4 Ink Interpretation

4.1 Purpose

The purpose of the Ink Interpretation module is to extract semantic information from digital ink, providing the aggregator with enough semantic information to group student answers into equivalence classes. From an implementation point of view, the ink interpretation module tags digital ink bits with metadata about the semantic information of the bits.

![Figure 4.1 Ink Interpretation Example of an Inked Box](image)

An example of ink interpretation is shown in Figure 4.1. A user draws a box using digital ink on a tablet-PC. The tablet-PC libraries capture a sequence of point coordinates (1). A sketch recognizer such as SketchRead [Alvarado and Davis, 2004], recognizes the sequence of dots as four intersecting lines (2). With knowledge of a domain, for example a box diagram domain, the interpreter deduces that the user drew a box (3). If we chose a mathematical domain, the interpreter would deduce that the user drew a square.

This thesis focuses primarily on the design and implementation of an appropriate framework for text and diagram interpretation. The implemented framework supports interpretation of strings, numbers, and symbols (namely arrows). In a second phase, outside the scope of this thesis, the interpreter will be expanded to include support for diagram interpretation. There currently exist state of the art recognizers that provide accurate recognition for text (Microsoft Tablet-PC recognizer), or diagrams [Alvarado and Davis, 2004]. Researchers recently have been integrating diagram and text recognition, such as in the recognition of chemical structures, but no off-the-shelf
recognizer provides functionality for recognizing both in a single sequence of ink. The goal of this thesis is to build on current state of the art recognizers to support the combination of diagram and text recognition.

In this thesis, I use the words recognition and interpretation, which are close in meaning but are subtly different. Recognition denotes the process of identification and classification of a certain object. I use the word interpretation as a superset of recognition to mean that there is context supporting the recognition. For example, recognition is a matter of shape matching, whereas interpretation is the process of passing a shape through a recognizer, and making a choice given context.

4.2 Background in Handwriting Recognition

The difficulties encountered in recognizing handwriting come mainly from variations in writing styles. These difficulties translate into a non-perfect degree of recognition accuracy. While a recognizer with 99% accuracy is considered a feat in academia, a commercial application with this accuracy would yield bad results when handling a large volume of inputs (in millions). The United States Postal Service, for example, uses a handwriting recognizer to recognize addresses printed on envelopes. We can assume that the software recognizes all machine written addresses and has a 99% accuracy rate on handwritten addresses. The USPS website [USPS 2005] claims that USPS delivered over 206 billion pieces of mail in 2005. There are multiple categories of mail but if we assume that handwritten addresses appear on 1% of the total volume, then the recognizer would miss the recognition of 1% x 206 billion x (100%-99%)= 20.6 million envelopes. This inaccuracy is potentially one of the reasons why people sometimes do not get their mail. In reality, USPS has workarounds because it benefits from constraints on the possible number of possibilities for zip codes, states etc… USPS also has a person overseeing the results of the recognizer, and this person can make corrections when the recognizer signals low confidence in results. Srihari details the system in place at USPS in [Srihari and Keubert, 1997].
In this thesis, achieving high recognition rates is important because students would like the machine to “get their answer right” or they will not trust the system; they may become distracted if concerned about recognition instead of working problems in class.

There is a distinction that has traditionally been made between on-line and off-line recognition. In on-line systems, the temporal information about the writing speed, acceleration and order of line segments making up a word is available to the recognizer. Tablet-PCs are examples of on-line systems. Off-line systems only use the handwriting information stored in an image. The USPS handwritten address recognizer is an example of an off-line system. There are methods to attempt the extraction of this temporal information from the handwriting in off-line systems, but they are generally not used in commercial systems. This extra information makes for higher recognition rates, leading on-line systems to perform better. Systems using tablet-PCs can be considered on-line because they can capture spatial temporal features, which can be used in recognition.

To some degree, recognition systems attempt to replicate human’s abilities to recognize handwriting. In this light, it was observed that humans recognize handwritten words based on the knowledge of possible character combinations. As a result, humans are able to recognize words with blurred or missing characters. [Jacobs et. al 1994] Systems performing recognition try to mimic humans. Thus, in recognizing natural language, a holistic (where the entire word is recognized at a time) approach may be more effective than a character based approach (where each character is recognized individually as part of the word). It is important to note that when a person does not recognize a word, she deciphers the different letters and looks words up in a dictionary. In the case of USPS, a character recognition system might work better than a word recognition system since the words used are not always legal (i.e., part of a dictionary).
A typical architecture for a recognizer:

![Diagram showing the architecture of a handwriting recognizer.]

