image. Online text recognition refers to a process that is able to capture text while it is written. For example, such a process might use information from a digital pen. This gives the recognition program more information to use when determining the most likely match. It may be able to capture pressure or temporal information. This additional information increases the likelihood of an accurate character match.[Stanhovich, 2004]

However, in this use case, it is unlikely that there will be any of this additional information to work with as elementary, middle, and high school students rarely do their mathematics homework on anything besides paper. Fortunately, there are a number of people working on offline text recognition as well. This is frequently used to recognize typed text as the lack of irregularities make for easier recognition, but it can be applied to handwritten text as well. There are three main approaches to offline text recognition.[Hull et al., 1990]

1. Template Matching: This approach compares an unknown character with a database of known examples to determine the best match.

2. Structural Analysis: This approach segments the unknown character into a 5x5 grid. It then looks for certain features like horizontal or vertical strokes, holes, endpoints, etc. in each cell of the grid. These features, along with a few global features of the character, are used in a Bayesian classifier that determines the best character matches.[Davis and Ouyang, 2009]

3. Contour Analysis: This approach calculates the curvature along the inner and outer curves of the unknown character. Eight features are defined based on the
curvature at each point considering whether the curve is concave or convex, as well as the direction and location. These features are matched against a set of 130 rules to determine the character’s identity. This approach currently works only for digits, but is being expanded for letters as well.[Hull et al., 1990]

For this program, I elected to use template matching. Contour Analysis was clearly insufficient as it has not yet been expanded to recognize non-digit characters. Even the most basic mathematics problem would require additional characters. While structural analysis may very well be the most effective in the long run, I instead chose template matching for the time being. Template matching is much faster than structural analysis and it is at least as effective for consistently written characters. As this program relies heavily on offline handwriting recognition procedures that are not yet effective enough to support it, I selected the faster method as a proof of concept. Assuming the offline text recognition works as specified, the rest of the program will work as specified as well.
Chapter 2

Design

2.1 Program Overview

This program allows users to select the scanned or typed problem they would like graded. The program uses character matching to determine what characters are present in the image. To grade the text, the program may do one of two things depending on the preferences of the user. It may parse each line of text found into abstract datatypes representing mathematical operators and terms and evaluate each line of text. If a pair of lines do not evaluate to the same answer, the solver must have made an error at that step. For a more detailed, but potentially less accurate error determination, the program may compare pairs of lines to determine what exactly the student changed at each step and attempt to guess the specific error. The next chapter will focus on each of these steps in greater detail. However, this chapter will describe the challenges introduced by the addition of each mathematical concept and the precise error finding method.
2.2 Supported Mathematics

2.2.1 Basic Arithmetic
The program supports basic arithmetic including addition, subtraction, multiplication, and division. To do this the database of known examples used for character matching includes an example character for each operation as well as the digits from zero to nine. Each abstract data type's evaluate method simply evaluates each of its terms and combines them with the corresponding operator.

2.2.2 Number Types: Decimals, Fractions, and Negatives
The program currently supports several different types of numbers. As with basic arithmetic, a period was added to the known example database to support decimals. A fraction bar was not needed as it appears indistinguishable from the subtraction symbol. Similarly, an additional symbol was not required to support negatives. This requires the program to determine when parsing whether a "-" is a fraction, a minus sign, or a negative. Assuming the input is well formed, a "-" used as a negative should follow an operator or appear at the beginning of an expression. A "-" used as subtraction should always follow a number or variable. And a "-" indicating a fraction should be have a small number above and below it. Due to the way the letter sorting occurs, the numerator should precede the fraction bar and the denominator should follow it. When translating the symbols, the program must simply check whether the preceding symbol was above the bar and whether the following symbol is below it.

2.2.3 Variables
This allows the program to support basic algebra. The known examples database is currently extended only with "x" as a proof of concept, but can easily be extending to support other letters that do not appear too similar to number. In other words, it would not be wise to include the letter "o" as a potential variable as it may be confused with the number 0. Because their usages are similar, it would be very
difficult to distinguish between twenty (20) and two times variable "O" (2O).

2.2.4 Equations

The character recognition step was first extended to recognize the equal sign. It is not recognized as a single character by the character finding step as it is not contiguous, so the character recognition step must look for two subtraction symbols over each other. Similarly to the procedure described above for recognizing fractions, because the symbols are sorted from top to bottom then left to right, they will appear next to each other in the list. Should the character recognition step find two "-" symbols back to back, it checks their original locations to determine whether they are on top of each other.

This added functionality changed the answer comparison step significantly. The evaluation methods were required to solve an equation rather than simply evaluating an expression. It does this by reversing the operations, just as one would do when solving it on paper. First it evaluates each side as an individual expression so that terms that can be combined are combined. Then it reverses each operation one by one trying to leave all variables on the left and all numbers on the right. For example if the original problem was:

$$x + 3 - 4x = -9$$

It will be parsed into:

$$Equation(Subtract(Add(Variable(x), Number(3)), Variable(4, x)), Number(-9))$$

Each side will be evaluated, turning it into:

$$Equation(Subtract(3, Variable(3, x)), Number(-9))$$

Then 3 will be subtracted from both sides leaving us with:

$$Equation(Variable(-3, x), Subtract(Number(-9), Number(3)))$$

Finally both sides will be divided by -3 to leave us with:

$$Equation(Variable(x), Divide(Subtract(Number(-9), Number(3)), -3))$$

Because the variable $x$ is the only thing remaining on the left side of the equation, the program has solved it. The right side of the equation can be simplified using the initial evaluate method to find that $x = 4$. 

23
2.2.5 Grouping

Just as with the other additions, new characters for the left and right parentheses were added to the known examples database. This only required a simple change to the parser generated by Antlr. Any text that falls inside a left and right parenthesis must be processed first. This was achieved simply by using Antlr’s rule priority. Antlr always chooses the first rule that applies in a set of alternatives. To force it to handle grouping first, the grouping rule was simply placed before the exponent rule, which was placed before the multiplication and division rules, etc.

