Aggregation of Student Answers in a Classroom Setting

by

Amanda C. Smith

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

August 31, 2006

Author ______________________________
Department of Electrical Engineering and Computer Science
August 31, 2006

Certified by ______________________________
Dr. Kimberle Koile
Research Scientist, CSAIL
Thesis Supervisor

Accepted by______________________________
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

Aggregation of Student Answers in a Classroom Setting

by
Amanda C. Smith

Submitted to the
Department of Electrical Engineering and Computer Science

August 31, 2006

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In a typical class, an instructor does not have enough time to poll all students for answers to questions, although it would be the best method for discovering students' misconceptions. The aggregator module of a system called Classroom Learning Partner provides a solution to this problem by collecting answers students wirelessly submit on tablet PCs and placing them in clusters, which then are displayed to the instructor in histogram form. The student answers are compared, via syntactic parsing and similarity measures, to each other and to instructor-provided example answers to form clusters, which represent student misconceptions. In tests, the aggregator module consistently created relevant clusters, very similar to those created by humans working with the same data. Classroom Learning Partner, including the aggregator module, has been deployed successfully in an MIT introductory computer science class.

Thesis Supervisor: Dr. Kimberle Koile
Title: Research Scientist, CSAIL

ACKNOWLEDGEMENTS

Dr. Kimberle Koile, without whom this project would not be possible

The CLP Group, who not only created the rest of the system, but also provided encouragement and camaraderie

Alejandro Morales

“If you can dream it, you can do it.” –Walt Disney

Table of Contents

Table of Figures

1. INTRODUCTION

Technologies aimed at improving student learning have recently experienced a resurgence in interest thanks to the development of wireless technology, new hardware platforms, and the Internet. Much of the current focus has been on student learning outside the classroom. Cybertutor, for example, allows students to answer physics problems online, with step-by-step help and feedback if they are having difficulty. The Classroom Learning Partner (CLP, http://projects.csail.mit.edu/clp) has a different aim: improving student learning in the classroom. In a traditional classroom setting, an instructor does not have time to look at every student's answer to an in-class exercise, and thus some misunderstandings go uncorrected. CLP addresses this problem by allowing students to submit their answers to the instructor immediately through the use of tablet PCs and wireless technology. Similarly, the instructor can provide immediate feedback to the students by displaying and discussing some of the submitted answers. He or she can choose to address the most prevalent misconceptions, illustrated by submitted answers, rather than guessing at what mistakes the students are likely to make. A similar approach is taken by systems such as Personal Response Systems (PRS), which employs a wireless polling mechanism and a computer to instantly tabulate results. While these sorts of system provide support for in-class exercises, they limit the instructor to asking multiple choice and true/false questions [Draper, 2004]. CLP, because it uses tablet PCs for input, has the potential to support any form of written answer, including diagrams and sketches. [Koile and Shrobe, 2005; Koile and Singer, 2006] The research in this thesis constitutes a major component of the CLP project: the aggregation of these open-ended written answers, so they may be presented in a format that is easy for the instructor to interpret.

2. BACKGROUND

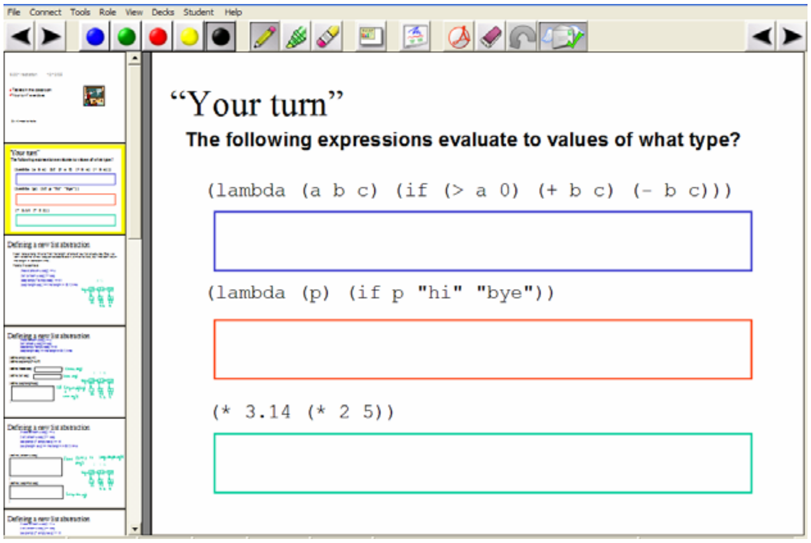
Classroom Presenter

The Classroom Presenter software, which is the basis for the CLP project, has the ability to allow students to send answers to the instructor during class. An instructor displays a Powerpoint-like slide presentation on her tablet, which also is displayed with an overhead projector and broadcast to student tablets [Anderson, et al. 2004; Anderson, et al. 2005]. Some of the slides contain in-class questions, to which the students can respond by writing directly on the slide in digital ink and sending the ink to the instructor tablet. The submitted slides are collected into a new slide deck on the instructor's machine, and can be displayed on the overhead projector and discussed in class. This scenario does not scale well to larger classes, as the instructor generally only has time to examine a handful of slides before it is necessary to continue with the class. Furthermore, the instructor probably would rather know how many students answered correctly instead of reading through many slides with near-identical correct answers, because these do not provide him or her with information about problems the students may be having. Successful aggregation of answers solves both of these problems by sorting similar student answers into categories, allowing the instructor to see common correct and incorrect answers at a glance and easily pull up student-created examples of these answers.

Classroom Learning Partner

In the full CLP system, the instructor is able to create question and answer sets using the Instructor Authoring Tool (IAT).[1] When an instructor creates such a set, he or she can enter one or more correct answers as well as any number of possible incorrect answers, along with a short description of the answer. This question and answer set is displayed on an instructor-viewable slide as part of a slide deck that is distributed to student tablets before class.

[1] The current version of the IAT was implemented by CLP group members Kevin Chevalier, Capen Low, Michel Rbeiz, and Kenneth Wu. [Chen 2005] describes an earlier implementation.



“Your turn”

The following expressions evaluate to values of what type?

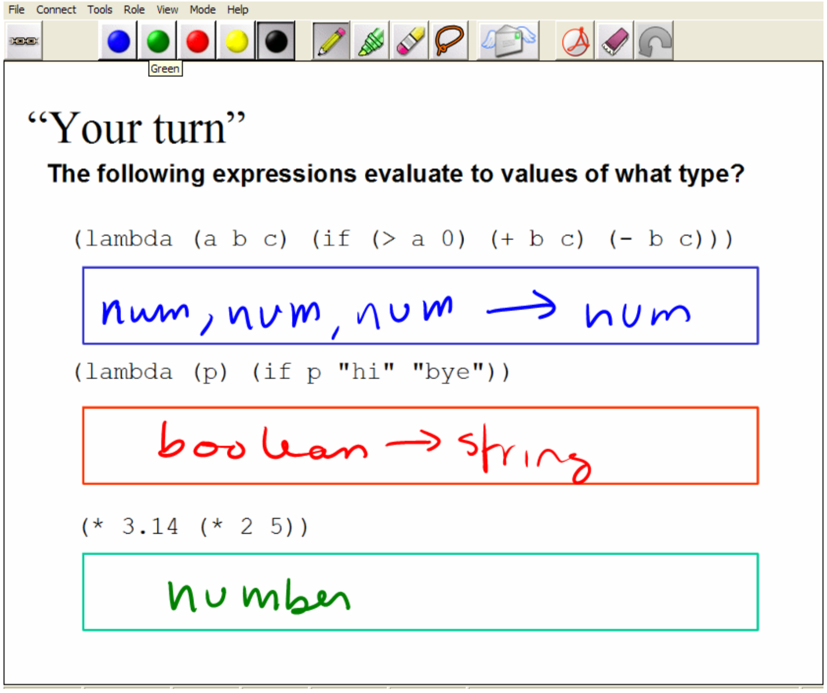
```
(lambda (a b c) (if (> a 0) (+ b c) (- b c)))
```

```
(lambda (p) (if p "hi" "bye"))
```

```
(* 3.14 (* 2 5))
```

Figure 1: A screenshot of a slide on the instructor’s machine in CLP [Koile & Singer, 2006]

During the lecture, the instructor displays the relevant slide (see Figure 1) and asks for student submissions.



