Interpretation of Box-and-Pointer Diagrams in Classroom Learning Partner

By

Kevin Chevalier

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

May 25th, 2007

©2007 Massachusetts Institute of Technology
All rights reserved.

Author ____
Department of Electrical Engineering and Computer Science
May 25, 2007

Certified by ____
Kimberle Koile, Ph.D.
Research Scientist, MIT CSAIL
Thesis Supervisor

Accepted by ____
Arthur C. Smith, Ph.D.
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

Interpretation of Box-and-Pointer Diagrams in Classroom Learning Partner

By

Kevin Chevalier

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

Abstract:

This thesis focuses on the design and implementation of a tool to interpret the meaning of box-and-pointer diagrams drawn in digital ink. The tool was designed to work with the Classroom Learning Partner presentation system. The interpreter was designed to use state of the art sketch recognition tools to recognize shapes, and state of the art text recognition tools to recognize text.

Thesis Supervisor: Kimberle Koile, Ph.D.
Title: Research Scientist

Acknowledgements

I would like to thank everyone who helped with this thesis. I would like to thank especially my advisor Dr. Kimberle Koile, who was always understanding and helpful, even when extremely busy. Also, Tracy Hammond and Aaron Adler, who helped with the integration of the LADDER shape recognizer into the interpreter. Many thanks go to the other members of my group who helped with this work, including Kah Seng Tay, Sanjukta Pal, and Adam Rogal. I would also like to acknowledge Michel Rbiez, who gave me ideas and advice on surviving the MEng. Also, thanks to Dan, Denise, Mom and Dad for support and understanding.

1. Introduction 5
2. Classroom Learning Partner Overview 7
2.1 Classroom Use 9
2.2 System Architecture 11
2.3 Interpretation 15
2.4 Aggregation 16
3. Cons and the Box-and-Pointer Diagram 17
4. Shape Recognition 22
4.1 Text Recognizer 23
4.2 Shape Recognizer 24
4.3 Clean-up Stage 30
4.4 Semantic Representation 36
4.4.1 XML Format 36
4.4.2 Semantic Representation Builder 38
4.4.3 Semantic Representation Reader 38
5. Results 41
6. Discussion and Future Work 45
References 48

1. Introduction

Classroom Learning Partner (CLP) is a presentation system designed to increase interaction between students and instructors during class. It uses Tablet PCs to allow students to write answers with digital ink and send completed exercises to the instructor during a lecture. The ink submissions from the students are automatically interpreted and classified together with other answers that are similar, and the instructor can view summary results which give an overview of students' understanding and misunderstanding. Given this information, the instructor can modify a lesson during class, skipping material if she sees that students understand, and going into more detail if the do not. The system has previously been used successfully for exercises with answers that were alphanumeric strings [Koile et al 2007].

The goal of this thesis is to extend CLP so that students also can provide sketched answers. Adding such answers will increase the types of exercises that an instructor can use in a class, thereby giving students opportunities to learn material not well presented as text. To investigate the plausibility of adding sketched answers to CLP, the prototype built for this thesis focuses on the interpretation of a commonly used diagram introductory computer science---what is called a box and pointer diagram in MIT's introductory computer science course. Such diagrams are commonly used when teaching the concepts of data structures and linked lists.

The interpreter was designed with three goals in mind. (1) We wanted a student to be able to sketch a diagram as easily as on paper and not be interrupted by being asked to check the validity of CLP's interpretation; we thought the interruption would be distracting to the student. (2) In keeping with the idea that students should be able to

easily sketch in CLP, we wanted the interpreter to work without asking users to train CLP with samples of their sketches; we wanted anyone to sit down and use CLP without a learning curve or delay, and without giving instructions that the diagrams be drawn in a specific way. (3) We wanted the interpreter to integrate easily with the existing CLP system, which means that it should operate offline, so that all interpretation is done after the entire diagram has been drawn. The architecture was chosen in this way because research has shown that there are no accuracy benefits to running an interpreter online, and an offline interpreter can be run on a different machine than the one being written on. [Rbeiz 2006]

This document provides an overview of CLP, followed by a description of box-and-pointer diagrams, and the potential issues faced when interpreting hand-drawn diagrams. Background information is then presented about sketch recognition and the blackboard problem-solving model used in the box-and-pointer interpreter. The prototype implementation of the box-and-pointer diagram interpreter for CLP is described and challenges discussed. The results of accuracy and performance testing on the interpreter then are presented, followed by a discussion of possible areas of improvement for the system that could be done in future work.

2. Classroom Learning Partner Overview

Classroom Learning Partner is a Tablet-PC-based wireless presentation system designed to support real-time feedback and assessment of in-class exercises. The system has been in development in the Computer Science and Artificial Intelligence Laboratory at the Massachusetts Institute of Technology from 2005-2007. The goal of the project is "to develop innovative technology to improve student experiences in large classes by increasing interaction between instructors and students." [Koile and Shrobe 2006, Koile and Singer 2006] The system is built on top of Classroom Presenter, a presentation system developed at the University of Washington that allow an instructor to display in-class exercises on her machine, a public display, and on students' Tablet PCs, Students can write with a digital pen on a Tablet PC screen, taking notes as if they were using a pen and paper. Students also can work out exercise solutions which then can be sent wirelessly and anonymously to the instructor, who can choose particular solutions to display on the public screen [Anderson 2003]. Figure 1 shows an example of a student's submission during class. Figure 2 shows the instructor's view of the exercise slide.

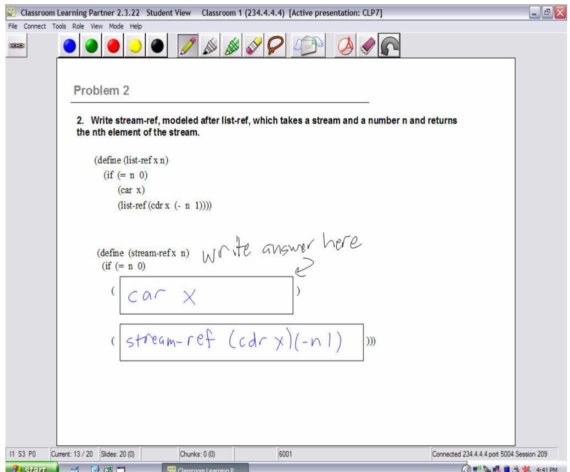


Figure 1: Sample student submission

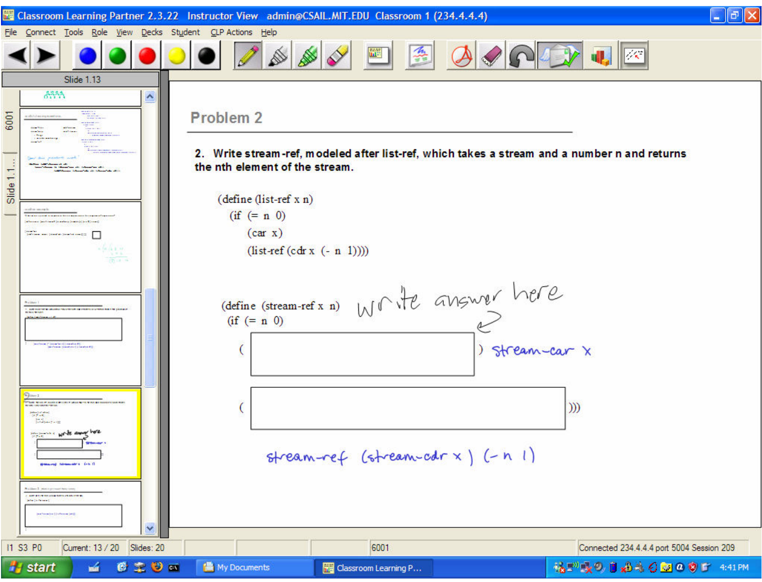


Figure 2: Instructor View; note that the instructor answers are not visible in the student view; the filmstrip to the left shows the presentation slides and can be made visible on the student machines (if the instructor allows the "full navigation" option)

Initial studies to judge the impact of Classroom Presenter style systems have shown the system to be beneficial to some students. In one course, 44 out of 45 students who used Classroom Presenter indicated that they felt CP had a positive effect on learning. [Anderson 2005] Another three studies conducted at MIT by Dr. Kimberle Koile in her 6.001 recitations have shown that students were much less likely to perform in the bottom 25% of the class on an examination after participating in classes in which Classroom Presenter or CLP and Tablet PCs were used by the instructor and each student [Koile and Singer 2006, Koile et al 2007 a, 2007b]. Another study is being conducted this term.

2.1 Classroom Use

The underlying Classroom Presenter system supports wireless transmission of ink from instructor to student (and vice versa). Even with the use of Classroom Presenter to increase student-instructor interaction, however, the number of student submissions that an instructor can view and provide feedback about during a class is small compared the number of student submissions. In large classes, it would be hard for an instructor to gain a complete understanding of the students' comprehension by looking through the large number of submissions, without slowing down the pace of teaching. The goal of CLP is to provide the instructor with summary information about student submissions for in-class exercises. We hypothesize that this information will allow the instructor to provide immediate feedback to the students, and to maximize the number of students who receive feedback. Figure 3 shows an example of a histogram produced after aggregation.