2.2.6 Exponents

Unlike the other extensions, this did not require any new characters in the database as exponents don’t contain any special characters. To determine whether a number is likely to be an exponent, the program keeps track of the current line height (defined as the minimum and maximum y-value of any character in the line. If the middle of a symbol is in the top \( \frac{2}{3} \) of the line, it is presumed to be a variable. When the text parsing step converts the equation information to strings, a \(^\) (caret) is added in front of the exponent to retain the necessary information. Otherwise, the parser would have no way of knowing that the exponent was distinct from any other number. The character parsing step also wraps the entire exponent in a group. This is simply done to ensure that the parser is aware of the entire exponent. Were the group not present, the parser would assume that only the first character was part of the exponent.

2.3 Precise Error Pinpointing

While it is helpful for students and teachers to be able to pinpoint the line with their error, it would be more helpful if they were told exactly what the problem is. To do this, after the character recognition step, once the program has strings representing each line, it compares neighboring pairs of strings to find the differences. That enables it to determine exactly what operations they evaluated at each step. For example, if
the problem is:
2 \times 3 + 8 \div 2
and the student’s second line is:
6 + 4
The program matches the parts the two strings have in common (here just a plus) which indicates that the student must have reduced 2 \times 3 to 6 and 8 \div 2 to 4. By translating these pieces into my abstract data types and comparing the answers of matching pieces on neighboring lines, it can pinpoint the precise error.

This is done using by applying the longest common subsequence problem to each pair of lines of the student’s work. I have made the assumption that students do not arbitrarily rearrange their expressions and equations. Usually if they are doing so it is to combine two terms or prepare to do so. By finding the longest common subsequence, the program can match as many similar parts between the two as possible. This also helps to avoid the scenario when two consecutive lines happen to have two terms that are the same, but effectively unrelated (i.e. another expression just happened to evaluate to the same value). One the program has found the matching components, it compares the sections between the matching components to see if one evaluates to the other. For example say the student has the following:
1 + 2 \times 3 + 7
1 + 6 + 7
The longest common subsequence between the two is comprised of 1+ and +7, leaving 2 \times 3 and 6. Because 2 \times 3 does evaluate to 6, the student has likely done the problem correctly. However, this must be verified. The program will check whether the both lines evaluate to the same value. If they do, the problem was done correctly, if they do not, the student has made an order of operations error. For example, say the student has written the following:
2 \times 3 + 4 \times 5
2 \times 7 \times 5
The longest common subsequence between the two is comprised of 2\times and \times5, leaving 3 + 4 and 7. Because 3 + 4 does evaluate to 7, it appears the student may have
done the problem correctly. However, the first line evaluates to 26 and the second evaluates to 70. The student has done the addition too soon. The program then checks the neighboring operations. If they have a higher priority then the current one (i.e. in this example the neighboring operations are multiplication and the current operation is addition), then the student has made an order of operations error.

This will find the correct mistake (or lack thereof) in the majority of cases where the student has not made significant changes (i.e. combining many steps into one).

If there are multiple non-matching sections that do not evaluate to their corresponding section, they will be carefully recombined and evaluated as a whole. This must be done carefully, because they are not necessarily able to be combined directly. For example, one section may be more nested than the other. Other matching pieces of the equation may need to be re-added to be able to accurately evaluate the equation.

For example, imagine one line is the expression \(4 + 3 + (2 \times 5)\). When compared to the next line of the student’s work, the only non-matching sections are 3 and \((2 \times .\). These cannot simply be recombined. The 2 must be recombined with the 5 before it can be combined with the 3. However, if the expression were \(4 + 3 + (2 - 5)\), the program would check to see whether the parentheses are truly necessary. If they are not (i.e. if the parentheses contain only addition and subtraction), then the 2 and 3 can be safely combined.

These cover the cases where the student is combining terms in an expression or on a single side of an equation. However, the student may be solving an equation. In this case, if they are simply adding or subtracting on both sides, the above process may still work, however, if very little in the two consecutive lines match, particularly if the matching pieces are very small, this may indicate that the student did not simply combine terms, rather the student may have moved to solve the equation. In this case, the program will try to evaluate a single division, multiplication, or exponentiation step and repeat the above process. For example, if a student made a division error in a single term, the program will locate that issue once it also performs the same division and finds that all but one term matches.
To make the program assess the solution as much like a human teacher as possible, a few steps have been added. After the error has been initially classified, the program evaluates each term (in other words, each portion of the expression or equation separated by an addition or subtraction sign) and attempts to rematch the strings. This solves the following edge case:

Consider the following lines of work:

\[
2x + 1 - 3x + 5 \times 4 + \frac{6x}{2}
\]
\[
-x + 1 + 20 - 3x
\]

In this case, teachers would likely conclude that the student accidentally wrote a minus instead of a plus after evaluating \(\frac{6x}{2}\). They would probably come to this conclusion because the \(-3x\) term follows the 20 term. Rather than concluding the 20 is an anomaly, teachers might guess that the student correctly evaluated \(5 \times 4\) and simply wrote the following line incorrectly. The initial error finding step of the program would match the two \(-3x\) terms in each line, preventing the program from matching \(2x - 3x\) as \(x\) because it attempts to maximize the number of identical matches. This would misclassify the most probable error as the arithmetic error \("2x + \frac{6x}{2} \neq -x"\). On the second run through, the program will compare the lines:

\[
2x + 1 - 3x + 20 + 3x
\]

and

\[
-x + 1 + 20 - 3x
\]

When maximizing the number of matches between the two, the program matches +1, +20, and the final 3x in each line. This leaves the first line with \(2x - 3x +\) and the second with \(-x-\). The program combines the two \(x\) terms in the first line and successfully matches it to \(-x\) in the second. This leaves the two operators, + and − as the only mismatch. To determine whether the error \("2x + \frac{6x}{2} \neq -x"\) or \("+ miswritten as \(-"\) is more likely, I rely on the assumption that students are more likely to make fewer mistakes. Because the error \("+ miswritten as \(-"\) is the result of fewer changes, the program declares this one more likely.
Chapter 3

Program Outline

3.1 Overall Goal

I wanted to create a program that requires as little effort as possible on the part of the teacher. Ideally the program should be able to use assignments as teachers currently create them. To make the problem more tractable, I have made a few simplifying assumptions, namely that the page includes no irrelevant text and that students write in straight lines.