“Your turn”

The following expressions evaluate to values of what type?

```
(lambda (a b c) (if (> a 0) (+ b c) (- b c)))
```

num, num, num → num

```
(lambda (p) (if p "hi" "bye"))
```

boolean → string

```
(* 3.14 (* 2 5))
```

number

Figure 2: A screenshot of a slide on a student's machine, with correct answers [Koile & Singer, 2006]

Students input answers directly on the slide in digital ink (see Figure 2), which is then processed into text format by a handwriting interpreter module

[Rbeiz, 2005]. The answers are transmitted to a central database, and when the instructor determines that students have had enough time to answer, he or she runs the aggregator. The aggregator obtains the student and instructor answers from the database and uses the instructor answers to evaluate the student answers. The student answers are then placed in “bins” based on their similarities to the instructor answers and amongst themselves. CLP fetches the completed bins from the database and constructs a histogram of the data for the instructor to examine (see Figure 3). The slides with the student's answers written on them are also presented to the instructor. Each slide has a colored mark corresponding to the bin in which the answer was placed, with the histogram serving as a color key, so that the instructor can quickly display an example of an answer from any bin.

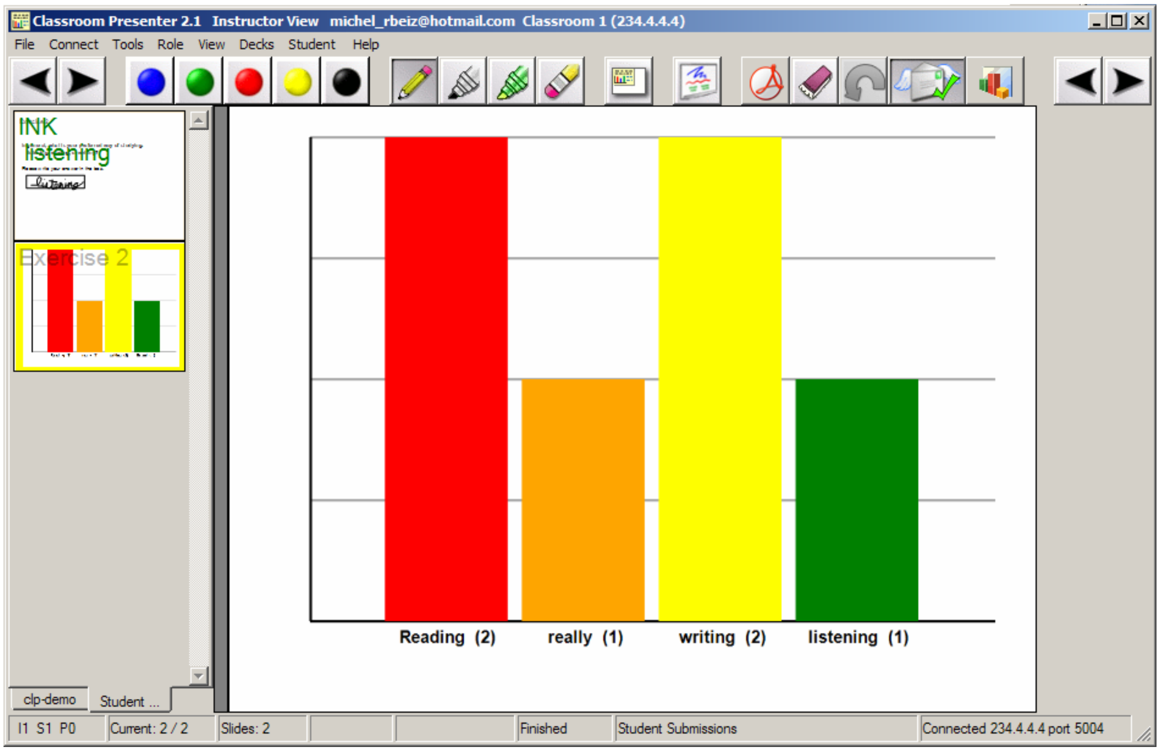


Figure 3: A histogram of aggregation results [Chevalier, 2006]

The system architecture for CLP is shown in Figure 4. Various components run on the instructor's tablet, the students' tablets, or a central server. In the current implementation, the aggregator runs on the instructor's tablet, and the database (aka repository) runs on a separate server, as database access on the tablet proved too slow.

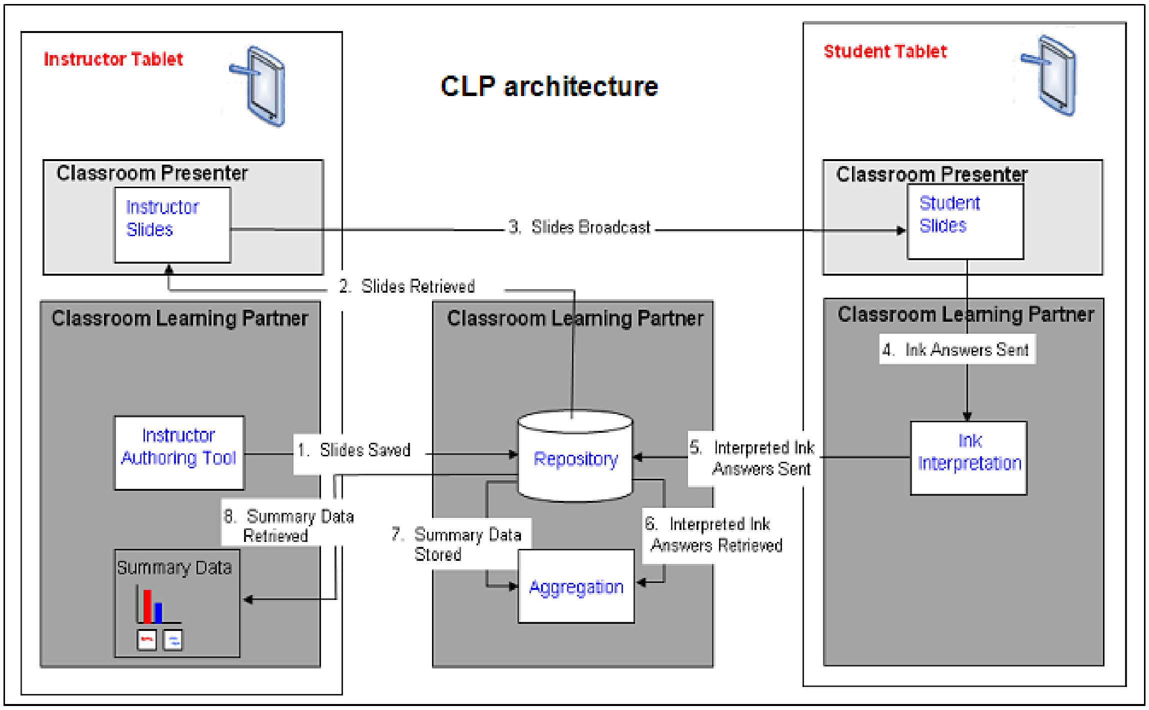


Figure 4: CLP Architecture and steps in using CLP

3. DESIGN

In designing the aggregator, the most important principle was modularity. Although the program is designed for use in MIT's introductory computer science course 6.001, Structure and Interpretation in Computer Programming [Abelson and Sussman, 1996], the intent was to create a program flexible enough to be used, with minimal additions, in any course. It is also beneficial to create a system where more advanced algorithms can replace old ones without affecting more than a single function. Another major consideration in the aggregator's design was efficiency. If the aggregator takes more than thirty seconds to return when clustering a classroom's worth of answers, it will not be a practical tool to use in class. The aggregator also must be robust. In particular, it should not be possible for a student to crash or otherwise negatively affect the aggregator through malformed input.

3.1 Programming Language

One of the most important early decisions was the choice of programming language for the aggregator. Since the CLP project as a whole is based on the Classroom Presenter code, the interpreter and the authoring tool modules were written in C#, the same language in which Classroom Presenter is written. C# also allows easy access to an MS SQL database, which was considered to be the easiest format for the central database. C#, however, is not necessarily the best language for the aggregator, and, since the aggregator is a separate module that only interacts with the rest of the system through the database and a notification scheme, other languages were considered for the aggregator. The aggregator, in particular, requires new functions to be created at runtime—the functions which check the student answers based on the provided instructor answers.