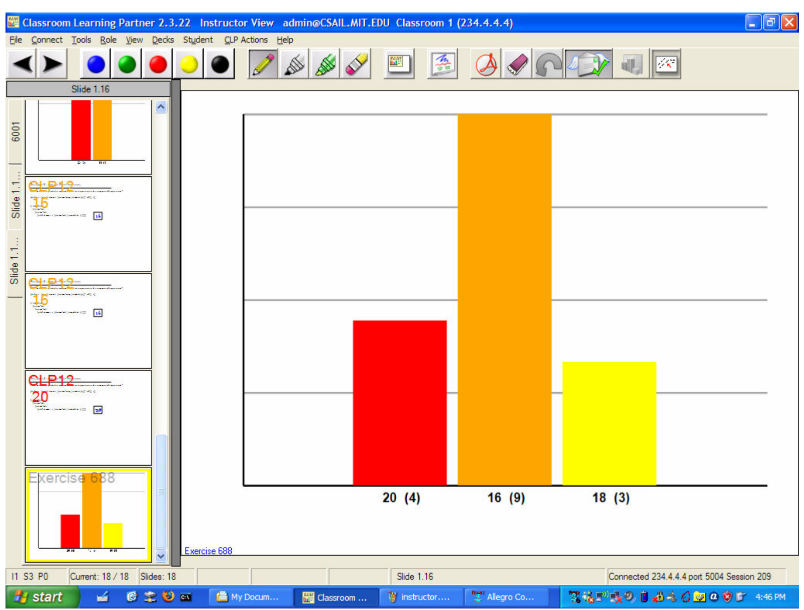
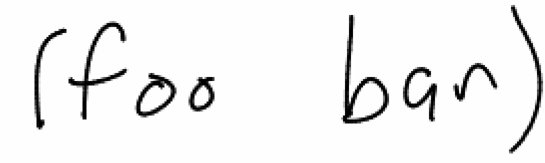


Figure 3: Results of aggregation as seen by instructor

This summary information consists of "bins" of student answers, with each bin corresponding to an equivalence class of student answers. The bins are generated by the

automatic interpretation and aggregation of student submissions. Interpretation is the process of converting a student's submission from hand drawn digital ink into a representation of the semantics of the answer. This representation is read by a program called an aggregator, which groups together answers with similar semantics. The results of the aggregation are displayed to the instructor in the form of a histogram, with each student submission given a color-coded label that indicates in which bin it has been placed. By viewing the histogram, the instructor can see how many students submitted right answers, how many submitted wrong answers, and then can dynamically change the content of the lesson based on the students' understanding and misunderstanding. Interpretation of answers consisting of alphanumeric characters and sequences of alphanumeric characters was started in 2006 [Rbiez 2006]. Research into the interpretation of these types of answers is continuing [Tay 2007]. Aggregation of these types of answers was started by [Smith 2006] and has been extended by [Pal 2007]. Figure 4 shows an example of an ink sequence and its corresponding semantic representation.



<Answer type = "sequence">(foo bar)</Answer>

Figure 4: Drawn sequence and interpreted semantic representation

2.2 System Architecture

The architecture of CLP consists of the components shown in Figure 5. The components run on either the instructor tablet or the student tablets, as indicated in Figure 5.

Before class:

1. Instructor creates presentation and exercises using IAT; exercises are stored in database, slides on file server.

During class:

2. Instructor retrieves presentation from database (or it is resident on her machine already).

3. In Classroom Presenter, presentation slides are broadcast to student machines; in CLP slides are automatically loaded onto student machines when they log in to their Tablet PCs.

4. When a slide with an exercise is displayed, student enters ink answer, which is interpreted on his or her machine.

5. Each student's ink answer and interpretation are transmitted to database via wireless peer-to-peer (P2P) network.

6. When instructor indicates end of student exercise period (e.g., orally), she clicks on aggregator icon, which causes aggregator to retrieve the interpreted ink answers, aggregate them, and produce summary data.

7. Summary data is transmitted to the database via wireless P2P network.

8. Summary data is displayed on instructor's machine. [Koile et al 2007]

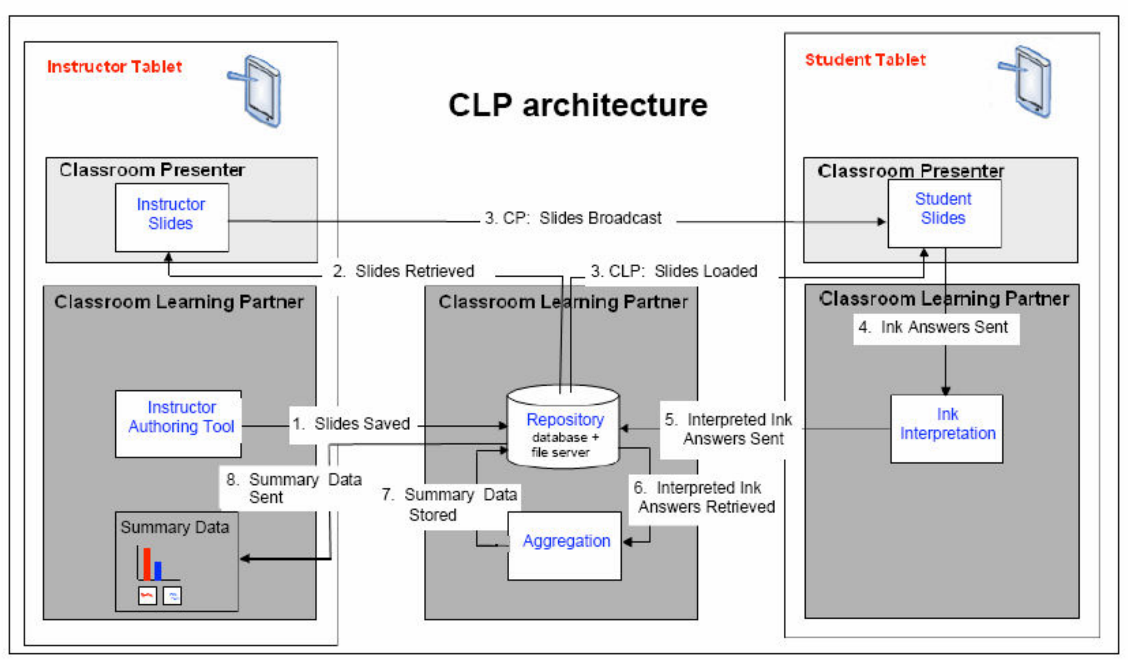


Figure 5: Architecture of Classroom Learning Partner.

Figure 6 shows the hardware and networking configuration that supports the use of CLP. The instructor, student, and projector Tablet PCs are connected to each other and to the central database computer via a wireless peer-to-peer network. The projector machine provides a means for displaying slides and student responses for in-class discussion. The database computer also serves as a gateway, giving the Tablet PCs access to the internet for automatic loading of instructor slides onto student machines at the beginning of each class. Student submissions are transmitted to the instructor machine, the projector machine, and to the database. In very large classes, the submissions will only be sent to the database; the summary results displayer will select a small number of representative submissions and transmit them to the instructor and projector machines.)

In a large lecture with over 100 student tablets submitting answers simultaneously, if all student tablets tried to send submission data to the instructor over the in-class wireless network, a bottleneck would be created and many of the submissions would be lost. In order to scale the system to large classes, the networking configuration will need to consist of several computers acting as wireless access points for the Tablet PCs, rather than just the database computer also acting as the only wireless access point. (Note: In our first implementation, the database was on an external computer reached via the internet, but repeated connectivity problems forced a move to a computer on the in-class wireless network.)