Irrelevant text may make the processing take much longer as the program would need to determine which characters should be removed. In the absence of any suggestive details (like distance from the other characters or characters that could not possibly be present in the solution) the program would have to guess which characters should be removed and try rerunning each of the new solutions. An inaccurate guess would render the program's error finder useless. So, for best results, the page should include no irrelevant text.

Students must write in straight lines so the program can accurately figure out which characters on the page belong to which step. If the student writes haphazardly, the program has no way to tell what is being done in each step. It will be forced to guess (potentially rendering its error finding useless) or it will need to try all reasonable combinations (greatly increasing the time spent on grading). So, for accuracy and efficiency, students must write each step in a straight line.
3.2 Step 1: Character Finding

To quickly and accurately determine what characters lie on the page, the program must first separate each character. To do this, it runs through the pixels of the image, searching for contiguous dark sections. It first searches for all dark pixels in the image and stores them in a set. Here, in a black and white image where every pixel is represented by an integer from 0 to 255, a darkly colored pixel is defined as one with a value less than 200. The program then chooses a pixel \((x, y)\) and recursively searches for other contiguous dark pixels that are still in the set by checking whether the eight surrounding pixels at coordinates \((x+1, y+1), (x+1, y), (x+1, y-1), (x, y+1), (x, y-1), (x-1, y+1), (x-1, y), \) and \((x-1, y-1)\) are in the set. If one of these eight are still in the set, meaning they are darkly colored and not yet searched, the pixel is removed and the recursion continues from that pixel. As it is doing this, it keeps track of the smallest and largest \(x\) and \(y\) coordinates that it has encountered. The recursion stops when there are no neighboring darkly colored pixels that have not already been searched.

This step makes the assumption that the assignment is written in dark ink on lightly colored paper. This allows me to assume that every lightly colored pixel is part of the background and every darkly colored pixel is part of a character.

This step also assumes that text is written in straight lines. This allows it to keep track of the order the characters on the page. After the program finds the locations of each character (as represented by the smallest and largest \(x\) and \(y\) coordinates of their pixels), it sorts them so they are in the order a human would likely read them. It does this by repeatedly searching for the character that is farthest to the top left of the page, with the distance from the top of page taking priority over the distance from the left of the page. When it has found the relevant character, it is removed from the current list of characters and added to the sorted list. This is repeated until there are no unsorted characters remaining. In other words, this step sorts the characters from top to bottom and then from left to right so that the characters end up reading from left to right down the page.
Figure 3-1: Character Sorting Example: the numbers indicate the order that the characters would be sorted into.

This program handles minor deviations in straight lines as the example in figure 1 demonstrates. It does this by defining a line of characters as any characters whose range of y coordinates (the smallest and largest) overlap. Note that here y-axis is here defined from top to bottom and the x-axis is from left to right. The program looks for the leftmost character in the topmost remaining line. In the example, the program would consider 1, 2, and 3 to be the first line and 4 and 5 to be the second. This was originally designed to handle operations that are typically written only as superscripts or subscripts such as exponents, however it also helps with handwriting that is not perfect linear and parallel to the top of the page. This approach cannot handle serious deviations. If the student intended 1, 2, and 5 to be in a line the program would be unable to recognize this.

In my tests, this step was run on typed examples. Unfortunately handwriting recognition is not yet accurate enough to support this system. However, using typewritten examples allowed me to build a proof of concept. Once handwriting recognition is viable, the method that recognizes each character can simply be substituted for one implementing the new algorithm as all subsequent steps operate independently.

3.3 Step 2: Character Recognition

Once the program has determined where in the image each character lies, it must determine what character each represents. It does this by comparing each character to known examples and taking the one that matches best. The program has a
Figure 3-2: An example of an image with the bounding boxes drawn around each character. Note that these bounding boxes are never drawn, this is simply used to illustrate the portions of the original image that are used for character comparison.

A database of characters in black ink on a white background. Each one cuts out as much whitespace as possible. In other words, there is no row or column in the 2D array of the image pixels that does not have a darkly colored pixel. Because the program has segmented the initial image in the same way (it separates each character according to its bounding box defined by the top-most and bottom-most x-coordinates and the left-most and right-most y-coordinates), this will provide the best comparison.

Then each known example is scaled to the precise dimensions of the segmented character. Then the corresponding pixels are compared. Each pixel is defined to be a match if the value at the corresponding pixel is within 5. The example with the most matching pixels is declared to be the best match and the program adds this character to a string representation of the text. Because the program has previously sorted each character into the order a human would likely read and maintains this order during the character recognition step, the string represents the text in order. However, the string does not keep any information about where on the page the characters occur, so all information about where the line breaks are would be lost. To maintain this information, whenever the program finishes parsing a full line of characters the current string is stored in a list and a new string is started for the next line. Because the strings lose all information about where on a page each character is, operations that rely on character placement may be misrepresented if care is not taken. To deal with this, the position of the new character is checked against the
previous characters in the line. In the case of exponents, if the next character is half
the size or less of the line thus far and the middle of the character is located at least
seventy-five percent of the way up the line, then the new character is declared to be
an exponent. To keep track of exponents in the string a carat is added preceding
the new character. Because exponents may contain multiple characters, exponents
are wrapped in parentheses. If the next character is also deemed to be part of the
exponent, it will be added inside the existing parentheses.

3.4 Step 3: Text Parsing

Next the program must parse the text. How this is done will vary for each type of
math problem that is supported. In the interest of brevity, I will only fully describe
here how the program will handle basic arithmetic as might be found in a second or
third grader's homework.

This step converts the strings representing each line of the problem to recursive
data types representing basic algebra such as addition, exponents, or variables. This
step has two substeps, lexing and parsing. First the program consumes the character
stream to convert each symbol into a token. There is a token for each operation,
variables, numbers, etc. This step makes sure to combine consecutive digits to form
the full number. Then the program parses the token into correct abstract data type
for each type of expression, equation, operation, variable, or number. The order
of operations is preserved by greedily matching each type of operation. First, we
match equations, then exponents, then multiplication and division, then subtraction
and addition, then variables and numbers, etc. If the input is well formed, there is
a non-operation token or parsed expression on each side of an operation token. In
other words, two operations should not appear consecutively. Note, that two minus
symbols may appear consecutively. However, the lexer will recognize the first as a
subtraction symbol and the second as a negative sign, assuming the rest of the input
is well formed. If it is not, an error is thrown. Otherwise, it takes these two numbers,
variables, or expressions and initializes the relevant abstract data type.
The parsing was done using a recursive descent parser generated by Antlr. That means Antlr will match each operation in the order indicated above, but then it must parse either side to determine what expressions lie on either side of the operation.