Lisp-based programming languages easily support the creation of functions at runtime. They treat functions as objects that can be passed freely to other functions. Lisp-based languages, furthermore, are not strongly typed, which allows great flexibility and reuse of functions for various purposes. Scheme, a variant of Lisp, was an early candidate for the aggregator's language, mostly due to the useful free development kits. Scheme, however, suffers from the fact that its object-oriented programming is extremely cumbersome. The decision was made, therefore, to write the aggregator in Common Lisp. Common Lisp has full capabilities for object-oriented programming, treats procedures as objects, and has a choice of several development programs that plug into the Emacs text editor to create a coding environment that is familiar to me and easily portable.

One problem with Common Lisp is that it does not have an interface for MS SQL, only for MySQL. This problem was solved by calling C# functions from within Lisp, as described in the section on integration.

3.2 Integration

Integrating a module written in Common Lisp with a system written primarily in C# provided its own interesting design challenges. It was decided early in development that the aggregator would essentially be a "black box" as far as the C# components are concerned. In keeping with this philosophy, the aggregator only interacts with the main system when it is notified to begin aggregating a batch of student answers, and when it adds and removes data from the database. CLP uses an MSSQL database, and there was no publicly available interface allowing Common Lisp applications to access an MSSQL database. A module known as RDNZL, which provides a wrapper for C# functions so they can be called from a Common Lisp application, provided the solution to this problem. With RDNZL, the aggregator was able to use the same procedures for database access as the other modules. Once the answer bins are in the database, CLP generates the histogram of the student answers and a filmstrip of the answers with their categories marked, and displays these on the instructor’s machine.

3.3 Answer Types

Possibly the most crucial early design decision was determining which types of answers the aggregator would focus on in its first iteration. Strings were deemed essential, because they are the answer type necessary for most basic questions asked in class, and many other possible answer types, such as code and diagrams, contain strings as subparts. Numbers were also implemented, since numbers are also a very common form of answer, and creating various procedures for comparing two numbers was relatively trivial. Sequences, i.e. ordered lists of answers, were also a crucial type, as this allows answers with multiple parts, as well as more complex answers. Currently the aggregator supports sequences of strings, which are not aggregated particularly differently from strings themselves, but eventually the aggregator will support sequences of other answer types, including sequences of sequences. The final answer types added to the aggregator were multiple choice and true/false. True/false questions are a special subclass of strings for which “t” is equal to “true” and “f” is equal to “false.” These last two answer types were added so that CLP could be used by instructors who may prefer PRS-style classroom interaction.

3.4 Knowledge Representation

The various CLP components, shown in Figure 3, all share a knowledge representation scheme that is centered on the notion of an answer. As shown in Figure 5, the class called Answer is the superclass of student and instructor answers.

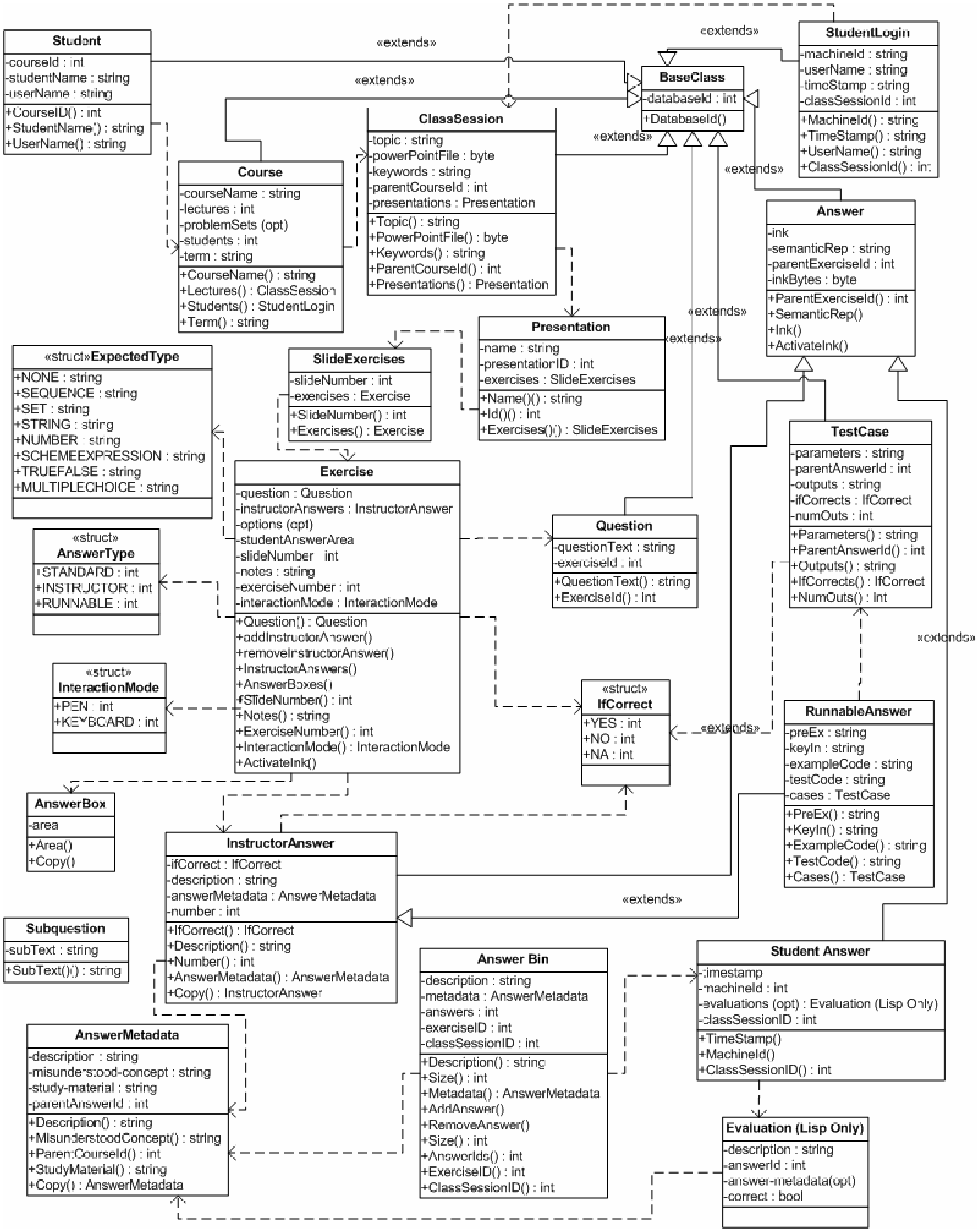


Figure 5: Object classes in CLP

Both student and instructor answers are created by requesting the appropriate data from the database. The text version of the answer is stored in the "interpreted ink" field, and the evaluations field of the student answer class is initially set to nil. The “evaluations” field will eventually hold an answer's associated evaluation objects. Each evaluation object is a description of how close the answer is to one of the instructor answers. The "ian," or instructor answer, field contains the interpreted ink of the associated instructor answer, the "description" field contains the description carried by the instructor answer, and the "score" field contains a numerical value of how close the student answer is to the instructor answer. This score ranges between zero and one, where zero represents an exact match, and one signifies that the two answers have nothing in common. These evaluation objects are generated by evaluation procedures. An evaluation procedure is a procedure generated at runtime that takes a student answer as input, calculates a score, creates an evaluation object, and adds the new evaluation object to the student answer's evaluations field. If the student answer already has evaluations, the new evaluation will be appended to the list. There is one evaluation procedure per instructor answer, so every student answer is compared with every instructor answer.

Another important object type is the answer bin. An answer bin contains a list of related answers and a description of the bin. The description is generated based on the answers it contains. If the bin contains a set of answers very similar to an instructor answer, the bin's description will match the instructor answer's description. If the bin contains a cluster that does not center around an instructor answer, the bin's description will be the most common answer in the bin with ties broken randomly. Answer bins are placed in the database, where they will be accessed by the main CLP program, upon completion of the aggregation.

4. IMPLEMENTATION

4.1 Aggregation Example

To illustrate the implementation of the design described above, it is easiest to walk through a simple aggregation of a small data set. For this example, I will use a data set drawn from a real question asked in a 6.001 lecture: "What is the value of the following Scheme expression: (- (+ 1 4) (* 2 (+ 4 1)))". Most students produced the correct answer, -5, but a few made mistakes, writing -50 (probably caused by violating Scheme's order of operations) and 5 (most likely a typo). This example aggregation uses the following data set: -5, -5, -5, -5, -5, -50, -50, -50, 5, 5. Before the aggregation, the instructor would have had to create the exercise in the Instructor Authoring Tool. When writing this exercise, the instructor chose "number" as the expected answer type and provided -5 as a correct answer, with no further information. When the students are given the question in class, the answers they write on the tablet are processed into text by the handwriting analyzer and placed in a database; this example assumes no errors originating from the handwriting analyzer. The instructor clicks the "aggregate" button, and the aggregation module is loaded.