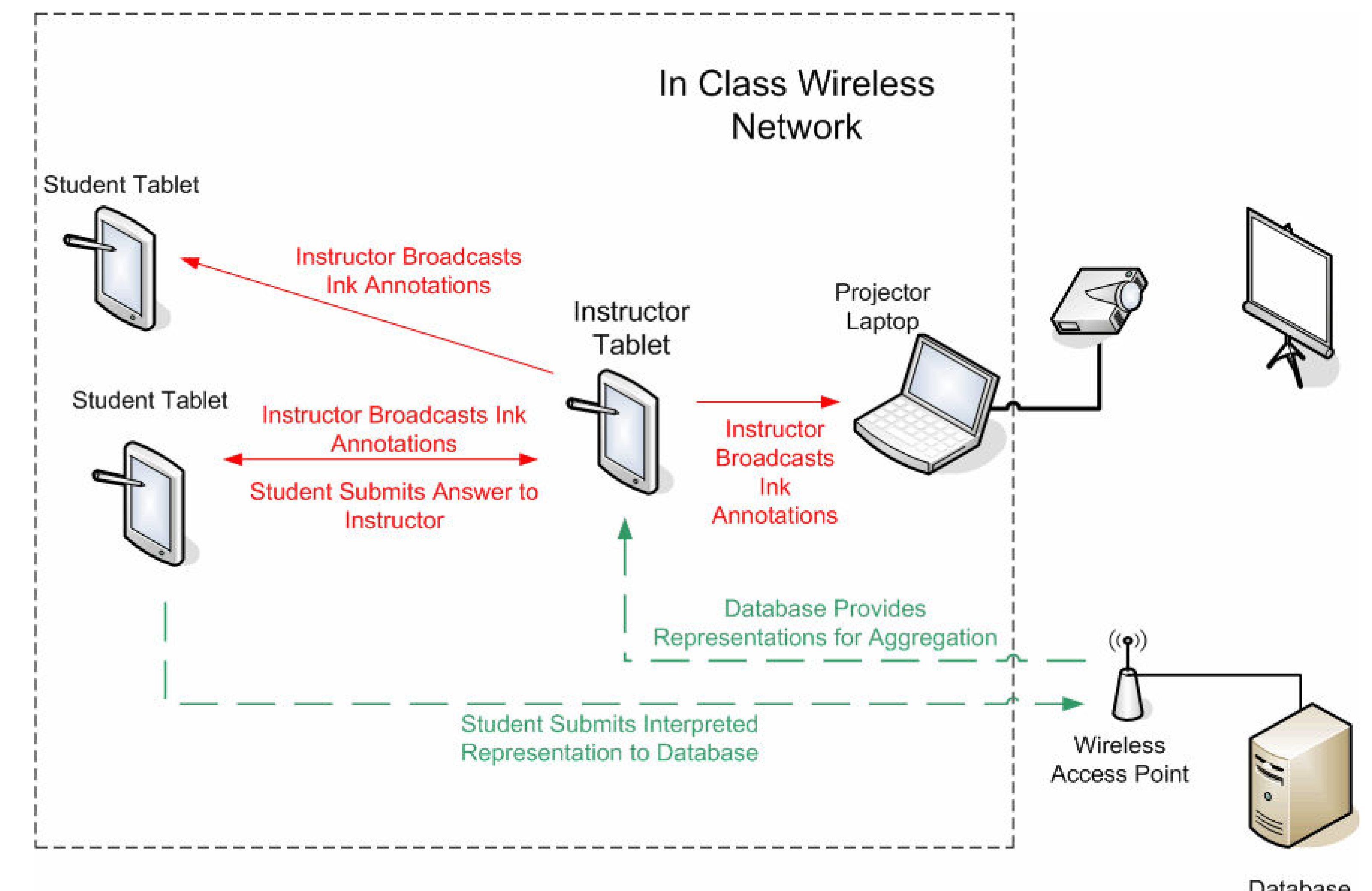


Figure 6: CLP networking configuration

2.3 Interpretation

When an answer to an in-class exercise is submitted to the instructor, the ink interpreter runs in the background on the student Tablet PC, creates a representation of the meaning of the submission, and stores that representation in the database. The student does not see the results of the interpretation, as we did not want the student to get distracted correcting a possible wrong interpretation. In CLP, the interpreted value of the input is not as important as the seamless interaction between the student and the user interface. Some interpretation errors are fine; the instructor is trying to understand the overall understanding in the class rather than looking at correct interpretations of every student response. This design decision differs from other software programs that interpret user input. In many applications, the user is aware that the interpreted value of the drawing will be used as input to a program. Such a program will often convert drawn shapes to a computer rendered version, and the user can correct any errors in the interpretation before the input is used. In many cases this user interface design choice is a critical one [Blostein 2002].

The first version of our interpreter was able to interpret answers that were a string, a sequence, a Scheme expression, a number, or a true-false answer [Rbeiz 2006]. Most of the exercises that are completed in 6.001 recitations fall into one of these categories. The accuracy rate of the interpreter for these types has been improved in the current version [Tay 2006], by allowing the instructor to specify more explicit types, such as sequence of number, or sequence of strings. Two new types being added are the box and pointer diagrams, described in this thesis, and marks, which are user ink strokes whose semantics depends on a background image [Koile, et al 2007a, Wu 2007].

2.4 Aggregation

Once the answers are interpreted and stored in the database, the instructor can aggregate the submissions. Aggregating submissions is the process of grouping submissions together with other submissions that have a similar meaning. The first version of our aggregator clusters alphanumeric answer types using edit distance, a similarity measure for two strings, to compute a score for each pair of answers [Smith 2006]. The aggregator clusters the answers into bins, up to a maximum number of bins. After the maximum number of bins has been reached, bins with similar answers are grouped together. The instructor then can see a description each bin and the number of students whose answers are in each bin.

3. Cons and the Box-and-Pointer Diagram

In the introductory computer science course at the Massachusetts Institute of Technology, called *Structure and Interpretation of Computer Programs*, students learn the Scheme programming language. In the language there exists a construct known as "cons", which pairs together two values. After two values have been paired with a "cons" statement, the first value can be accessed with the "car" operator, while the second can be accessed with the "cdr" operator. In order to pair the two symbols "foo" and "bar" together, for example, the statement "(cons foo bar)" is evaluated. By combining multiple cons statements, more complicated data structures can be created. In Scheme, lists and trees can be created by using multiple, nested cons statements. In order to visualize and represent these complicated structures, students learn to create box-and-pointer diagrams which use three kinds of shapes. Rectangles split in half, known as a box, represent the cons structure. Words represent symbols that the values in the cons statement. Arrows, known as pointers, indicate the values in the pair. The pointers can reference another box (cons statement) or a symbol. An example of a box-and-pointer diagram is shown in Figure 7.

By combining multiple cons statements, very complicated data structures can be created in Scheme. In a singly linked list, there is a cons structure for each position in the list. The car references the content at that position, while the cdr references the next cons structure in the list. The final cons statement in the list contains the value null. An example of the two element list "(foo bar)" is shown in Figure 7. In order to create a tree, the content of an element in a list can refer to another list. An example of a tree structure is shown in Figure 8.

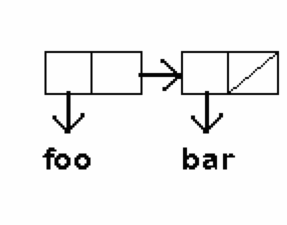


Figure 7: Simple box-and-pointer diagram

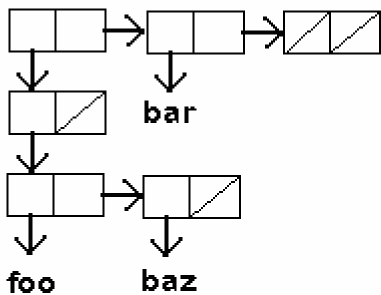


Figure 8: Complicated box-and-pointer diagram

When we start to think about interpreting box-and-pointer diagrams, there are certain factors that should be considered. One important thing to note is that there are often many ways to draw box-and-pointer diagrams. Figures 9b-d show three drawn box-and-pointer diagrams. Each diagram is drawn differently, but each one has the same meaning. Some of the differences in style that do not affect the meaning, or the semantics of the diagram include whether or not arrows start inside or outside of the box, whether arrows point up or down, and boxes location relative to other boxes. These are just some of the ways that diagrams can be drawn by students in class, and because the stated goal of allowing students to use CLP without any training or instructions on how to draw box-and-pointer diagrams, all of these styles should be considered when implementing the interpreter as long as the style does not change the semantics of the diagram. Two features are important for the semantics of a drawing. The boxes must be drawn horizontally with a vertical dividing line in the box. The left half is assumed to be car, and the right half is assumed to be cdr. Arrows must be drawn with a head on one and only one side. While some people use lines to connect objects in box-and-pointer

diagrams, the meaning of these lines can sometimes be ambiguous. Drawing each arrow with a head ensures that the proper referencing is chosen. If these two rules are met, the interpreter should be able to interpret the diagram.