For example if the program were parsing the following:

\[ 15 - 4 \times 7 \]

the tokens would be:

15, -, 4, \times, 7

As this is an expression containing no exponents, the parser will match the multiplication first. In this parser, a multiplication expression is defined as a multiplication symbol with an expression on either side. So it recognizes the first expression as 4 and the second as 7.

Now the parser recognizes that it has \( 15 - \) an expression (namely \( 4 \times 7 \)). So it matches the subtraction expression next, where a subtraction expression is defined similarly to the multiplication, as expression - expression. It matches the 15 as the first expression and the \( 4 \times 7 \) as the other.

In the general case, this process is repeated first for an equation if present, then for exponents. When there are none remaining, the program looks for multiplication and division operators and repeats the process. It then proceeds with addition and subtraction and finally with terms. Processing in this order ensures that multiplication and division will always be more nested than addition and subtraction and exponents will be more nested than multiplication and division, etc. So when the program attempts to evaluate the answer, the multiplication and division will be performed first as the addition and subtraction problems will rely on the answers to the multiplication and division problems, thus preserving the order of operations.

3.5 Step 4: Answer Comparison

Finally, the program can use these instances to determine whether the student has done the problem correctly. Initially, the program simply checks whether each line of the problem evaluates to the same value. More specifically, it checks that neighboring
Figure 3-3: This is an example of the image shown to the student or teacher at the conclusion of the program. There is an error in the arithmetic going from the first step to the second step. To help the teacher pinpoint the problem, both steps are highlighted so the teacher knows immediately that there was some type of error between the two steps.

pairs of steps in the student’s work evaluate to the same value. This is done by simply calling the evaluate method for each abstract data type. Just like with the text parsing step, the specific implementation of the evaluate method will depend on the type of mathematics problem the program is grading. So again, I will just describe how the program handles basic arithmetic problems.

The evaluate methods for the four abstract data types operate in the same way. They call the evaluate method for each of their parameters (the left and right sides for subtraction and division and for each term for addition and multiplication) and combine the answers they receive with their corresponding operator. Evaluate for a string of digits simply returns the integer value of those digits (i.e. “14” returns 14) so the recursion stops once the evaluate method is called on the innermost data type.

If some pair does not evaluate to the same value, we can conclude that the student has made a mistake at that step. The program both returns a string listing the steps with errors and displays an image with the problematic steps highlighted.
Chapter 4

Evaluation

4.1 Testing

The following section describes the end to end tests I ran to ensure the entire program worked as specified. Unit tests are described in Appendix C.

4.1.1 End to End Testing

In these tests, the functionality of the entire program is tested by grading examples in the GUI. Each of the examples in Appendix C were run through the autograder. The following is the expected output.

<table>
<thead>
<tr>
<th>Test</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example B-1</td>
<td>Correctly Solved</td>
<td>Tests multiplication before addition</td>
</tr>
<tr>
<td>Example B-2</td>
<td>Correctly Solved</td>
<td>Tests basic multiplication and addition</td>
</tr>
<tr>
<td>Example B-3</td>
<td>Lines 1 and 2 highlighted</td>
<td>Tests wrong operator arithmetic error</td>
</tr>
<tr>
<td>Example B-4</td>
<td>Lines 1 and 2 highlighted</td>
<td>Tests basic arithmetic error</td>
</tr>
<tr>
<td>Example B-5</td>
<td>Correctly Solved</td>
<td>Tests multiple steps per line</td>
</tr>
<tr>
<td>Example B-6</td>
<td>Lines 1 and 2 highlighted</td>
<td>Tests basic arithmetic error</td>
</tr>
<tr>
<td>Example B-7</td>
<td>Correctly Solved</td>
<td>Tests characters close together</td>
</tr>
</tbody>
</table>
Example B-8 Correctly Solved Tests characters spread further apart
Example B-9 Correctly Solved Tests decimals
Example B-10 Lines 1 and 2 highlighted Tests incorrect operator arithmetic error
Example B-11 Correctly Solved Tests exponents
Example B-12 Correctly Solved Tests equation solving
Example B-13 Correctly Solved Tests grouping
Example B-14 Lines 1 and 2 highlighted Tests arithmetic error with multiple steps per line
Example B-15 Correctly Solved Tests exponents with multiple steps per line
Example B-16 Lines 1 and 2 highlighted Tests order of operations error

**Most Probable Error Determination**

In these tests, the ability of the program to guess the error made by the student is tested. Pairs of Strings representing consecutive lines of a student’s work are used as inputs to the findDifferences method which returns a String describing the most likely error.

<table>
<thead>
<tr>
<th>Test</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$4 \times 5 + 3, 18 + 3$</td>
<td>$5 \times 4 \neq 18$</td>
<td>Evaluating multiplication incorrectly</td>
</tr>
<tr>
<td>$1 + 3 \times 2, 1 + 5$</td>
<td>$3 \times 2 \neq 5$</td>
<td>Evaluating multiplication as addition</td>
</tr>
<tr>
<td>$2^3 + 4, 6 + 4$</td>
<td>$2^3 \neq 6$</td>
<td>Evaluating an exponent incorrectly</td>
</tr>
<tr>
<td>$2^{10} - 1000, 4$</td>
<td>$2^{10} - 1000 \neq 4$</td>
<td>Testing no similarities between the two lines</td>
</tr>
<tr>
<td>$2 + 3 \times 4 + 5, 5 \times O.O. Error: 2 + 3^{**}$</td>
<td></td>
<td>Testing evaluating addition before multiplication</td>
</tr>
<tr>
<td>$4 + 5$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.2 Results and Limitations

The expression and equation solving portion of the program can handle basic humansolvable algebra consisting of basic arithmetic operations, exponents, variables, and parenthesis. For example, it can solve equations like $x^2 + 4 = 20$ but it cannot solve $x^2 = 27$ as this equation does not have a straightforward methodology to follow to the solution besides guess and check. This means that the program can perform a simple evaluation (flagging the lines with errors only) of a student’s solution to any problem so long as it only consists of the elements previously indicated and has a solution that does not involve guessing and checking.