The first step in the aggregation process is to obtain the submitted student answers and instructor information from the database. A list of student answer objects is created in the aggregator's runtime environment, each one containing a submitted student answer as a text string; likewise, the single instructor answer object is created with "-5" as its text string and "true" as its correctness designation. When the answer objects are created, the aggregator checks the expected type of the answer and performs any necessary pre-processing. In this case, the expected type is "number," and so the text strings from the database are changed into numerical values with a simple string-to-number function. The instructor answer becomes part of an "evaluator function." This evaluator function takes a student answer as input and produces as output a numerical value designating how close the student answer is to the answer "-5". When comparing numerical answers, the evaluator function returns the absolute value of the difference between the two numerical answers, and so a value of 0 designates a complete match. Each student answer is provided as input to the evaluator function. The result of the evaluator function is appended to the student answer in its "evaluation" field. Thus, after evaluation, the student answers containing -5 will be tagged as exact matches to the instructor answer, and the other student answers will be tagged as very different from the instructor answer.

Once evaluation is complete, the student answers are ready to be placed in bins. The actual bin objects are not created immediately. Rather, a hash map is used to store the temporary bins, with a representative answer from the bin as the key and the list of student answers contained by the bin as the value. The first bins created are those which center around instructor answers. All answers which match an instructor answer closely will be placed in the corresponding bin, and all other answers will be placed in a miscellaneous bin. In this case, all of the -5 answers are put in a -5 bin and the -50 and 5 answers are put into the miscellaneous bin. The next step is to create interesting clusters from the student answers in the miscellaneous bin. The aggregator contains a parameter for the maximum number of bins which should be created, and clusters will be created from the contents of the miscellaneous bin until the maximum number is reached or until no further clusters can be created. In this case, only two bins have been created, and the default maximum number of bins is seven, so the aggregator will try to split the miscellaneous bin up to five times.

In order to split a meaningful cluster from the miscellaneous bin, a logical cluster center must be chosen. This need leads to the question of which answer bins the instructor would most like to see. Obviously, the answer to this will be very different depending on the instructor and the question asked. One of the goals of this tool, however, is to allow instructors to see problems and mistakes common to several members of the class, so the instructor can discuss his or her class's particular misconceptions. With this in mind, the priority in choosing a cluster center is creating the largest possible cluster. Thus, when choosing a cluster center, the aggregator tests each student answer as a possible cluster center, and observes how many of the remaining student answers would be placed in that cluster. For this example, the contents of the miscellaneous bin are -50, -50, -50, 5, 5. Using -50 as a cluster center nets a new bin with three members but using 5 as a cluster center nets a new bin with two members, so -50 is chosen as a cluster center. A -50 key is added to the hash table, with a list of the three corresponding student answers as the value; the -50 answers are removed from the miscellaneous bin. The aggregator will then attempt a second split. In this case, only one cluster is possible, a cluster centered around 5 containing two student answers, and so it is created. At this point, the miscellaneous bin is empty and the aggregator will cease attempting to split new bins.

The final step is to create the actual bin objects from the hash table. Each key and its corresponding value are extracted from the hash table, and a new bin is created. The key is used as a bin description, the list of student answers is the bin's contents, and the number of student answers in the bin is the bin's size. The bin descriptions, contents, and sizes are placed in the database. The aggregation is then complete, and the bin data is used to create a histogram showing the frequency of various student answers, and to color-code the student answers as per the histogram.

The above data set was chosen specifically to illustrate the basic workings of the aggregator. Most sets of student answers are larger and contain considerably more variation. Here is a more diverse set of student answers, which are responses to the question, "What is the type of this Scheme expression:

(lambda (x) (if x "true" "false")) ?"

```
procedure
bool -> str.
(boolean) -> (string)
proc. boolean -> string
procedure
procedure: boolean -> string
bool -> string
string
boolean -> string
string
proc: bool -> string
compound procedure
proc (A -> string)
proc
boolean -> string
```

This set of answers poses a more interesting set of challenges. One issue with aggregating data such as the above is the issue of commonly-used abbreviations. An instructor would likely consider "bool" an acceptable alternative to "boolean"; likewise with "string"/"str" and "procedure"/"proc". Thus, the aggregator should not mark "bool -> string" as incorrect if the instructor's correct answer was "boolean -> string". Without context, it is nearly impossible for the computer to determine which substitutions are acceptable. Therefore, when the student and instructor answers are pulled from the database, any abbreviations are changed into the full form by a function which simply looks up known abbreviations for common terms used in the class. Furthermore, excess spaces and punctuation are stripped from the text; this step generally removes all punctuation except the "arrows" (->), but a "Scheme" option exists that keeps the parentheses, and other such options could be added for different contexts.

Strings and sequences of words must use a different similarity measure than the one used for numbers. Measuring how "alike" two strings are has a well-known dynamic programming solution which was employed here. In particular, the strings are compared character-by-character and word-by-word. A missing, extra, or incorrect character adds a point to a running score, and a missing or excess word adds three points to the running score. The two strings are compared in such a way as to minimize the point count. For instance, comparing the two strings "abc" and "ac" would result in a point count of 1, with the score calculated as follows:

```
a b c
a   c
0 1 1
```

as opposed to the possible score of 2:

```
a b c
a c
0 1 2
```

The three-point penalty for a missing or excess word is imposed so that the severity of the mistake of leaving out a word does not depend on the length of the word. Thus, if the correct answer is "foo bar baz quux," the answers "foo bar quux" and "foo bar baz" will be equally penalized. The final score ranges between zero and the length of the longer word, and thus can be changed into a percentage. When comparing two answers, the resulting percentage must fall below a certain "similarity parameter" for both answers to be placed in the same bin. In the example of "abc" and "ac", the result would be 1/3 or approximately 33% different, and thus the two answers would not be placed in the same bin if the similarity parameter is set to 10%. The aggregator also contains a similarity measure for use with multiple choice questions, which simply determines if the two answers (presumably, two letters or two numbers designating choices) are exactly the same.

4.2 Factors that Affect Aggregation

The most important factor affecting aggregation is the list of correct and incorrect answers provided by the instructor. Each of these answers will have a corresponding answer bin associated with it if at least one student gave a similar answer. Thus, instructor answers are a way to ensure that the instructor will always get to see the number of students who gave a particular answer. The number of instructor answers given also affects the number of computer-generated bins that are created, since there is generally a maximum number of bins the aggregator will create in total, and instructor-defined bins use up some of that allotment.

One very important consideration in the quality of the aggregation is the number and variety of answers. In particular, a large data set with a large variety of answers will generally result in many answers relegated to the miscellaneous bin, unless the maximum number of bins created is set to be very high. As an example, imagine a class of two hundred people, where no student answer is shared by more than four people. If the maximum number of bins is seven, then the aggregator will create seven bins of three or four answers each and then stop, leaving the vast majority of answers uncategorized in the miscellaneous bin. The aggregator provides its best results when used in small classes or on data sets with a small variety of answers. If the number of unique answers is less than the maximum number of bins, aggregation is trivial.

The order in which the instructor and student answers are provided to the aggregator will make no difference as to the final set of bins, with two exceptions. If a student answer is very similar to two different instructor answers, the bin in which it is placed is determined by the order in which the instructor answers were provided. Furthermore, if the aggregator is deciding between two student answers that are considered to be equally appropriate cluster centers, the first answer will be picked. These cases both involve the aggregator choosing between two equally "good" options, and thus the order of answers should not affect the quality of the aggregation significantly.

The aggregator contains two important, adjustable parameters: the maximum number of bins the aggregator will create, and the similarity parameter discussed above. The maximum number of bins is set by default to seven. The default value of seven was chosen because it is considered to be the number of elements in a list an adult can remember at once; it is large enough to show off the most interesting clusters in a class of about thirty, but small enough such that the histogram of bins fits neatly on the screen and does not take much time for the instructor to read. The temptation exists to set the maximum number of bins to a very high number or do away with it entirely so the instructor can see all of the possible clusters, but there are disadvantages in either case. The first disadvantage is that reading clustering output takes time away from lecture, and if there is too much output, the instructor may feel overwhelmed or waste time reading it all. Furthermore, one of the major purposes of this tool is for instructors to see problems which several students are having so he or she can talk about related misconceptions in class. In a large class, the instructor likely does not wish to discuss every single mistake, especially those only committed by one or two people. Thus, providing every possible cluster to the instructor would cause unnecessary cluster and provide unnecessary information.