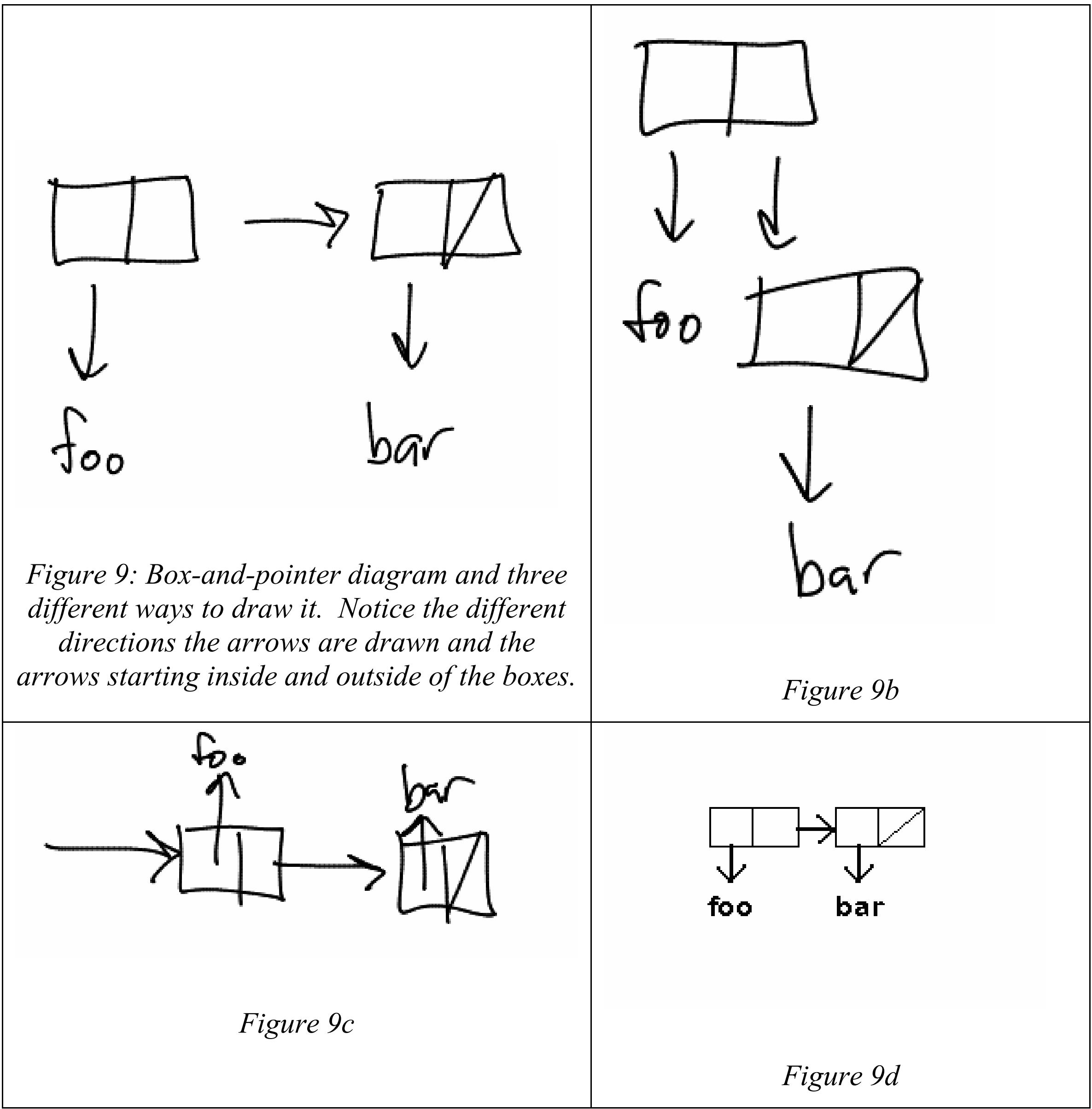


Figure 9: Box-and-pointer diagram and three different ways to draw it. Notice the different directions the arrows are drawn and the arrows starting inside and outside of the boxes.

Figure 9b

Figure 9c

Figure 9d

When we think about the semantic representation of box-and-pointer diagrams, or the way in which we represent the meaning of the diagrams so that the automatic aggregator can use it, a few options come to mind. Many instances of box-and-pointer

diagrams can be represented in a list format. A list format sequence is an alphanumeric sequence that can be used to represent lists and trees in Scheme. The mapping between box-and-pointer diagrams and list representations is deterministic, and two box-and-pointer diagrams that have the same meaning will share the same list representation. Many box-and-pointer diagrams, however, cannot be represented as lists. If a diagram contains a loop, as in Figure 10, trying to convert the diagram to list structure will result in infinite recursion, and no list representation is possible. Also, list structures cannot represent many of the errors that are interesting to an instructor. The box-and-pointer diagram shown in Figure 11 has more than one pointer emanating from the cdr position of the first box and therefore is not valid. In CLP, it is much more important to record why a student's answer is wrong than whether or not it is a valid box-and-pointer diagram. If many students are drawing diagram with extra pointers, for example, the instructor can see this situation and cover in class why this sort of diagram is wrong. If the instructor only knows that the diagram is not valid, she might miss an opportunity to help students understand the course content. It is important for our system, therefore, to not simply classify a student answer as correct or incorrect, but to save a semantic representation for each answer. It is also important that our implementation of semantic representations make no assumptions about the validity of an answer.

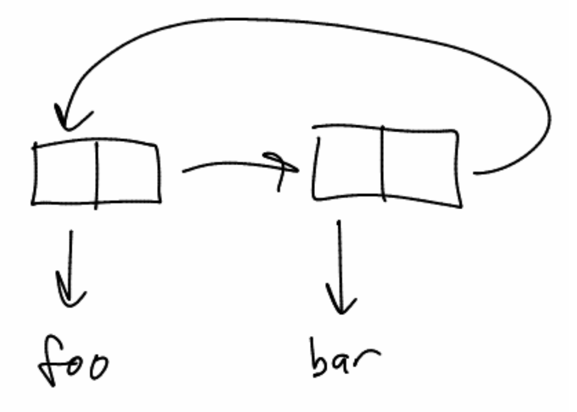


Figure 10: Loop box-and-pointer diagram

Figure 11: Illegal box-and-pointer diagram - 2 pointers from one box

4. Shape Recognition

As previously mentioned, in order to classify student answers into groups so that an instructor can view the results of aggregation, each student's written response must be interpreted. Interpreting an answer drawn on a tablet with digital ink involves tagging sections of the drawing with information about the semantics of each section. [Rbeiz 2006] The challenge in interpreting box-and-pointer diagrams is separating and interpreting both text and shapes such as rectangles and arrows. Recognizers exist for text and shapes, but not for an ink sequence that contains both. As a result, the box-and-pointer interpreter uses two separate recognizers to perform shape recognition and text recognition separately, then combines the results.

The box-and-pointer interpreter contains four stages, as shown in Figure 12. In the first stage, the digital ink from a student submission is run through the Microsoft Text Recognizer. This recognizer produces a list of all words that it thinks were contained within the ink. At the same time, in a second stage the digital ink is passed into the LADDER shape recognizer that has been tuned for use in the box-and-pointer domain. The shapes that are recognized in this stage are combined with the text items that were found in the text recognizer, and this collection of possible text and shape are the hypothesized objects in the diagram. In the third stage hypothesized objects are input into the interpreter's clean-up. The clean-up stage produces a list of objects that are actually in the diagram from the list of possible objects that were recognized. The list of actual objects is often much shorter than the list of possible objects, as it eliminates combinations of objects that are impossible, such as one stroke representing an arrow and the text "->". After the actual objects are known, in the last stage the list of actual objects

is passed through the geometric section which identifies the objects at the head and tail of each arrow. After this stage is finished, all of the semantics have been extracted from the digital ink, and the interpretation process is complete.

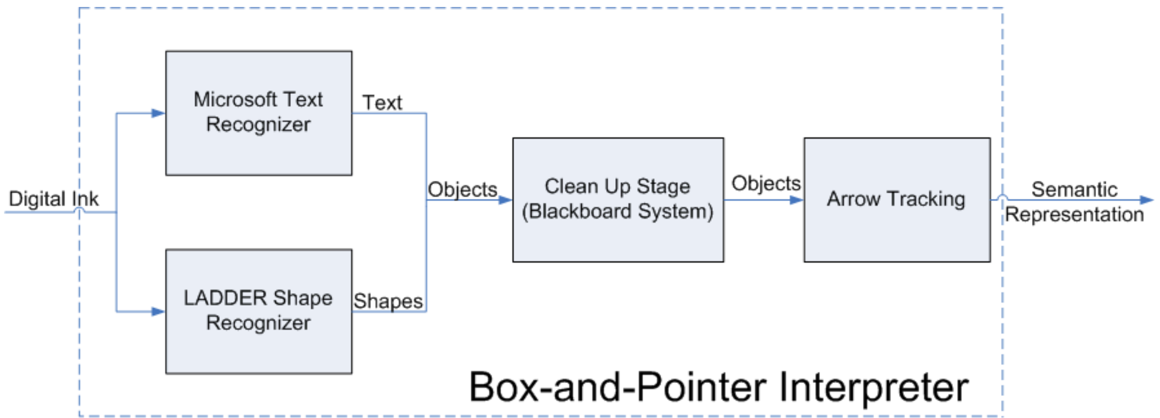


Figure 12: The four stages of box-and-pointer interpretation

4.1 Text Recognizer

Tablet PCs that run Windows operating system use Microsoft Text Recognizer for the recognition of written text. This is also the recognizer that was chosen to recognize the text portions of box-and-pointer diagrams. This recognizer was chosen because it is the most stable and accurate text recognizer available for recognizing short strings, and is installed on all of the Tablet PCs used in the Classroom Learning Partner project. When the interpreter begins, it sends a copy of the digital ink to the text recognizer. All of the words that are recognized in the drawing are added to a list of objects recognized in the diagram. In Figure 13, the text was correctly recognized as the word "foo" and the word "bar". The output of the recognizer is often, e.g., because text objects and shape objects may share the same stroke. In Figure 14, the output of the text recognizer included the word "sis", which it recognized from the highlighted arrow pointing to the word "foo". It is the goal of the clean-up stage to correct these errors.

Figure 13: Text recognized as two objects, "foo" and "bar"

Figure 14: The highlighted area was incorrectly recognized as "sis"

The recognizer also creates errors by not recognizing text that does not form an English word. The recognizer is biased to recognize words in English. Many words which are written in box-and-pointer diagrams are not English words, such as “foo” and “bar”. These words are commonly used in 6.001 due to their occurrence in the textbook for the class, Structure and Interpretation of Computer Programs. In order to recognize these words, it is necessary to add them to the dictionary of words to which the Microsoft recognizer compares hypotheses. A list of common words used in box-and-pointer diagrams is added to the Microsoft Text Recognizer dictionary before interpretation begins.