However, the most probable error determination method is somewhat less reliable. It can quickly and accurately identify simple errors when students are thoroughly showing their work. For example, if students only perform a small number (relative to the length of the problem) of steps per line and do not rearrange their work, the program can usually determine what changes they have made and whether those changes are accurate. However, if students do make many unnecessary rearrangements or they make many changes each line it becomes much more difficult for the program to tell what they were thinking. As most students do not arbitrarily rearrange their work or make so many changes it becomes difficult to tell what they simplified, the program should generally be able to guess their thought process. Unfortunately, the nature of the method is to look for errors and errors are by definition not always logical. As a result, students may not always make logical changes between steps, making their errors more difficult to find.
Chapter 5

Discussion

5.1 News

In short, this program allows users to automatically grade their students’ or their own solutions to elementary mathematics problems. Unlike other autograders currently available, this program does not simply check the final answer. Instead, based on the work the student has shown, the autograder attempts to guess their mistake. This allows students to better learn from the mistakes they have made. Sometimes the guessed mistake will be wrong (because mistakes need not be logical), but the program will always be able to flag the step that introduces the error. This enables the student to find their own mistake more quickly if the program is incorrect.

5.2 Contributions

In this thesis, I have described my two main contributions.

1. I developed a program that can take a scanned picture of a student’s solution to a problem, read it, and determine whether they have solved the problem correctly. While handwriting recognition is not yet good enough to make this a feasible solution for grading handwritten homework or class assignments, once it is, the method that performs the text recognition step can be easily switched out for one implementing the new algorithm. In the meantime, the program
can successfully interpret and grade typed text. This can currently be put to good use in online programs.

2. I also developed a method to classify errors students have made to allow the autograder to give more relevant feedback. This program operates by looking at two pairs of lines in a student’s work and attempts to guess, based on the differences between the two, what portions of the first line the student simplified to get the second line. This allows the program to figure out where the error lies based on the portions that have been incorrectly evaluated.

5.3 Future Work

5.3.1 Handwriting Recognition

Math homework is very infrequently typed before the college level. For this program to truly be useful to teachers it must recognize handwriting with reasonable accuracy. The easiest way to promote this would be to expand the database of known examples. Even if I only wanted to recognize my own handwriting, if I provided ten or so examples per character it would capture more of the natural variation of my writing. However, this can cause two problems. First, the larger the database, the longer it will take. Second, natural variation may cause two characters to appear very similarly. Having overlapping known examples will cause recognition errors. For example some people might write 8s with a very large top half and a small bottom half. This may end up conflicting with 0s.

To help alleviate these problems, the program can keep track of which known examples came from the same people. If it finds that a student more frequently matches examples from only a few people, it can try to first match characters with just their known examples. If it finds a match above a certain empirically determined threshold, it will take the best match, otherwise it will have to look through more of the dataset.

If it does have to look through more of the dataset, it will consider the top $x$
number of most likely matches, where \( x \) will be empirically determined and allow the top matches to cast a weighted vote on the outcome. For example if \( x \) is 3 and the top matches are 8, 0, and 0, with matches of 95\%, 90\%, and 85\% respectively, the program would declare the character a 0, even though the best single match was 8. In this case it might also be a good idea to keep track of the character I did not choose. Then if it turns out the equation is malformed with the selection it made or the selection causes an error, it could check to see if the next best guess fixes these problems.

### 5.3.2 Usability

To make the program as easy to use and intuitive as possible a few features should be added. First, the GUI should be extended to allow the teacher to automatically grade each file in a particular folder and export the graded options to the folder of their choice. This will allow teachers to simply press a button and leave to do something else. They should also be able to indicate a grading scheme. For example, they can indicate how many points each question is worth and how many points should be lost for various types of errors. This will allow the program to truly grade the students, rather than simply giving them feedback about where they made mistakes.

### 5.3.3 Error Finding

As mentioned previously in Results and Limitations (Section 4.2), sometimes the program has difficulty determining an error if a student makes less logical mistakes. To help alleviate this problem, the program should keep track of the locations of each of the characters. Students are far more likely to write their answer to a portion of the problem underneath the part it evaluates than anywhere else. For example, if the student is evaluating \( 2 + 3 \times 4 \), they are much more likely to write 12 underneath \( 3 \times 4 \) than anywhere else. Currently the program prioritizes the explanation with the fewest mistakes made by the student. The program could be extended to consider both explanations with fewer mistakes and explanations that assume the student wrote
their answers directly beneath the expression evaluated. The weighting between the two factors would need to be determined empirically.

5.3.4 Other Mathematics

The more types of mathematics the program can handle, the more useful it will be, particularly because more difficult math typically takes longer to grade than simpler math. The next step would be to extend the program to handle other types of algebra, such as systems of equations. Later on, other types of mathematics could be added, such as basic calculus.
Appendix A

Unit Tests

A.1 Mathematics Functions and Terms

A.1.1 Numbers

The following tests ensure that numbers can be properly instantiated. In this program numbers are implemented as a special case of Variables, so all other functionality is covered by the variable tests.

- Integers
- Negative Integers
- Decimals
- Negative Decimals
- Fractions
- Negative Fractions

A.1.2 Variables

- Variable Without Coefficient: this test checks that a variable can be properly instantiated without a coefficient. No errors should be thrown and the coefficient should default to 1.
- Variable With Coefficient/Variable With Decimal Coefficient/Variable With Fraction Coefficient: these tests check that a variable can be instantiated with
different types of coefficients.

- Invalid Variable Character: this test checks that a variable cannot be instantiated with invalid characters (such as the operator signs or parentheses).
- Variable Order Tests: these tests verify that the order function works as specified. This function is used to order variables in a consistent manner. It is used primarily when dividing to find the largest term. The order first prioritizes variables in alphabetical order, then by largest exponent, and finally by coefficient.