The similarity parameter is set by default to 10%, meaning that answers must be 90% alike, according to the algorithm described above, to be placed together in the same bin. The 10% was chosen to mitigate the impact of mistakes originating from the handwriting analyzer; as the analyzer improves, this number would probably be set lower. Currently, the handwriting analyzer has an average success rate of 87%, and so 10% was chosen as a compromise between compensating for the analyzer’s mistakes and creating accurate bins. This parameter can also be set to 0 if the instructor wishes only for answers which are exactly alike to be clustered together.

5. TESTING

The aggregator is part of a system designed to help instructors in a classroom setting, and the aggregator's results are meant to be displayed to a classroom full of students. Thus, not only is it absolutely crucial that the aggregator cannot crash, which would disrupt lecture and waste the instructor's time, it is also crucial that the aggregator provides consistently reasonable results. Creating lectures with interactive questions in Classroom Learning Partner requires significantly more time than writing basic Powerpoint slides, overheads, or working on a blackboard. If the aggregator's results are not useful enough to justify the time commitment, then the instructor will no longer wish to use CLP. Furthermore, if the aggregator produces poor results which are displayed to the class, the students will likely lose interest in answering the questions to the best of their ability. The aggregator was tested in three ways: deployed in a classroom setting, outside the classroom using sample student answers from the 6.001 online tutor, and comparing the grouping of student answers from the tests with “human aggregator” groupings

5.1 Deployment

The full system with aggregator module was deployed in Dr. Kimberle Koile's 6.001 recitation at the end of the spring term in order to test the aggregation of handwritten answers in a classroom setting. Students were asked questions such as "Out of reading, writing, and listening, what is your favorite learning style?" Although the aggregator produced the expected results in most cases, deployment led to several important changes. The first version of the aggregator, for example, tended to choose the most unique answers as cluster centers. This clustering method, unfortunately, meant that if a student entered a joke answer, then that joke answer would almost certainly be chosen as a cluster center. In practice, several meaningless clusters of one answer each were produced. The problem was solved by favoring large clusters, rather than unique answers, while aggregating, which ensures that joke answers will likely remain in the miscellaneous bin. Deployment also led to a more efficient string comparison algorithm, as the original version caused the aggregator to run too slowly for real-time use (on the order of minutes) with answers to one of the questions. On the whole, the deployment was successful, and students generally enjoyed seeing histograms of what others in the class had answered. Whether to show the histograms to the student or only to the instructor is an open research question. Anecdotal evidence suggests that the histogram should be reserved for the instructor's use.[2]

5.2 Aggregator

While the system was deployed in the classroom, the majority of aggregator testing was performed outside the classroom, because the various CLP components were not all available at the same time. In addition, each CLP module needed to be tested on its own to ensure that it would not cause the system to crash and would behave as expected. In testing the aggregator, we used sets of student answers collected from the 6.001 online tutor. The online tutor asks exactly the same sorts of questions that the instructor would ask in 6.001 recitation and catalogs both student successes and mistakes, and so it can be considered a reasonable simulation of classroom use.

The first tests of the aggregator used small, artificially-created data sets to ensure that the aggregator was working properly and did not have bugs that would cause it to crash. An example of this kind of data set is discussed in the implementation section of this paper; the example with three unique numbers repeated several times each was a test case designed to check the basic functionality with the system. Small batches of string answers were obtained from the handwriting analyzer testing, in which about twenty participants were instructed to answer simple 6.001 questions [Rbeiz 2006]. This test evaluated

[2] Instructors who have used MIT's TEAL (Teaching Enabled Active Learning) classroom report that some students feel uncomfortable when they are one of a small group of students with an incorrect answer.

basic functionality, this time including a complicated string comparison algorithm. Tests were run both with instructor answers and without.

The second batch of testing used the student answers from the 6.001 online tutor. There were many questions in the system, from which four canonical test cases were chosen. Many questions required long pieces of code as answers, which the current version of the aggregator is not designed to handle. See Future Work below for more on aggregating student code. Some questions with shorter answers were so easy that most students submitted the correct answer on the first try, and there were not enough unique answers to produce interesting answer bin behavior. If the number of unique answers present in an answer set is less than the maximum number of bins the aggregator will generate, then every unique answer can have its own bin and the results are trivial. The questions chosen for the test represent typical questions which a 6.001 instructor would ask in class. For each question, two hundred student answers were chosen from the tutor's files.[3] A sample of these student answers can be found in the Appendix.

Question 1

Lec.2.4.4: `Write an expression that is equivalent to the following, but that uses lambda explicitly: '(define (fizz x y z) (+ x z))'.`

Correct answer: `(define fizz (lambda (x y z) (+ x z)))`

[Singer, 2006]

[3] Students can check their answers before submitting final answers to the tutor. Our example answers were chosen from the file of checked answers because many interesting incorrect answers were present.

Possibly due to the length of the student answers, the logic in the clusters chosen by the aggregator are nonobvious. Many answers ended up in the miscellaneous bin. The largest cluster is the one centered around answers like (define (fizz x y z) (lambda (x z) (+ x z))), an error which is likely caused by people mistakenly including the argument names both where they would be in the correct answer and where they were in the original statement. There are also interesting clusters based around answers missing "define fizz", so an instructor might glean that people did not understand that the question requires the procedure to be linked to the name "fizz".

Question 2

Lec.3.2.2.p1: Assume that we have evaluated the following definition: ‘(define fizz (lambda (a b) (* (/ a b) (/ b a))))’. Now, we would like to evaluate the expression ‘(fizz (+ 1 -1) 1)’. Indicate the first step of the substitution model.

Correct answer: (fizz 0 1)

This answer set results in an even larger miscellaneous bin, but the logic behind the other bins is much clearer. The largest bin results from the answers that included "define" and "lambda", mistakenly thinking that the first step involved evaluating the statement that defined the procedure. Another bin consists of answers that include "a" and "b" from students who did not realize they were supposed to substitute the argument's values in for the argument's names. The question asks for the first step of the evaluation, so unsurprisingly there is a bin that centers on the second step in the evaluation and one that centers on the third step. These mistakes might prompt the instructor to remind the students of the step(s) they forgot.

Question 3

Lec.3.2.2.p2: This is a continuation of the previous question, where the students are asked to provide the second step of the evaluation model.

Correct answer: `(* (/ 0 1) (/ 1 0))`

Since this question is a continuation of the previous question, it isn't surprising that some of the bins from the last question also appear here. In particular, we have a bin resulting from students forgetting to substitute in values for argument names and one from students who tried to evaluate the statement that defined the procedure. There is also a bin centered on the third step of the evaluation, and the rest of the bins, apart from the miscellaneous bin, appear to be from students who performed the math in a different order than the Scheme interpreter.

Question 4

Lec.3.2.3.p1: `Assume that we have evaluated the following definition: ‘(define fuzz (lambda (a b) (if (= b 0) a (/ 1 b))))’. Now, we would like to evaluate the expression “(fuzz -1 0)”. Indicate the first step of the substitution model.`

Correct answer: `(if (= 0 0) -1 (/ 1 0))`

This question is similar to Lec.3.2.2.p1, and the aggregator's bin creation is fairly consistent. Again we see the bin from students evaluating the statement defining the procedure and a bin from people who did not fully substitute values in for argument names. Another bin results from those who did not include the consequences of the "if" statement, perhaps feeling that since the test is evaluated before the consequents, only the test should appear in the first step of the evaluation. Another bin is produced from answers that mistakenly substituted the value of "a" in for the argument "b" in one position.

5.3 Human Aggregators

The tests using student answers from the online tutor served to ensure that the aggregator would not crash, and that the results seemed reasonable. When categorizing student answers, however, there is no true "gold standard" for what is reasonable. Even if humans categorize the data, unless there is some truly obvious grouping (i.e. if there are only three unique answers, creating three answer groups is trivial), they will probably produce different clusters. Of course, this does not mean that any grouping the computer produces is acceptable, and so some attempt at a standard should be made. To test the reasonableness of the system's groupings, CLP's aggregator results were compared with "human aggregator" results.

To compare machine and human aggregator results, student answers from the online tutor were printed on paper and given to 6.001 instructors and teaching assistants so they could create groups. These groups were compared to the aggregator's results on the same data, to provide some basis for evaluating the aggregation.