4.2 Shape Recognizer

Our box-and-pointer recognizer first preprocesses each ink stroke into a primitive point, line, or curve [Sezgin 2001]. Then, LADDER is used to identify new, unique shapes [Hammond 2005]. LADDER if a domain-independent shape recognition engine. With LADDER, users define shapes that are to be recognized in their particular application. Each shape is defined by listing the primitives or other previously defined

shapes and the constraints that specify the relationship between the different components. The constraints can be chosen from a list of predefined constraints, such as "near" or "perpendicular" or they can be written as a Java procedure. An example of a LADDER shape definition for an arrow is shown in Figure 15.

```
(define shape Arrow
  (isA Shape)
  (components
    (Line shaft)
    (Line head1)
    (Line head2)
  )
  (constraints
    (near shaft.p1 head1.p1)
    (near head1.p1 head2.p1)
    (equalLength head1 head2)
    (longer shaft head1)
    (acuteMeet head1 shaft)
    (acuteMeet shaft head2)
  )
  (aliases
    (Point head shaft.p1)
    (Point tail shaft.p2)
  )
  (display
    (color green)
  )
)
```

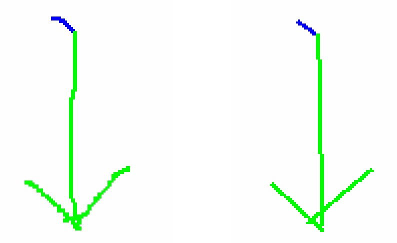


Figure 15: LADDER definition of an arrow is shown at left. The hand drawn arrow, and the recognized lines (shown in green) displayed by the LADDER Recognizer software

The LADDER shape recognizer takes as input digital ink and returns a list of shapes it has recognized. For box-and-pointer domain, there are two possible shapes that can be recognized. A box is a horizontal rectangle split down the middle by a vertical line. An arrow consists of three lines which meet at one end of the longest line, and the other two lines meet with an acute angle to the longest line. Any other lines that are recognized but not attributed to a shape are returned by the recognizer along with the list of shapes. These orphan lines may actually be part of a shape that was missed by the shape recognizer, and could be recognized by the clean-up stage of the interpreter.

Although the recognizer is tuned to recognize shapes in the box-and-pointer domain, often recognizes shapes incorrectly. There are two common errors that affect recognition results. The first error involves recognizing shapes in strokes that are actually words. Because there is no separation of text strokes and shape strokes before recognition takes place, the recognizer tries to find primitives in every stroke, including ones that are part of text. Because text is often more complicated than shape strokes (in the case of cursive writing) or have many more strokes that comprise the object (in the case of printed letters) many lines are found by the LADDER recognizer. Not only does this slow down the recognizer, as constraints are applied to each primitive when trying to find a shape, but it also provides false shapes which must be pruned by the clean-up stage. Figure 16 shows a box-and-pointer diagram that has been run through the LADDER shape recognizer. In this simple diagram, 11 extra lines were recognized because of the two words. The word "foo" resulted in five false lines, and the word "bar" was recognized as six false lines.

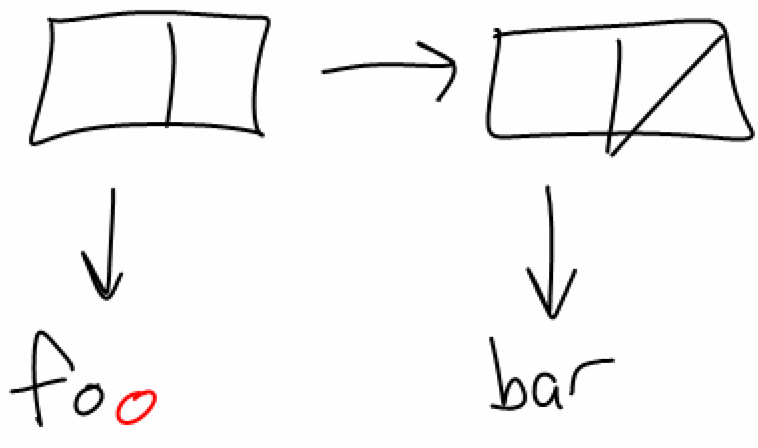


Figure 16: The 'o' was misrecognized as three lines

The other common error occurs when the shape recognizer misses an actual shape. This can occur from any number of reasons. Often the Microsoft ink library does not record information at a high enough resolution or the shape was drawn in a way that hinders recognition. This can happen when a student draws a shape sloppily, so that it can be recognized by a trained human eye, but does not fit the rigid constraints set forth in LADDER. In this case, the shape is not returned by the recognizer, but instead the individual lines are returned. The clean-up stage tries to find these shapes by performing analysis on orphaned lines with an alternate set of constraints than the LADDER recognizer uses. In Figure 16, the arrow connecting the two boxes was not recognized as an arrow, but as four lines. This was caught in the clean-up stage, and correct interpretation was possible.

Classroom Learning Partner is written in C# running on .NET technology and uses Microsoft Ink data structure to store the ink information. The LADDER system is written in Java and uses an array of points with timestamps to represent the ink information. In order to run the recognizer it is necessary to start the LADDER system in a different process and pass information about ink strokes from C# to Java, and information about recognized shapes from Java to C#. When the interpreter is called for the first time from CLP or another program, the interpreter creates a new process that starts the Java virtual machine. The C# program creates a new socket, and passes the address of this socket to the new Java process. All communication between CLP and the shape recognizer takes place through this socket. The new process creates an instance of the LADDER shape recognizer and a wrapper that marshals information to and from the

socket. This process is only started once, and subsequent calls to the shape recognizer use the same port. This minimized the time lost to creating the process.

After the process is created, the wrapper listens for connections on the socket. CLP then connects to the socket and requests a new recognition instance. The wrapper accepts the new recognition instance and begins listening for strokes. A stroke is a list of two dimensional Euclidian coordinate and timestamp pair. Each element in the stroke is sent over the socket. When the stroke is completely sent, the recognizer performs recognition on the stroke. The LADDER recognizer operates on one stroke at a time, recognizing primitives from the new stroke, then building shapes from the new stroke and shapes recognized from previously recognized strokes. After the recognizer is finished processing the stroke, it sends a message over the socket asking for the next stroke. When the last stroke has been sent and recognized by LADDER, CLP sends a message informing the wrapper to read back shape information, and that recognition is complete. The wrapper then sends information about each shape and primitive recognized through the socket, which is read by CLP and used to create a list of shapes recognized in the diagram. This interaction protocol is shown in Figure 17.

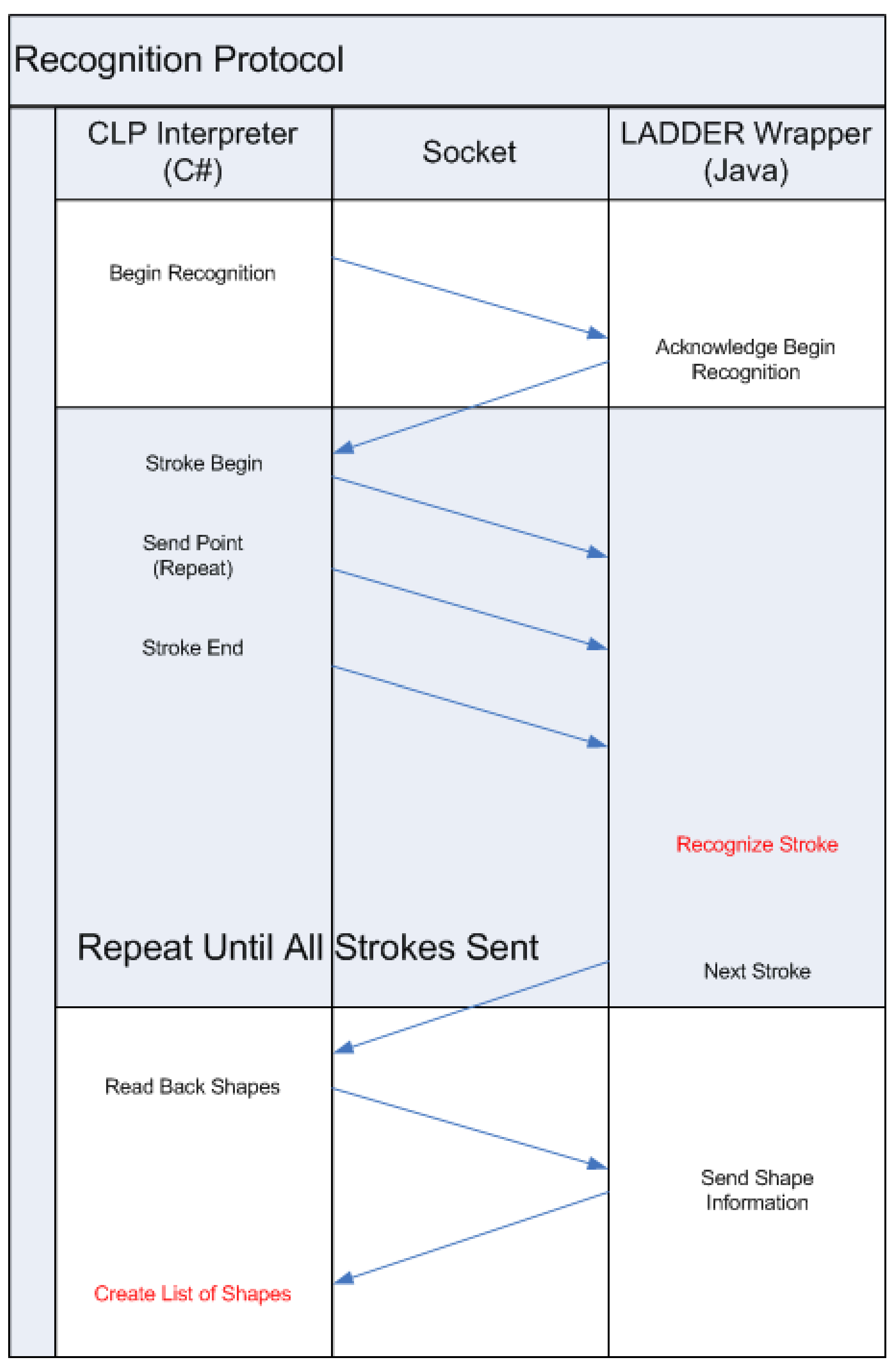


Figure 17: Protocol for communicating with Java LADDER recognizer from C# - There are long pauses when the LADDER recognizer recognizes the stroke, which is why recognition is multi-threaded on the C# side.