A list of representative test cases follows:

<table>
<thead>
<tr>
<th>Test</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2</td>
<td>2 &gt; 1</td>
<td>Both are numbers, so they are ordered numerically with the larger taking priority.</td>
</tr>
<tr>
<td>1, 1.5</td>
<td>1.5 &gt; 1</td>
<td>This verifies that decimals are interpreted correctly.</td>
</tr>
<tr>
<td>1/2, 1</td>
<td>1 &gt; 1/2</td>
<td>This checks that fractions are interpreted correctly.</td>
</tr>
<tr>
<td>1/2, 0.5</td>
<td>1/2 = 0.5</td>
<td>This checks that fractions and decimals can be accurately compared.</td>
</tr>
<tr>
<td>x, x^2</td>
<td>x^2 &gt; x</td>
<td>Here the variables are the same, but the exponent of the first is higher, so it has higher priority.</td>
</tr>
<tr>
<td>x, y</td>
<td>x &gt; y</td>
<td>The variables are different, so x should come first as they are ordered alphabetically.</td>
</tr>
<tr>
<td>x, 2</td>
<td>x &gt; 2</td>
<td>Variables take priority over numbers as numbers are the same as the variable to the 0th power.</td>
</tr>
<tr>
<td>2x, 2</td>
<td>2x &gt; 2</td>
<td>This checks to make sure that two different terms with the same coefficient are treated appropriately.</td>
</tr>
<tr>
<td>3x, 4</td>
<td>3x &gt; 4</td>
<td>This verifies that coefficients do not take priority over variables.</td>
</tr>
<tr>
<td>2x, 5y</td>
<td>2x &gt; 5y</td>
<td>This checks that different coefficients do not take priority over different variables.</td>
</tr>
</tbody>
</table>
\[
x, y^2\quad x > y^2\quad \text{Although the } y \text{ has a larger exponent, alphabetical order takes precedence over the order of exponential powers.}
\]
\[
x, x\quad x = x\quad \text{This verifies that two equivalent variables will have the same priority.}
\]
\[
xy, xy^2\quad xy^2 > xy\quad \text{This verifies that two similar multi-term variables with different exponents are ordered correctly.}
\]
\[
3xyz, 2xyz^4\quad 2xyz^4 > 3xyz\quad \text{This checks that two similar multi-term variables with different exponents are ordered correctly when different coefficients are present.}
\]
\[
3x, -2x^2\quad -2x^2 > 3x\quad \text{This test verifies that coefficients (both positive and negative) are taken into account after ordering by exponential power.}
\]
\[
-4x^2, 7y^2\quad -4x^2 > 7y^2\quad \text{This test checks that coefficients (both positive and negative) are taken into account after ordering alphabetically.}
\]

### A.1.3 Addition

The following tests check that the valid addition expressions can be instantiated and evaluated correctly.

<table>
<thead>
<tr>
<th>Test</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 + 1</td>
<td>3</td>
<td>Addition of two numbers</td>
</tr>
<tr>
<td>1 + 2</td>
<td>3</td>
<td>Verifying the commutative property of addition holds</td>
</tr>
<tr>
<td>1.5 + 2.4</td>
<td>3.9</td>
<td>Addition of two decimals</td>
</tr>
<tr>
<td>3.2 + 2</td>
<td>5.2</td>
<td>Addition of an integer and a decimal</td>
</tr>
<tr>
<td>(\frac{1}{2} + \frac{2}{3})</td>
<td>(\frac{7}{6})</td>
<td>Addition of two fractions</td>
</tr>
<tr>
<td>(\frac{4}{5} + 2)</td>
<td>(\frac{14}{5})</td>
<td>Addition of an integer and a fraction</td>
</tr>
<tr>
<td>(\frac{4}{3} + 2.5)</td>
<td>(\frac{23}{6})</td>
<td>Addition of a fraction and a decimal</td>
</tr>
</tbody>
</table>
1 + -2 \quad -1 \quad \text{Addition including a negative number}

x + 3x \quad 4x \quad \text{Addition of two matching variables}

x + 5 \quad x + 5 \quad \text{Addition of a variable and a number}

x + y \quad x + y \quad \text{Addition of two mismatched variables}

x + x^2 \quad x + x^2 \quad \text{Addition of matching variables with different exponents}

2x + 2y \quad 2x + 2y \quad \text{Addition of mismatched variables with the same coefficients}

3y + 3y^2 \quad 3y + 3y^2 \quad \text{Addition of matching variables with different exponents and the same coefficients}

x + y + 2x \quad 3x + y \quad \text{Addition of matching and mismatching variables}

\subsection{Subtraction}

The following tests check that the valid subtraction expressions can be instantiated and evaluated correctly.

<table>
<thead>
<tr>
<th>Test</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 - 1</td>
<td>3</td>
<td>Subtraction between two numbers</td>
</tr>
<tr>
<td>1.5 - 2.4</td>
<td>-0.9</td>
<td>Subtraction between two decimals</td>
</tr>
<tr>
<td>8 - 4.6</td>
<td>3.4</td>
<td>Subtraction between an integer and a decimal</td>
</tr>
<tr>
<td>3/4 - 1/5</td>
<td>11/20</td>
<td>Subtraction between two fractions</td>
</tr>
<tr>
<td>2 - 4/5</td>
<td>2/5</td>
<td>Subtraction between an integer and a fraction</td>
</tr>
<tr>
<td>4/3 - 1.2</td>
<td>2/15</td>
<td>Subtraction between a fraction and a decimal</td>
</tr>
<tr>
<td>2 - -3</td>
<td>5</td>
<td>Subtraction including a negative number</td>
</tr>
<tr>
<td>2x - 3x</td>
<td>-x</td>
<td>Subtraction between two matching variables</td>
</tr>
<tr>
<td>x - 1</td>
<td>x - 1</td>
<td>Subtraction between a variable and a number</td>
</tr>
<tr>
<td>1 - x</td>
<td>1 - x</td>
<td>Subtraction between a number and a variable</td>
</tr>
<tr>
<td>x - y</td>
<td>x - y</td>
<td>Subtraction between two mismatched variables</td>
</tr>
<tr>
<td>x - x^2</td>
<td>x - x^2</td>
<td>Subtraction between matching variables with different exponents</td>
</tr>
</tbody>
</table>
\[
\begin{align*}
5x - 5y & \quad 5x - 5y & \text{Subtraction between mismatched variables with the same coefficients} \\
2y - 2y^2 & \quad 2y - 2y^2 & \text{Subtraction between matching variables with different exponents and the same coefficients}
\end{align*}
\]

### A.1.5 Multiplication

The following tests check that the valid multiplication expressions can be instantiated and evaluated correctly.