Smaller versions of the data sets discussed above were created for the human aggregators, in order to save time. Two hundred answers would take a prohibitively long time for a human to categorize, and so the intent was to create sets of student answers that reflected the original data but were comparable in size to a typical classroom—approximately twenty-five answers per set. To create these smaller data sets, the original large data sets were aggregated, and answers were drawn from each bin in proportion to the relative sizes of the bins (with the exception that a less than proportionate number of answers was drawn from the miscellaneous bin). The answers were printed and cut into individual slips of paper. The answers used for each test question are shown in the Appendix.

We asked four volunteers to cluster the answers by any method they wished. Each volunteer had different knowledge and/or teaching experience with 6.001, so that even though the sample size was small, we could have a sense of how experience affects aggregation. The volunteers were: a graduate student teaching assistant who had taught 6.001 tutorials for one term; a graduate student instructor who had taken the course as an undergraduate and taught 6.001 recitation for one term; a faculty member experienced with teaching the course several years ago; and a faculty member very experienced with creating and teaching the course.

When the people arrived for testing, they were handed the slips of paper corresponding to the first question. They also were given the question and the canonical correct answer to read. They then were asked to put the student answers in categories of their own design. Once they were finished, they were asked to explain their categorization scheme. Then, if they had created more than seven bins, they were asked to categorize the student answers again in only seven bins, and explain the new categorization scheme. This process was repeated for each test question.

In conducting the tests, it was notable that different people tended to cite very different logic when describing their categories. Two subjects talked mainly about placing answers in categories based on the mistakes they believed the students had made, while another hardly mentioned possible mistakes and instead sorted the answers on the basis of key words and answer length. Another subject, noting that the sample student answers were Scheme code fragments, chose to focus on what results the fragments would produce in a Scheme interpreter. This served to confirm that teaching experience affected aggregation, as the more experienced faculty members were focused on student mistakes, while the graduate student who had spent the least time teaching 6.001 was the one who focused primarily on key words. There was a constant in the human testing, however—all of them took far longer than the aggregator to produce their clusters. In fact, they commonly took ten minutes per set of student answers. One even commented aggregator in this situation, as the process was much too time-consuming and tedious for a teaching assistant to do in class.

Question 1

Lec.2.4.4: `Write an expression that is equivalent to the following, but that uses lambda explicitly: ‘(define (fizz x y z) (+ x z))’.`

Correct answer: `(define fizz (lambda (x y z) (+ x z)))`

The most noticeable common thread among the subjects' groupings for this question is that they all made at least initial groupings based on keywords. All of them split answers beginning with "define" from answers beginning with "lambda," while the aggregator was not so strict in making this distinction. Apart from this split, the bins created by the subjects are quite different from one another and from the aggregator's bins. It is possible that there simply is not an intuitive grouping for this set of student answers.

Question 2

Lec.3.2.2.p1: `Assume that we have evaluated the following definition: ‘(define fizz (lambda (a b) (* (/ a b) (/ b a))))’. Now, we would like to evaluate the expression ‘(fizz (+ 1 -1) 1)’. Indicate the first step of the substitution model.`

Correct answer: `(fizz 0 1)`

For this set of student answers, the human testers produced several of the same bins the aggregator produced. The aggregator generated a bin of answers containing "define" and "lambda," and the human aggregators consistently created separate bins for answers containing "define" and answers containing "lambda," citing different keywords or the idea that misunderstanding how "define" works is a more serious mistake than misunderstanding the "lambda" construction. All of the human subjects and the aggregator created a bin of answers in which the student forgot to substitute values for variables. They also all created a bin of “almost correct” answers that closely resemble the correct answer but for a typo or parenthesis error. The aggregator created two bins containing answers in which students had skipped a step or more; the humans, for the most part, created one bin for all of these answers. Overall, the subjects generated very similar clusters to those created by the aggregator, although they differed a bit on which clusters to “merge” and “split.”

Question 3

Lec.3.2.2.p2: This is a continuation of the previous question, where the students are asked to provide the second step of the evaluation model.

Correct answer: `(* (/ 0 1) (/ 1 0))`

For this question, the only keyword present was “lambda,” and both human and machine aggregators predictably made a bin for the student answers containing “lambda.” They also both created bins of student answers in which the student had not substituted values for variables, as in the previous question. This step left the answers that contained only numbers and operators, and here the aggregator and subjects diverged somewhat. One subject based his bins on the length of the expression and what operators it contained, which produced results similar to the aggregator’s, but another split the expressions into bins based on what their evaluations in Scheme would be, which produced a larger difference.

Question 4

Lec.3.2.3.p1: `Assume that we have evaluated the following definition: ‘(define fuzz (lambda (a b) (if (= b 0) a (/ 1 b))))’. Now, we would like to evaluate the expression “(fuzz -1 0)”. Indicate the first step of the substitution model.`

Correct answer: `(if (= 0 0) -1 (/ 1 0))`

For this question, the subjects and the aggregator were mostly in agreement. Common bins included a bin for those who erroneously used “define” and “lambda” in their answer, a bin for those who forgot to substitute values for variables, and a bin for those who only included the test of the if statement and left out the consequents. These bins left answers which were mostly correct, and humans tended to divide these into typo / missing parenthesis and erroneous consequent, which is also similar to the aggregator’s division.

6. DISCUSSION AND FUTURE WORK

6.1 Semantics

Although the aggregator was originally designed with 6.001 in mind, one of the goals was to make it as universal as possible. Thus, there are only two references to 6.001-specific concepts in the code base: the procedure that expands abbreviated words into their full forms, which has a lookup table of common

6.001 abbreviations, and the Scheme expression option for string handling, which does not strip away parentheses like the normal string processor. This generality carries a price, however, as the aggregator makes very little use of semantics.

In theory, there are many ways in which the aggregator could use semantics. For example, if the instructor asks "What Scheme procedure can be used for (a task)?, the aggregator could compare answers not only to instructor answers, but also to a list of common Scheme procedures. This functionality would help remedy the mistakes made by the handwriting analyzer, as a word that is just one or two characters apart from a known Scheme procedure would be matched to its closest possible answer instead of appearing as a confusing mistake. The aggregator also could adjust penalties based on an answer's semantics—leaving out the procedure in a Scheme expression could carry a larger penalty than leaving out an argument, for instance.

Knowledge of semantics would help the aggregator in many other contexts as well. If the instructor asked a math problem, it would be ideal if the aggregator could parse the math problem and auto-generate a list of common incorrect answers, stemming from likely mistakes such as performing the operations out of order or dropping a negative sign. If the aggregator had a built-in dictionary, handwriting analyzer mistakes would be less of a problem, because if a handwriting analyzer mistake resulted in a non-dictionary word, the aggregator could provide a likely alternative. Of course, this functionality might pose a problem if the instructor asked a question that did not have an answer that appears in a standard dictionary. In this case, specialty dictionaries would have to be loaded depending on the class, such as the list of Scheme procedures mentioned above.

Although these semantic additions probably would improve the aggregator's results, the improvements might not outweigh the disadvantages. One problem is that producing a semantics module that helps with a particular class would require a great deal of knowledge associated with class, so the programmer would have to work with an instructor or someone else highly knowledgeable in the field to create the module. Another issue is efficiency. The aggregator is already pushing the limit of how long an instructor should have to wait for an answer, and adding a feature like dictionary lookup will certainly increase the time requirement. There also exists the possibility that in some cases, a semantics module would hurt more than help. If the instructor wanted to ask a question with an answer that the aggregator was not capable of parsing with its particular semantics module, for example, the aggregator might attempt to fit the answer to what it was expecting, which could lead to errors. Despite these issues, investigating the use of semantics modules for specific classes would be a good direction of attempted improvement for future research.

6.2 Additional Answer Types

As stated previously, one of the major considerations in designing the aggregator was the ability to easily add more features—in particular, new types of answer. To add a new answer type, one must be able to store and extract data of that type from the database, process the data so that it can be handled by the Lisp interpreter, and, most significantly, write an algorithm that can compare two answers of this type and return a percentage reflecting how alike the two answers are. The comparison algorithm moreover must be capable of making O(n^2) comparisons, where n is the number of student answers received, in less than thirty seconds, or the aggregator will take too long to be of practical use in the classroom. In the case of complicated answer types, such as full essays, designing an algorithm that meets these requirements is likely a full thesis' worth of work in itself. What follows are some design thoughts on how several proposed data types—diagrams, code, and longer text passages—might be incorporated into the aggregator.