4.3 Clean-up Stage

When the ink has been recognized by the shape and text recognizers, the output of the two is passed into the clean-up stage of the interpreter. As previously mentioned, the recognized shapes and text, known as the recognized objects, often contain many errors. Because the two recognizers operate independently, the output of the two often conflict, creating a set of hypothesized objects that is not valid. A set of objects is invalid if one piece of a stroke is recognized as two different objects. This situation often happens when the shape recognizer finds shapes in written text, and when the text recognizer recognizes shape strokes as text. The goal of the clean-up stage is to locate and fix the conflicting hypothesized objects, and produce a valid set of objects. These output shapes and text are the objects that are used when computing the semantic representation of the drawn box-and-pointer diagrams.

We have implemented the clean-up stage using a blackboard architecture [Nii 1986a, 1986b]. The blackboard problem solving model is analogous to a group of experts standing around a blackboard. Each expert can make changes to the information on the blackboard, and each one is an expert in a different field. As the problem progresses, the experts take turns making improvements to the solution until no more improvements can be made. A blackboard consists of three components. The information known to the system is contained within the blackboard. The blackboard is a data structure that can contain information of any type. Before a blackboard system begins to solve a problem, the input to the system is contained in the blackboard. Knowledge sources act on the information in the blackboard, and each try to make improvements to the solution. Each knowledge source uses a different rule to improve the accuracy of the solution. The last

important component of a blackboard system is the dispatching system for knowledge sources. Only one KS can act on the blackboard at once. At any time, a knowledge source can request to make a change to the blackboard. If no other knowledge source is making a change, it is allowed. When no more improvements can be made, the solution is read from the blackboard.

The input to CLP's clean-up blackboard is a set of recognized objects. As the clean-up stage progresses, different knowledge sources act on the set, and can remove or add shapes to this set. When a knowledge source deems an object to be correct, the shape is moved to a list of known objects. When the clean-up stage completes, all objects in the output set form a valid hypothesis. There are seven different knowledge sources that operate on the hypothesized object set. The first knowledge source attempts to find boxes that were not recognized by the shape recognizer. The constraints used to recognizes boxes are relaxed, and if a set of lines conforms to the new relaxed constraint, the box is added as a possible shape. The second knowledge source does the same for pointers. The third knowledge source moves objects from the input list to the output list if no other object shares the same ink.

The remaining knowledge sources are designed to fix common recognition errors. With these recognition errors corrected, it is easier for the first three knowledge sources to fix conflicts. The fourth knowledge source uses the fact that when a box is recognized by the LADDER recognizer, it is nearly always a box. The constraints are set very tight, and the false positive rate of boxes in the shape recognizer is very low. When this knowledge source is used, it removes all text objects that share a stroke with a recognized box.

The first four knowledge sources are able to fix some of the conflicts, but most of the time the clean-up stage reaches a point where no more optimizations can be made, but there are still many objects left in the input set. Two very common recognition errors that are not fixed by the first four knowledge sources are the misrecognition of text as lines and arrows. Because written text is very complicated, the shape recognizer divides the words into many lines. Often times these lines fit the constraints of an arrow, and are recognized as such. One knowledge source then tests each arrow whose ink is also part of a recognized text object. If the arrow is separate from the rest of the text object, it is saved. If the arrow is in the middle of the rest of the text object, it is removed. Another knowledge source does the same for lines were recognized inside of text objects. Once these arrows and lines are removed, the text can be moved to the output object list.

The final knowledge source rerecognizes text. Often times, the text recognition is wrong because shapes are recognized as part of a text. After some of the ink has been interpreted as shapes, the ink is rerecognized, so that only text ink is run through the text recognizer. This oftentimes results in more accurate text recognition. This knowledge source is often able to improve the results after other knowledge sources have moved objects to the output list. The knowledge sources make changes to the input and output object sets until no more improvements are possible. After this has occurred, the output set consists of the objects recognized by the interpreter.

An example of the clean-up system running is shown in Tables 1 through 4. The original ink input to the interpreter is shown in Figure 18.

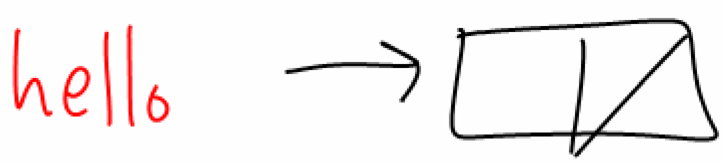
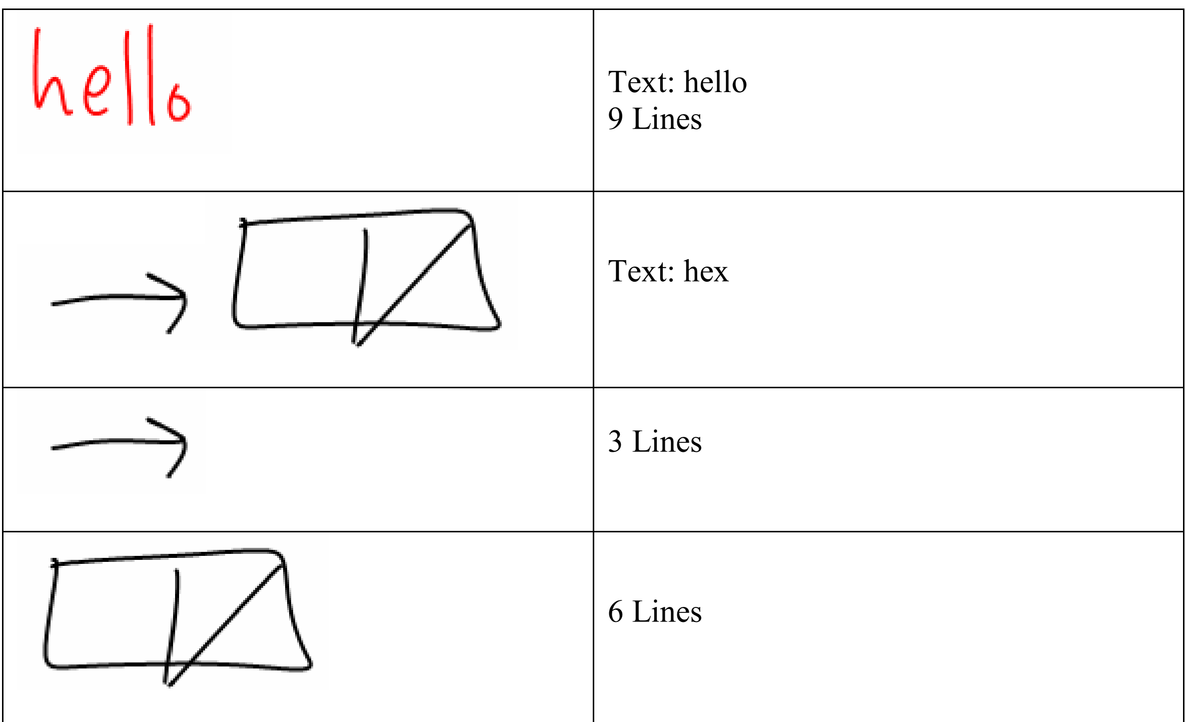


Figure 18: Original drawn box-and-pointer diagram

Table 1 shows the list of recognized objects available after text and shape recognition,. Notable errors include the word "hello" being interpreted at nine lines in the shape recognizer. The text recognizer found the word "hex" in the ink that forms the arrow and box. Also, the shape recognizer missed the arrow and box completely, instead recognizing three lines for the arrow and six lines for the box. These errors must be corrected in the clean-up stage. First, the knowledge sources that look for arrows and boxes operate on the set of hypothesized objects. The box finder finds a box out of six lines, and the arrow finder recognizes an arrow from three lines. The arrow recognizer also adds three arrows to the list of shapes, all of which were found from the lines in the word "hello". All objects share ink with another object, so there are no valid objects to place in the output list.



Strokes	Recognition results
hello	Text: hello 9 Lines
	Text: hex
	3 Lines
	6 Lines

Table 1: The recognition results for each set of strokes returned by Microsoft Interpreter and the LADDER recognizer.