<table>
<thead>
<tr>
<th>Test</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3 \times 4)</td>
<td>12</td>
<td>Multiplication of two numbers</td>
</tr>
<tr>
<td>(4 \times 3)</td>
<td>12</td>
<td>Verifying the commutative property of multiplication holds</td>
</tr>
<tr>
<td>(2.5 \times 1.5)</td>
<td>3.75</td>
<td>Multiplication of two decimals</td>
</tr>
<tr>
<td>(3 \times 4.2)</td>
<td>12.6</td>
<td>Multiplication of an integer and a decimal</td>
</tr>
<tr>
<td>(\frac{1}{3} \times \frac{2}{5})</td>
<td>(\frac{2}{15})</td>
<td>Multiplication of two fractions</td>
</tr>
<tr>
<td>(2 \times \frac{4}{5})</td>
<td>(\frac{8}{5})</td>
<td>Multiplication of an integer and a fraction</td>
</tr>
<tr>
<td>(\frac{4}{3} \times 1.5)</td>
<td>2</td>
<td>Multiplication of a fraction and a decimal</td>
</tr>
<tr>
<td>(2 \times -4)</td>
<td>-8</td>
<td>Multiplication including a negative number</td>
</tr>
<tr>
<td>(2x \times 5x)</td>
<td>(10x^2)</td>
<td>Multiplication of two matching variables</td>
</tr>
<tr>
<td>(x \times 5)</td>
<td>(5x)</td>
<td>Multiplication of a variable and a number</td>
</tr>
<tr>
<td>(x \times y)</td>
<td>(xy)</td>
<td>Multiplication of two mismatched variables</td>
</tr>
<tr>
<td>(x \times x^2)</td>
<td>(x^3)</td>
<td>Multiplication of matching variables with different exponents</td>
</tr>
<tr>
<td>(3x \times 3y)</td>
<td>(9xy)</td>
<td>Multiplication of mismatched variables with the same coefficients</td>
</tr>
<tr>
<td>(2y \times 2y^2)</td>
<td>(4y^3)</td>
<td>Multiplication of matching variables with different exponents and the same coefficients</td>
</tr>
<tr>
<td>(x \times y \times 2x)</td>
<td>(2x^2y)</td>
<td>Multiplication of matching and mismatching variables</td>
</tr>
</tbody>
</table>
A.1.6 Division

The following tests check that the valid division expressions can be instantiated and evaluated correctly.

<table>
<thead>
<tr>
<th>Test</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3 \div 2$</td>
<td>1.5</td>
<td>Division of two numbers</td>
</tr>
<tr>
<td>$1.5 \div 2.5$</td>
<td>0.6</td>
<td>Division of two decimals</td>
</tr>
<tr>
<td>$4.5 \div 3$</td>
<td>1.5</td>
<td>Division of an integer and a decimal</td>
</tr>
<tr>
<td>$\frac{1}{3} \div \frac{1}{2}$</td>
<td>$\frac{2}{3}$</td>
<td>Division of two fractions</td>
</tr>
<tr>
<td>$3 \div \frac{2}{5}$</td>
<td>$\frac{15}{2}$</td>
<td>Division of an integer and a fraction</td>
</tr>
<tr>
<td>$\frac{4}{3} \div 0.5$</td>
<td>$\frac{8}{3}$</td>
<td>Division of a fraction and a decimal</td>
</tr>
<tr>
<td>$4 \div -2$</td>
<td>-2</td>
<td>Division including a negative number</td>
</tr>
<tr>
<td>$4x \div 2x$</td>
<td>2</td>
<td>Division of two matching variables</td>
</tr>
<tr>
<td>$x \div 2$</td>
<td>$\frac{x}{2}$</td>
<td>Division of a variable and a number</td>
</tr>
<tr>
<td>$1 \div x$</td>
<td>$\frac{1}{x}$</td>
<td>Division of a number and a variable</td>
</tr>
<tr>
<td>$x \div y$</td>
<td>$\frac{x}{y}$</td>
<td>Division of two mismatched variables</td>
</tr>
<tr>
<td>$x \div x^2$</td>
<td>$\frac{1}{x}$</td>
<td>Division of matching variables with different exponents</td>
</tr>
<tr>
<td>$4x \div 4y$</td>
<td>$\frac{x}{y}$</td>
<td>Division mismatched variables with the same coefficients</td>
</tr>
<tr>
<td>$2y^2 \div 2y$</td>
<td>$y$</td>
<td>Division between matching variables with different exponents and the same coefficients</td>
</tr>
</tbody>
</table>

A.1.7 Exponents

The following tests check that the valid exponential expressions can be instantiated and evaluated correctly.

<table>
<thead>
<tr>
<th>Test</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$50$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A.1.8 Groups

The following tests check that the valid grouped expressions can be instantiated and evaluated correctly.

<table>
<thead>
<tr>
<th>Test</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2 \times (3 + 4)$</td>
<td>14</td>
<td>Addition before multiplication</td>
</tr>
<tr>
<td>$(2 - 6) ÷ 2$</td>
<td>-4</td>
<td>Subtraction before division</td>
</tr>
<tr>
<td>$(x \times x^2)^3$</td>
<td>$x^{27}$</td>
<td>Multiplication before exponentiation</td>
</tr>
<tr>
<td>$(\frac{x}{2})^3$</td>
<td>$\frac{x^3}{8}$</td>
<td>Distributing an exponent over division</td>
</tr>
</tbody>
</table>
\[(x + 2) \times (x - 3)\]  
\[x^2 - x - 6\]  
Multiplication of an addition and a subtraction expression

\[\frac{x^2 + 5x + 6}{x+2}\]  
x + 3  
Division of two addition expression

\[(x + 1)^3\]  
x^3 + 3x^2 + 3x + 1  
Exponentiation of an addition expression

### A.2 Equation Testing

The following tests check that the valid equations can be instantiated and solved correctly.