Diagrams

The next answer type that will be added to the aggregator is the diagram. The aggregation of sketched diagrams has been one of CLP’s goals from the beginning, and the main reason why the project has focused on tablet PCs instead of more traditional laptops. One of the major stumbling blocks to diagram aggregation has been designing an ink interpreter that can detect shapes and spatial relations as well as characters. The current focus, therefore, is on a series of formulaic diagrams with a limited set of possible shapes: trees and box-and-pointer diagrams. See Figures 6 and 7 for examples.

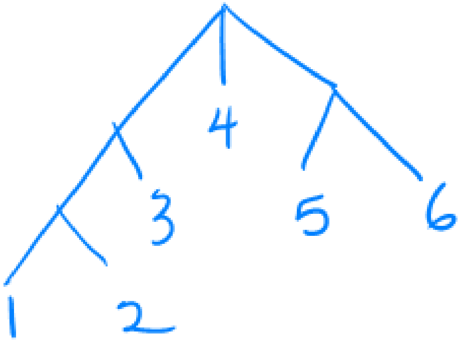


Figure 6: An example tree

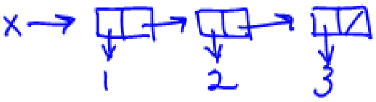


Figure 7: An example box-and-pointer diagram

All of these diagrams can be expressed in an XML-type language. For example, the tree above can be described as:

```
<node> <node> <node> 1, 2 </node>, 3 </node>, 4,
<node> 5, 6 </node> </node>
```

Likewise, the box-and-pointer diagram can be written as:

```
 <boxpointer> x, <cons> 1, <cons> 2, <cons> 3, null
</cons> </cons> </cons> </boxpointer>
```

When expressed in these terms, the answers are very similar to sequences, as they are simply a collection of elements which are supposed to occur in a particular order. Thus, the comparison algorithm for these diagrams will be very much like the comparison algorithm for sequences. Two important differences between these diagram sequences and sequences of strings are that 1) the diagram sequences contain XML tags, which must be treated as symbols, not strings, and 2) the diagram sequences must allow nested answers. In other words, one piece of the diagram (a square, for example) can potentially contain another piece of diagram (a circle, perhaps), or a string, or a sequence of strings. One of the major challenges with writing diagrams in XML terms is ensuring that the order is consistent from diagram to diagram. Ideally, two diagrams that are semantically the same should write the same XML representation, but it is possible that the relative placement of objects on the slide might cause the diagram analyzer to write two different representations. Two sister leaves on a tree, for example, might correctly be written in either order. Ideally, the aggregator will have reasoning more advanced than simple pattern-matching, and will be able to determine whether two diagrams are semantically identical.

Graphs

All of the diagrams discussed thus far are well-structured. Each has only a small number of possible pieces; for example, a box-and-pointer diagram contains only boxes, boxes with slashes through them, and arrows. The dimensions of the boxes and lengths of the arrows do not matter. Thus, it is possible to represent these objects in XML without having to encode information about their sizes. While this simplifies matters, it also unfortunately disallows a wide variety of interesting diagrams for which the dimensions are important, such as graphs. The representation of a graph would have to include at least the endpoints of every line, and curves would either require a vector of points or an approximated equation representing the curve. The aggregator would need geometric comparison algorithms to compare two lines or two curves in order to determine how alike two graphs are. Furthermore, the instructor might need to provide additional information rather than simply drawing an example of a correct graph, for in some cases the instructor might wish for the students to draw a line with a particular slope, to demonstrate a trend, whereas in other cases the instructor might care more about the line including particular data points. For many types of diagrams, it might be difficult for the instructor to articulate exactly what parts of the diagram must be present for a student answer to be correct. If the instructor asks students to draw a graph of a particular trend, he or she might want a detailed graph with carefully labeled axes and known data points marked, or he or she might accept any graph with an upward slope as correct. The aggregator may need to employ different algorithms depending on the accuracy the instructor requests. Another closely related answer type is to request that students circle a particular piece of a diagram. When people are asked to circle something, they will often draw a sloppy circle which cuts off the edges of the thing they are trying to circle, or one that includes bits of other pieces of the diagram. The aggregator will have to know how much of the correct answer the student is allowed to exclude, and how much extraneous material the student is allowed to include, for the answer to still be considered correct.

Code

As the aggregator was originally designed for 6.001, which teaches the principles of programming, accurate aggregation of code fragments is important. Currently, code fragments are aggregated as strings or sequences, but this solution is far from optimal. In fact, this approach will only produce reasonable results if all variable and procedure names are provided by the instructor, and if there are only a small number of possible ways to write the code. These restrictions exclude all but the most basic programming questions. The 6.001 online tutor checks student-written procedures by running instructor-created test cases, which, while also imperfect, is a far more robust system than treating the code fragments as simple strings. If this functionality were incorporated into the aggregator, it would be relatively simple to place code fragments into answer bins based on which tests they pass and the results of any failed tests. Two code fragments that pass all tests but one and produce the same answer for the failed test are likely to exhibit a similar coding mistake. The aggregator also would need to catch errors thrown by the student code and take them into consideration when creating the answer bins.

While the ability to run student code would be a great asset to the aggregator, adding this feature would be far from trivial. In order to run student-written Scheme procedures, the aggregator would need to be able to quickly load and call a Scheme interpreter. Moreover, the aggregator would be required to run several tests on each code fragment, so time requirements might become an issue, particularly if the class is large or the requested code is complex. To protect the aggregator from infinite loops and very inefficient code, each test would have to time out after a very short time interval; this restriction means that the aggregator could not differentiate between code containing infinite loops and code that is functionally correct that takes a long time to return. Ideally, the instructor could define his or her own timeout for each question, as he or she might be willing to wait longer for aggregation results from a particularly complex programming question.

Another issue with this approach to aggregating code is that it does not allow instructors to ask questions about programming style. In 6.001, the instructor often asks for the students to write a certain procedure using recursion as opposed to iteration (and vice versa), but simply running the code on test cases is not enough to differentiate a correctly-written recursive procedure from a correctly-written iterative procedure. Furthermore, many questions in 6.001 require a student to use a certain function, or request that a procedure be written without a certain function. If the instructor asks the students to code their own version of the procedure "map" from scratch, for example, then they obviously should not be allowed to use the predefined procedure "map" in their code.

The above issues can be partially resolved through the use of keywords. An instructor, when defining the question, could specify which keywords and phrases the student code should include, and which keywords and phrases it should not include. A recursive procedure should contain a particular signature code fragment which differentiates it from the iterative version of the procedure, and thus instructors would have increased ability to request a particular coding style from the students. If this approach to Scheme questions works well, it might be beneficial to expand this answer type to incorporate many types of "runnable" objects, such as code in other languages, or mathematical expressions, which can be tested by substituting in values for variables and observing the result.

Text

Finally, one answer type of particular interest is freely written text. The aggregator can currently handle text, but only can compare the text character by character with an instructor-defined answer. The instructor therefore is limited to asking questions with predefined answers that do not vary from person to person; he or she cannot ask the student to explain a concept or define a term, because there will be a multitude of answers that differ only in superficial wording. In the current system, if the instructor asked, "Why do objects fall?", the answers "gravity," "because of gravity," and "due to gravity" would all be considered different. One solution to this problem is to use keywords, as has been proposed for programming questions. In the above case, the keyword would obviously be “gravity”. Keywords alone will not be enough, however. For code fragments, the test cases prevent students from writing incorrect code containing the keywords, but if keywords alone were used for text fragments, then the aggregator could not properly distinguish between the phrases "because of gravity," "it has nothing to do with gravity," and "I don't know what gravity is." More sophisticated solutions to this problem include syntax and semantic parsers. These parsers traditionally require large amounts of time to produce a correct answer, however, and it might be years before tablet PCs are available to the classroom that can run sophisticated algorithms on each student answer and return in less than thirty seconds. When one considers that such parsers are frequently inaccurate, it might not be worth the added time burden to add such technology to the aggregator, at least until hardware has improved.