Strokes	Recognition results
hello	Text: hello 2 Lines 3 Arrows
	Text: hex
	Arrow
	Box

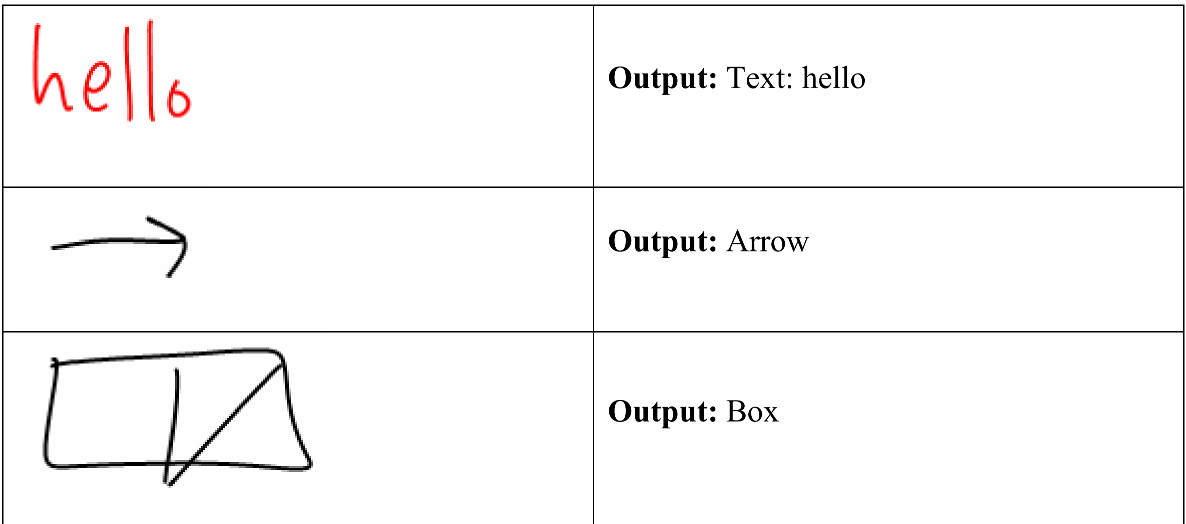
Table 2: The results after the box finder and arrow finder have made improvements to recognition in the clean-up stage.

The next knowledge source removes the box from the text object “hex”. The text recognizer then tries to rerecognize the arrow, which does not produce a recognized text object. The box and arrow then do not share ink with another shape and so the two shapes are moved to the output set. The last two knowledge sources then act on the input list, and remove the arrows and lines from set. This leaves only the text “hello”, which is added to the output set.

hello	Text: hello 2 Lines 3 Arrows
	Output: Arrow
	Output: Box

Table 3: After text that shared the strokes from the box were removed, the arrow and box shared strokes with no text, and are moved to the list of output objects.

The output set of objects now contains all of the objects that exist in the box-and-pointer diagram. This list is then used to create the semantic representation for the diagram.



Input	Output
hello	**Output:** Text: hello
	Output: Arrow
	Output: Box

Table 4: Results of interpretation after orphaned lines are removed.

4.4 Semantic Representation

As previously mentioned, CLP stores the meaning of an interpreted student submission in a format known as the semantic representation. Two properties that a semantic representation must have are the ability to store all information about the meaning of a submission, and the ability to store the information in the form of text, so that the information can be read to and from a database. The storage format for box-and-pointer semantic representations XML, which is the same as for the other CLP answer types [Rbeiz 2006]. The semantic representations for non-box-and-pointer answers in CLP are rendered as strings, which are easily read by humans. It is often difficult, however, to render box-and-pointer semantic representations in a string form so two library procedures were created instead.

4.4.1 XML Format

In order for the interpreted answer to be used in the aggregator, the semantic representation must be saved to the database. It is saved to the database in XML format.

In order to do this, an XML node is created for each object in the diagram, either a text, arrow, or box. The XML node contains all of the information that in needed to understand the meaning of the document. The different fields of information that are saved for each type is shown in Table 5. A XML parsing function is provided to convert from the XML representation to the C# representation object and back.

Object	**Property**	**Description**
Box	ID	The object unique identification number for this box.
	NullCar	0 if the car of this box is null. 1 if the car of this box contains contents.
	NullCdr	0 if the cdr of this box is null. 1 if the cdr of this box contains contents.
Arrow	ID	The object unique identification number for this arrow.
	From	The identification number of the object that this arrow points from.
	FromHalf	CAR if this arrow points from the car position of a box. CDR if this arrow points from the cdr position of a box. NONE if this arrow does not point from a box.
	To	The identification number of the object that this arrow points to.
Text	ID	The object unique identification number for this box.
	Content	The alphanumeric string that was drawn.

Table 5: List of information saved in XML semantic representation.

4.4.2 Semantic Representation Builder

The semantic representation builder is a library procedure used to simplify the creation of box-and-pointer semantic representations. The process to create a new semantic representation involves two parts. First, the shapes and text objects that are found in the diagram are input into the builder. The builder then computes how the objects relate to each other. Information about what arrows point to and from is essential in understanding the meaning of a box-and-pointer diagram. For each arrow recognized in the diagram, an arrow tracking algorithm calculates which other object the arrow points to and from. The angle from each point on each bounding box to the line on which the arrow lies is checked, and if the bounding box straddles the line on which the arrow lies, the arrow points to or from that object. Then, the closest object to or from the arrow is recorded. If the arrow points from a box, then the portion of the box that the line intersects is found. For each arrow, the boxes or text to and from which the arrow point are recorded. This information is stored in the semantic representation.

4.4.3 Semantic Representation Reader

The semantic representation is available to developers who wish to use information about the meaning of a box-and-pointer diagram. However, parsing the XML document is tedious, and a simple representation is not available for some box-and-pointer diagrams. For this reason, a library procedure was created to help developers read box-and-pointer diagrams into their applications. The semantic representation reader provides support for most operations which a developer of an application, e.g., an aggregator [Pal 2007], would need to extract information from a box-and-pointer representation. The procdures that are included in the reader are shown in Table 6.

Function	Description
FindHead()	Attempts to find the head of the box-and-pointer tree. If the diagram is a connected, acyclical tree, the head is returned. Otherwise, returns no shape.
CarContents(Box)	Returns the shape or text that the first half (car) of this box points to.
CdrContents(Box)	Returns the shape or text that the second half (cdr) of this box points to.
GetBoxes()	Returns a list of all of the boxes in the diagram.
GetArrows()	Returns a list of all of the arrows in the diagram.
GetTexts()	Returns a list of all of the text objects in the diagram.
IsLabel(Text)	Returns true if the text points to another object. False if no arrow points from the text to another object.
LeadingArrow()	Returns true if there is an arrow pointing to the head of the diagram, false otherwise.
LabelPointsTo(Text)	Returns the text or box that the label points to, if the text is a label. Returns no shape otherwise.
GetDescription()	Attempts to create a list description of the diagram. Returns nothing if diagram is cyclical or not connected.

Table 6: Procedures provided by the Semantic Representation Reader for other applications that wish to interpret box-and-pointer diagram.

Most of the procedures provided are straightforward, but there are two that are worth noting. The first procedure to note is the FindHead procedure. If the diagram drawn represents a connected tree, this returns the head of the tree. The tree can either be a box or a text. The procedure retains a set of each text and box in the diagram. It then removes from the set the object which each arrow points to. It then iterates through the list of arrows, and if the arrow points from something, the object to which it points is removed from the set of objects. After viewing each arrow, if more than one object remains, there is no head. If there is one object remaining, that is the head. If more than one object remains, there is more than one head, and the procedure returns no shape.

Another procedure worth noting is the GetDescription() procedure. This procedure attempts to create a list description of the diagram by finding the head of the list, and then recursively walks through the tree. As noted earlier, not all box-and-pointer diagrams, can be parsed into a list description, e.g. if there is a loop in the diagram. A list representation nonetheless is quite useful when possible, hence the creation of this procedure.

5. Results

The interpreter was completed late in the Spring of 2007. Testing was conducted to obtain information about the accuracy and performance of the interpreter using two different kinds of data sets: diagrams collected from former 6.001 students, and diagrams collected from current 6.001 students on two different days during recitation. Two aspects of the performance of the interpreter were analyzed. The accuracy of the interpreter is an important characteristic, as well as the time elapsed during interpretation.

There are two aspects of accuracy that can be explored with respect to the box-and-pointer interpreter. The first aspect is shape level accuracy, which is the accuracy of the interpreter when trying to label each individual shape or text in the diagram. The second aspect of accuracy is stricter and involves measuring the ability of the interpreter to interpret the correct meaning of the handwritten diagram. Both aspects are useful when measuring the accuracy of an algorithm. The shape-based measure can help determine under what circumstances the interpreter encounters accuracy problems. If this information is known an instructor can create exercises that focus more on diagrams that the interpreter can recognize well. If the shape-based accuracy is high, then even if the entire diagram were not interpreted correctly, the meaning can be inferred from the shapes that were interpreted correctly. On the other hand, if no error correction is going to be done in the aggregator, then the semantic accuracy rate is the important measure of the accuracy of the system.

Samples of drawn box-and-pointer diagrams were collected from 15 users who have taken 6.001. Each user provided eight different diagrams. Table 7 shows the accuracy rates for each type of object. The data showed that boxes were recognized 70%

of the time, arrows were recognized 45% of the time, and the existence of a text object was recognized 54% of the time. The content of the text was recognized correctly 21% of the time. Out of the samples taken, the interpreter produced the correct semantic representation 25% of the time. Most of the successful interpretations, however, were executed on very simple diagrams. The accuracy fell dramatically when the diagram became complex. Similar results were obtained from the data sets collected in the 6.001 class.

Shape	**Accuracy Rate**
Box	70%
Arrow	54%
Text	56%
Text Content	21%
Semantics	27%

Table 7: Recognition rates for different objects in box-and-pointer diagrams. Text is counted as correct if the interpreter is correct in recognizing that a text object exists in a group of strokes. If it is interpreted correctly, then Text Content is also counted.