<table>
<thead>
<tr>
<th>Test</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[x = 2]</td>
<td>[x = 2]</td>
<td>Tests an already solved equation</td>
</tr>
<tr>
<td>[x + 1 = 2]</td>
<td>[x = 1]</td>
<td>Tests the program’s ability to reverse addition</td>
</tr>
<tr>
<td>[x - 4 = 3]</td>
<td>[x = 7]</td>
<td>Tests the program’s ability to reverse subtraction</td>
</tr>
<tr>
<td>[x \times 3 = 12]</td>
<td>[x = 4]</td>
<td>Tests the program’s ability to reverse multiplication</td>
</tr>
<tr>
<td>[x \div 4 = 7]</td>
<td>[x = 28]</td>
<td>Tests the program’s ability to reverse division</td>
</tr>
<tr>
<td>[x^3 = 216]</td>
<td>[x = 6]</td>
<td>Tests the program’s ability to reverse exponentiation</td>
</tr>
<tr>
<td>[2x = 8]</td>
<td>[x = 4]</td>
<td>Tests the program’s ability to solve an equation</td>
</tr>
<tr>
<td>[2x + 1 + 3x = 26]</td>
<td>[x = 5]</td>
<td>Tests the program’s ability to combine terms and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>execute multiple steps to solve the equation</td>
</tr>
<tr>
<td>[3 \times (x + 1) = 9]</td>
<td>[x = 2]</td>
<td>Tests the program’s ability to solve an equation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>containing term grouping</td>
</tr>
</tbody>
</table>
A.2.1 Character Parsing

These tests checked whether the program was able to figure out what characters were present in each example file. For a list of example files see appendices B and C. Appendix B contains files that contain a single character. These tests verify that the program can correctly interpret each symbol. Appendix C is comprised of files that contain solved problems. These tests check that the program can correctly find and identify each letter and that it maintains the order of the characters so the outputted text reads the same as the original file.

A.2.2 Text Parsing

These tests checked that text was correctly parsed into abstract data types. The tests were performed on each line of each of the examples in Appendix C. The answers were verified with an equivalence method which returns true if two expressions or equations are mathematically equivalent. This method operates similarly to an equals method, but it is designed to ignore irrelevant differences. For example, it ignores parentheses that do not change the structure of the problem as in \((x + 1) + y\).

A.3 GUI Functionality

- Resize Window: This test checks that resizing the window also resizes the viewing pane for the current problem without causing problems for other elements.
- Change Current Example: In this test, the user clicks on a different example in the directory. The example displayed in the viewing pane should change to the highlighted one.
- Change Current Directory: In this test, the user clicks on the Change Directory button which should bring up a file selection window displaying the current directory. The user should navigate to the directory they would like to select and press Open. The display should change to reflect the example files present in the new directory.
• Grade Correct Solution: While displaying an example containing a correctly solved problem (such as that in Example C-1 of Appendix C), the user presses the grade problem button. The display should not display any highlighted lines as these indicate errors in the solution.

• Grade Incorrect Solution: When displaying an example containing an incorrectly solved problem (such as that in Example C-3 of Appendix C), the user presses the grade problem button. The display should display at least two highlighted lines which indicate the where the student made an error. In Example C-3 the first two lines will be highlighted as the student incorrectly evaluated $3 \times 2$ as 5.
Appendix B

Single Character Test Files

Example B-1: Number 0
0

Example B-2: Number 1
1

Example B-3: Number 2
2

Example B-4: Number 3
3
Example B-5: Number 4
4
Example B-6: Number 5
5
Example B-7: Number 6
6
Example B-8: Number 7
7
Example B-9: Number 8
8
Example B-10: Number 9
9
Example B-11: Decimal Point
.
Example B-12: Left Parenthesis

( 

Example B-13: Right Parenthesis

) 

Example B-14: Addition Symbol

+ 

Example B-15: Subtraction/Negative Symbol

- 

Example B-16: Multiplication Symbol

* 

Example B-17: Division Symbol

/ 

Example B-18: Variable x

x
Appendix C

Solution Test Files

Example C-1

\[ 2 + 2 \times 3 \]
\[ 2 + 6 \]
\[ 8 \]

Example C-2

\[ 3 \times 5 \times 2 + 72 \]
\[ 3 \times 10 + 72 \]
\[ 30 + 72 \]
\[ 102 \]
Example C-3

\[ 1 + 3 \times 2 \]
\[ 1 + 5 \]
\[ 6 \]

Example C-4

\[ 4 \times 5 + 3 \]
\[ 18 + 3 \]
\[ 23 \]

Example C-5

\[ 6 \div 2 + 3 \times 5 - 14 \]
\[ 3 + 15 - 14 \]
\[ 4 \]
Example C-6

\[ 4 \times 8 - 23 \]
\[ 28 - 23 \]
\[ 5 \]

Example C-7

\[ -3 + 5 \times 4 \]
\[ -3 + 20 \]
\[ 17 \]

Example C-8

\[ -3 + 5 \times 4 \]
\[ -3 + 20 \]
\[ 17 \]
Example C-9

\[ 3.2 + 2.3 \]
\[ 5.5 \]

Example C-10

\[ 2^3 + 4 \]
\[ 6 + 4 \]
\[ 10 \]

Example C-11

\[ 3^2 - 7 \]
\[ 9 - 7 \]
\[ 2 \]
Example C-12

\[ x^2 + 3 = 12 \]
\[ x^2 = 9 \]
\[ x = 3 \]

Example C-13

\[ 2 \times (3 + 4) \]
\[ 2 \times 7 \]
\[ 14 \]

Example C-14

\[ 2^{10} - 1000 \]
\[ 4 \]
Example C-15

\[2^{10} - 1000\]

\[24\]

Example C-16

\[2 + 3 \times 4 + 5\]

\[5 \times 4 + 5\]

\[20 + 5\]
Bibliography


Lori Horvitz and Mary Shanklin. Old, poor schools have more outdated, irrelevant books. Orlando Sentinel, November 10, 2002.


Swinton Hudson Jean Mandernach and Shanna Wise. Faculty activities and time commitments in the online classroom. 2013.