Figure 4.2 Handwriting Recognizer Architecture

As shown in Figure 4.2, each recognizer has several components. The feature extractor attempts to reduce noise found in the input and extracts features relevant to the recognizer. Each recognizer is different in implementation in the sense that the internals of a recognizer, as well as the features emphasized, vary. Examples of these features include slant ("slant" is defined as the average slope of a word with respect to the vertical direction), outer contours (extracted using Gabor Filters) etc. Using these features, the recognizer extracts a list of N-Best possibilities. The recognition process is divided into two sub-processes: segmentation and recognition. Most recognizers operate in a divide and conquer approach: they divide the strokes or the image into pieces and then perform recognition on these pieces. Typical segmentation approaches are segmentation in words or characters. Character segmentation is a challenging task because it is difficult to determine the beginning and ending of a character. Following segmentation, recognition is then performed on the strokes using several AI and pattern recognition algorithms. These pattern recognition algorithms take advantage of statistical information stored in training data in conjunction with shape matching algorithms used in computer vision. The statistical information helps algorithms deal with noise and variations in handwriting, whereas the structural approach with shape matching algorithm helps with machine-printed handwriting. Some of the AI techniques used are:

- Support Vector Machines
- Hidden Markov Models (HMM) [Hu et al, 1996] [Yasuda et al, 2000]
- Neural Networks
- Genetic Algorithms
- Convolutional Time Delay Neural Networks (TDNN)
The latest handwriting recognition methods are very similar on a high level to speech recognition [Makhoul et al 1994] and seem to be focused on modified Hidden Markov Models (HMM) customized to handwriting.

Using HMM, there are generally two basic approaches to perform recognition: word-based recognition and character-based recognition. In word-based recognition, each word has an HMM, which makes the recognizer fairly hard to train because each HMM needs multiple training examples. This approach is limited if the lexicon is big because it has problems recognizing out-of-vocabulary words, i.e., words not present in the training data. Systems with word-level models are therefore appropriate for tasks with small lexicons. For unconstrained handwriting, recognizers typically perform character recognition where each character has an HMM. The HMM, however, can produce an illegal set of characters. The features used in each HMM, i.e., what defines state and transition probabilities, vary by implementation. The HMM produces an N-Best List which is then compared to a list of possibilities and checked against a dictionary or language model. From 26 letters, an HMM potentially can generate all words in a dictionary. Word recognition is performed by matching a possible sequence of character models with the help of a given dictionary.

The recognizer feeds a list of possibilities to the language model module. The complexity of this module depends on implementation: the language model could be as simple as a lexicon of words and a string matching algorithm between candidate words to rank the lexicon. [Plamondon and Srihari, 2000] Another possibility is a dictionary improved by incorporating statistical information for each word. Context-free grammars, and N-Gram class models can also be used as language models, as they are in speech recognition.

One area of research relevant to this thesis is how much weight to give to the language model module and to the recognizer module. If a user writes “confidence”, for example, the recognizer might accurately output “confidence”. The language model, however, could reject the possibility since “confidence” is not legal and take the alternate which is “confidence”. While the language model could potentially recognize the word as it was written, it may use its dictionary to correct the result.
Finally, due to how handwriting recognizers are implemented, they have difficulty recognizing very slanted handwriting. Writing legibly at a 90 degree angle, for example, would throw off most recognizers.

For a more extensive background on handwriting recognition, see [Liu and Cai 2003] and [Plamondon and Srihari, 2000].

4.3 Background in Sketch Recognition

Sketch and Diagram recognition is inherently different than handwriting recognition due to the nature of the task. Single stroke recognition, for example, where recognizers perform incremental character by character recognition such as recognition in handheld computers, is not applicable in sketch recognition. Users tend to draw sketches in multiple steps. It would be terribly awkward to draw a pulley in a single stroke.

Similarly to handwriting recognition, an important sub-problem in recognition is the segmentation of strokes: where are the beginning and the end of a symbol? Sketch recognition also has to deal with the difficulty of context because unlike language, there exists no language models for sketching. A representation of the knowledge contained in a sketch therefore needs to be put in place.