6.3 Concluding Thoughts

Without the use of technology, it is difficult or impossible for an instructor to poll his or her students on non-multiple-choice questions, as it simply takes too long to ask a class full of students to answer a question and then process all the results to determine which misconceptions are the most severe and widespread. Classroom Learning Partner, which contains an aggregation component, is an effective solution to this problem. The aggregator module can take a class' worth of student answers and return in less than a minute with easy-to-read results showing the most common correct and incorrect student answers given. In all tests run, the aggregator produced bins which, for the most part, easily could be explained in terms of the mistakes the students had made. Even if the aggregator's bins don't match the instructor's ideal clustering—an impossible task, since among instructors there is disagreement as to what an ideal clustering would be—the aggregator is still a useful tool for personalizing the instructor's teaching for a particular class. The successful results, furthermore, were accomplished almost entirely without analyzing the semantics of the answers, which suggests an applicability across domains. The aggregator is designed to be easy to improve in the future, with additional supported answer types, semantics parsing, and more efficient algorithms, but as it stands it forms a solid foundation for a useful and unique classroom tool.

7. REFERENCES

Abelson, H. and Sussman, G.J. *Structure and Interpretation of Computer Programs, Second Edition.* MIT Press, Cambridge, 1996.

Anderson, R., Anderson, R., Simon, B., Wolfman, S., VanDeGrift, T., and Yasuhara, K. Experiences with a tablet-pc-based lecture presentation system in computer science courses. In *Proc. Of SIGCSE '04.*

Anderson, R., Anderson, R., McDowell, L., and Simon, B. Use of Classroom Presenter in engineering courses. In *Proc of ASEE/IEEE Frontiers in Education Conference, 2005.*

Chen, Jessica. Instructor Authoring Tool: A Step Toward Promoting Dynamic Lecture-Style Classrooms. M.Eng. Thesis, MIT Department of Electrical Engineering and Computer Science, February 2005.

Chevalier, Kevin. An Aggregation User Interface for the Classroom Learning Partner. Undergraduate Advanced Project report, MIT Department of Electrical Engineering and Computer Science. December, 2005.

Draper, S.W. From active learning to interactive teaching: Individual activity and interpersonal interaction. In *Proc. of Teaching and Learning Symposium: Teaching Innovations, 2004,* The Hong Kong University of Science and Technology.

Koile, K. and Shrobe, H.E., 2005a, The Classroom Learning Partner: Promoting Meaningful Instructor-Student Interactions in Large Classes, MIT CSAIL TR., In preparation.

Koile, Kimberle and Singer, David. Improving Learning in CS1 via Tablet-PC-based In-class Assessment. In *Proc. of ICER 2006(Second International Computing Education Workshop).*

Rbeiz, Michel. Semantic Representation of Digital Ink in the Classroom Learning Partner. M.Eng. Thesis, MIT Department of Electrical Engineering and Computer Science, 2005.

Singer, David. Personal correspondence, 2006.

8. APPENDIX

This appendix contains a sample of student answers, taken from the 6.001 online tutor, that were used for testing the aggregator. They are also the selected student answers given to the "human aggregators."

Lec.2.4.4

```
(lambda (xyz) (+xz))
(lambda (x y z) (- (+ x z y) y))
(lambda (fizz x y z) (lambda (x y z) (+ x z)))
define (lambda (x y z) (+ x z)) (+x z) )
(define x (lamda (x y z) (+ x z)))
((define (fizz x y z) (lambda (x z) (+ x z)))
(define \"fizz x y z\" (lambda (x y z) (+ x z)))
(define (fizz x y z) lambda (x z) (+ x z))
(define (fizz x y z) (lambda x z (+ x z)))
(define (fizz x y z) (lambda () (+ x z)))
((lambda (fizz) (+ x z)) x y z)
((lambda (x y z) (+ x z)) x y z)
(lambda (x y z) (+ x z) (fizz x y z))
((lambda (x z) (+ x z)) (fizz x y z))
(lambda (x y z) + x z)
(lambda (x y z) (+ x z) )
(define (fizz) (lambda (x z) (+ x z)))
(lambda (x y z) (+ x z))
(define fizz (lambda (x y z) (+xz)) )
(lambda (x y z) (+ x z))
lambda (x y z) + x z
(lambda () (+ x z))
(lambda ( ) (+ x z) )
(lambda (fizz x y z) + x z)
(lambda fizz (x y z) (+ x z))
```

Lec.3.2.2.p1

```
(* (/ a b) (/ b a))))
(* (/ a b) (/ b a))
fizz 0 -1)
fizz ((* (/ 0) 1) (/ 1 (+ 1 -1)))
(* (/(+ 1 (- 0 1)) 1) (/ 1 (+ 1 (- 0 -1))))
(* (/ (+ 1 (- 0 1)) 1) (/ 1 (+ 1 (- 0 1))))
(fizz (* (/ (+ 1 -1) 1) (/ 1 (+ -1 1))))
(lambda ((+ 1 -1) 1) (* (/ ((+ 1 -1) 1)) (/ ((+ 1 -1) 1))
(+ 1 -1) 1 ))
(procedure (a b) (* (/ a b) (/ b a))) (+ 1 -1) 1
(lambda (a b) (* (/ a b) (/ b a)) (+ 1 -1) 1)
fizz
(* (/ (+ 1 -1) (1)) (/ 1 (+ 1 -1)))
((* (/ (+ 1 -1) (/ 1 (+ 1 -1)))))
(* (/ (+ 1 -1) 1) (/ 1 (+ 1 -1) ) )
(* (/ (+ 1 -1) 1)(/ 1 (+ 1 -1)))
(* (/ (+ 1 -1) 1) (/ b (+ 1 -1))))
(* (/ (+ 1 -1) 1) (/ 1 (+ 1 -1)))
(define fizz (lambda ((+ 1 -1) 1) (* (/ (+ 1 -1) 1) (/ 1 (+
1 -1)))
(define fizz (lambda (+ 1 -1) 1) (*(/ (+ 1 -1) 1) (/(-1 1)
1)))
(fizz (lambda ((+ 1 -1) 1) (* (/ (+ 1 -1) 1) (/ 1 (+ 1 -
1)))))
(fizz (+ 1 -1) 1)
(fizz (+ 1 -1) 1)
(fizz (0 1)
(fizz 0 1)
```

Lec.3.2.2.p2

```
(* (/ 0 1) (/ 1 0)" "(* (/ 0 1) (/1 0)
(* (/ 0 1) (1 0))
(* (/ 0 1) (/ 1 0)
(* (/ (+1 -1) 1) (/1 (+1 -1))))
(/ 1 0)
(fizz (/ 0 1) (/ 1 0))
(/ 0 -1)
fizz ( 0 0)
(* (/ 0 1) (/1 0)) error
(lambda (0 1) (* (/ a b) (/ b a)))
fizz
(/ 1 (+ 1 -1))
0
(/ -1 +1)
(* (/ 0 -1) (/ -1 0))
1
(lambda (a b) (* (/ a b) (/ b a))) 0 1
(lambda (a b) (* (/ a b) (/ b a))) 0 1
 (* (/0 1) (/1 0))))
(* (/0 1) (/1 0))
(lambda (0 1) (* (/ 0 1) (/ 1 0))
(lambda (0 1) (* (/ 0 1) (/ 0 1)))
(0 1) (* (/ 0 1) (/ 1 0))
(* (/ (+ 1 -1)) (/ 1 (+ 1 -1)))
(* (/ (+ 1 -1) 1) (/ 1 (+ 1 -1)))
(* (0) (/ 0 1))
(* 0 (/ 0 1))
```

Lec.3.2.3.p1

```
(define fuzz (lambda -1 0) (if (= 0 0) -1 (/ 1 0)))
(lambda (-1 0) (if (=0 0) -1 (/ 1 b)))
if(= 0 0)
(if (= b 0) -1 (1/b))
(if(= 0 0) 1 (/ 1 0))
(if (= 0 0) 0 (/ 1 0))
(lambda (-1 0) (if (= b 0) a (/ 1 b)))
(fuzz (a b))
a
(if (= 0 0) -1 0)
(= a -1)
-1
(if #t a (/ 1 b))
(if (= 0 -1) -1 (/ 1 0))
((Iambda (a b) (if (=b 0) a (/ b 0))) -1 0)
(fuzz -1 0)
(if (= 0 0) -1)
(if (= 0 0) 0)
(if (= 0 0))
(if (=0 0) -1 (/ 1 b))
(if (= 0 0) -1 (/ 0 b))
(if (= 0 0) -1 (/ a 0))
(if (= 0 0) -1 (/1 0))
(if (= 0 0) 1 (/ 1 0))
(if (= 0 0) -1 (/ 10))
(if (= 0 0) -1 (/1 0))
(fuzz (if (= 0 0) -1 (/ 1 0)))
(if (= 0 0) a (/ 1 b))
(if (= 0 0) a (/ 1 0))
```