When analyzing the accuracy of the interpreter, a strange interpretation problem appeared. If the user drew a shape quickly, the LADDER recognizer would incorrectly create lines from the shape. An example is shown in Figure 19. The highlighted part of the diagram shows one stroke that comprises three sides of a box. In most cases, this stroke would be processed and three lines would be created. In this case, the preprocessor fails to accurately segment the stroke, and the two thin lines shown on the diagram were created from the stroke. The top and side lines were not created, and instead a line was created that spanned from the top left of the box to the bottom right.

One assumption that was made when the interpreter was designed was that the preprocessor would correctly segment the strokes. The accuracy of the clean-up stage and the interpreter depend on this assumption. While this error does not account for all of the errors in interpretation seen in the results, around 5% of the incorrectly recognized boxes were a result of incorrectly segmented strokes.

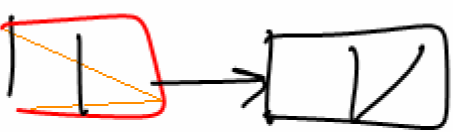


Figure 19: Polyline misrecognized as one line

Box-and-Pointer Diagram	**Average Running Time**
bar foo	12.2 Seconds
bar foo baz	102.3 Seconds

Table 8: The average running time of the interpreter on two box-and-pointer diagrams.

Another measure of the performance of the system is the running time of the interpreter on diagrams of differing levels of complexity. Two different diagrams were drawn by different users who have taken the introductory computer science course at MIT and are familiar with box-and-pointer diagrams. These diagrams were run through the interpreter and the running time of the interpreter was measured. During the course of an application, the LADDER shape recognition process only needs to be started once, and on subsequent interpretations the process is already running, the time to start this process, about 5 seconds on the HP TabletPCs used in the experiment were not counted in the running time totals. The average running time increases with the number of strokes in the diagram. In particular, it increases dramatically with the number of text strokes. When the text was removed from the more complicated diagram shown in Table 8, the average running time decreased to 25 seconds. This descrease is because the text strokes are divided by the shape recognizer into more primitives. There are more lines found in each text object, and each of the lines is used when the recognizer tests constraints to find new shapes.

6. Discussion and Future Work

While the accuracy of the box-and-pointer interpretation system for Classroom Learning Partner has been shown to approach 70% element recognition for boxes, and around 50% recognition for text and arrows, there are many ways to improve the accuracy and speed of the system. One way to improve the accuracy would be to implement the algorithm described by Shilman and Viola for recognition and grouping of strokes into text and shape strokes [Shilman 2002]. Much of the time used to interpret a box-and-pointer diagram is spent recognizing shapes in the LADDER interpreter. During this computation, much time is spent recognizing the shapes contained in text strokes. Because text strokes are normally much more complex than strokes drawn for shape objects, the text strokes are divided into more primitives. More lines are detected in the average text stroke than the average shape stroke. These primitives then are used to find other shapes in the LADDER interpreter. The constraint testing in LADDER is exponential in the number of primitives, so the amount of time to recognize all of the shapes in the LADDER interpreter increases dramatically when text is written. If we could separate the text strokes from the shape strokes, the amount of time used by the LADDER interpreter would decrease, making classroom interaction faster. The method described by Shilman and Viola finds the optimal chunking of the diagram into sections that contain only text or only shapes. A search tree that contains all possible stroke chunking is created. This tree is then traversed, recognizing text and shapes for each chunking hypothesis, until the optimal recognition results are found. In order for this strategy to work, the recognizer must be called many times. The LADDER recognizer could not be used in this way, as it is too slow.

Reimplementing LADDER in C# would improve the speed performance of the interpreter greatly. Because CLP is written in the C# programming language, and LADDER is written in Java, the two languages are run in separate virtual machines. As a result, when recognizing shapes the interpreter must create a new Java process to start the LADDER shape recognizer. CLP then must pass stroke information through a socket to LADDER, which then recognizes the strokes and passes the shape information back through the socket to CLP, where the information is marshaled into C# objects so that the clean-up phase of the interpreter can use the shape information in the interpretation. Starting the LADDER process is only done once per instance of CLP, and takes 3.2 seconds. Passing the stroke information takes about 7 seconds for a medium complexity box-and-pointer diagram. Passing the shape information back to CLP takes a negligible amount of time. Overall, using the Java LADDER implementation can add up to 10.2 seconds for a medium complexity diagram interpretation. The speed of interpretation is a major factor in the usefulness of interpretation in class, thus implementing LADDER in C# would improve the speed performance of the interpreter.

In addition to the problems associated with the LADDER recognizer running in a different process and language, the recognition speed is incredibly slow. While separating the text ink from shape ink would greatly improve the speed of the shape recognizer, it still may not be fast enough to run on the student machines. A better, although more expensive system architecture could involve one or more very fast computers dedicated to interpreting the submitted diagrams. These computers could take the load off of the already taxed, slower student Tablet PCs, and improve the speed of interpretation. This would again improve the speed of the interaction that can take place

between the students and instructor, allowing the instructor to more immediately receive feedback on how the students understand the material.

The box-and-pointer interpreter was designed with a few goals in mind. The need for off-line interpretation, no user verification of interpretation results, and no student or algorithm training ensured that the interpreter could fit into the existing CLP architecture without distracting students. These design goals presented challenges. To overcome these challenges, the system was designed to use different state of the art recognizers for shapes and text objects. The results were then combined. This approach is unable to account for all of the errors, however, because the text recognizer produces many errors when presented with a shape, and the shape recognizer produces many errors when presented with a text object. As a result the interpretation rate is low. The lessons learned and framework produced, however, provide good starting points for further investigation of incorporating hand-drawn sketched answers such as box-and-pointer diagrams into Classroom Learning Partner.

References

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

Alvarado, C and Davis, R, SketchREAD: A Multi-Domain Sketch Recognition Engine, *UIST,* 2004.

Alvarado, C., Oltmans, M., and Davis, R., A Framework for Multi-Domain Sketch Recognition, *AAAI,* 2002.

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. *SIGCSE,* 2004.

Anderson, R, Anderson, R, McDowell, L, and Simon, B, Use of Classroom Presenter in Engineering Courses, *Proceedings of ASEE/IEEE Frontiers in Education Conference,* 2005.

Anderson, R, Anderson, R, VanDeGrift, T, Wolfman, S, and Yasuhara, K, Promoting Interaction in Large Classes with Computer-Mediated Feedback, *CSCL*, 2003.

Blostein, D., Lank, E., Rose, A., and Zanibbi, R., User Interfaces for On-Line Diagram Recognition. In *Selected Papers From the Fourth international Workshop on Graphics Recognition Algorithms and Applications* (September 07 - 08, 2001). D. Blostein and Y. Kwon, Eds. Lecture Notes In Computer Science, vol. 2390. Springer-Verlag, London, 92-103.

Tracy Hammond and Randall Davis. LADDER, a sketching language for user interface developers. Elsevier, Computers and Graphics 28, pp.518-532. 2005.

Koile, K., Chevalier, K., Low, C., Pal, S., Rogal, A., Singer, D., Sorensen, J., Tay, K.S., Wu, K. Supporting Pen-Based Classroom Interaction: New Findings and Functionality for Classroom Learning Partner. First International Workshop on Pen-Based Learning Technologies, May 24-26, 2007.

Koile, K., Chevalier, K., Rbeiz, M., Rogal, A., Singer, D., Sorensen, J., Smith, A., Tay, K.S., Wu, K. Supporting Feedback and Assessment of Digital Ink Answers to In-Class Exercises. To appear in *Proceedings of the Nineteenth Conference on Innovative Applications of AI*, July 22-29, 2007.

Koile, K. and Singer, D., Improving Learning in CS1 with Tablet-PC-based In-Class Assessment, *ICER 2006 (Second International Computing Education Research Workshop)*, September 9-10, 2006, University of Kent, Canterbury, UK.

Koile, K. and Singer, D., Development of a Tablet-PC-based System to Increase Instructor-Student Classroom Interactions and Student Learning, *The Impact of Pen-based Technology on Education; Vignettes, Evaluations, and Future Directions.* Berque, D., Gray, J., and Reed, R. (editors). Purdue University Press, 2006.

Nii, H.P. Blackboard Systems, Part One: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures. *AI Magazine 7*(2): 38-53, 1986a.

Nii, H.P. Blackboard Systems, Part Two: Blackboard Application Systems, Blackboard Systems from a Knowledge Engineering Perspective. *AI Magazine 7* (3): 82-106, 1986b.

Pal, S, *Aggregation Sketched Box and Pointer Diagrams in Classroom Learning Partner,* 2007

Rbeiz, M, Semantic Representation of Digital Ink in the Classroom Learning Partner, Massachusetts Institute of Technology MEng Thesis, 2006.

Sezgin, M. "Generating Domain Specific Sketch Recognizers From Object Descriptions" *Proceedings of the MIT Student Oxygen Workshop*. 2002.

Shilman, M and Viola, P, "Spatial Recognition and Grouping of Text and Graphics" *Eurographics*, 2004.

Smith, A., "Aggregation of Student Answers in a Classroom Setting", Massachusetts Institute of Technology MEng Thesis, August 2006.

Tay, K, "Improving Sequence Interpretation in CLP", unpublished memo 2006.

Wu, K. MEng Thesis, to be completed December, 2007.