The MIT Design Rationale Group (DRG) has extensively studied sketch recognition. The DRG group took advantage of the fact that complex sketches are composed of simpler sketches and defined a hierarchical sketch recognition language called LADDER. [Hammond and Davis, 2005] They defined primitive objects such as lines and curves usually drawn in a single stroke, making strokes easier to segment. Compound objects are made of primitive objects and the properties of these compound objects vary from domain to domain. A rectangle, for example, could be interpreted as a box in a diagram domain and as a block in a construction domain.
Figure 4.3 Definition of an arrow in LADDER

Figure 4.3 shows the definition of an arrow as a list of object definitions with some constraints in the LADDER recognition language. Recognition is done in incremental, unobtrusive fashion. At no point is the user asked to redraw something because the recognition engine misinterpreted the sketches. If the system tries to interpret the user’s strokes before it has enough information, it thus is more likely to err in interpreting pieces of the user’s drawing. The system is aware of when it has enough information to interpret a piece of the drawing. Furthermore, the system incrementally adds data to its internal representation making it more complete. Symbol recognition in SketchREAD is done in a 4 stage process, using a blackboard architecture, as illustrated in Figure 4.4:

- **Bottom-Up Step:** Primitive objects and simple compound objects are recognized. [Sezgin et al 2001]

- **Top-Down Step:** Domain-dependent objects are recognized, as in identifying 4 intersecting lines such as in Figure 4.1 as a box in a box and diagram domain. If a compound object has been hypothesized, the search for missing strokes from that compound takes place.

- **Ranking Process:** Many different interpretations may have been proposed for the same strokes. Based on previously interpreted parts of the sketch, the system identifies temporal and spatial context for the newly recognized patterns and uses this context to assign likelihoods to the templates that were generated in steps 1 and 2. [Alvarado et al, 2002] A Bayesian network is dynamically constructed to represent each hypothesis. The hypotheses are then compared using the likelihood of the Bayesian network. The Bayesian network (illustrated in Figure 4.5)
integrates the influence of stroke data and domain specific context in recognition, enabling the recognition engine to recognize noisy input. In addition, input is incrementally added to the Bayesian Network as the sketch is being drawn. The system is designed to be dynamic and flexible. [Alvarado and Davis, 2005]

- **Pruning Process:** The system prunes unlikely interpretations.

---

![Figure 4.4 Recognition Process in the drawing of a force pushing on a rectangular body.](image)

As shown in Figure 4.4, the blackboard data structure is divided into various levels of information. Italics indicate that an object was hypothesized by the system. In most cases, bottom up information causes the recognition of higher level objects, but in the case of the mechanical body, top down information reinforces the system’s hypotheses.
Shown in Figure 4.5, the Bayesian network created from the description of an arrow. The Bayesian network integrates the influence of stroke data and domain specific context in recognition. In this example, Ls denote the lines and Cs denote the connections. The likelihood of the Bayesian network is used as a score in the recognition of different objects.

It is important to note that because the recognition algorithm is stroke-based, spurious lines and over-tracing hindered the system’s performance in both accuracy and running time. A preprocessing step to merge strokes into single lines greatly improved the system’s performance. This approach yielded great results with 2D sketches across multiple domains.

For a more comprehensive summary on sketch recognition, see [Alvarado, 2004].

4.4 Architecture

Version one of CLP is designed to interpret and aggregate handwritten text and arrows using existing state of the art recognizers. Interpreting both text and arrows presents a challenge because recognizers, e.g., the Microsoft English Handwriting Recognizer and the Sketch Recognizer from the Design Rationale Group at MIT [Alvarado and Davis, 2004], are able to recognize text or a sketch, respectively, but not both in the same sequence of ink.

In the context of mathematics symbols and mechanics, Dr. LaViola from Brown University was able to accurately recognize mathematic symbols. In addition, he extracted semantic context from the mathematical equations written by users and used this information to animate physics experiments. [LaViola et. al, 2004] Our goals are
similar, so we replicated a similar architecture. The CLP ink interpreter uses a two tiered architecture similar to that of LaViola’s MathPad program:

- An **Interpreter** that performs recognition and semantic information extraction from digital ink.
- A **Renderer** that renders and displays semantic representation of digital ink. This module displays to the user what it “thinks” the user input was. Rendering is useful for ensuring that the recognized information matches the user’s input. (Building a recognizer from scratch such as MathPad is beyond the scope of this thesis.)

Unlike MathPad, however, our interpretation happens synchronously, i.e., after the user inputs ink. While asynchronous interpretation provides less time to complete, it requires the use of an event-based system within Classroom Presenter\(^1\). Doing so would break the separation between CLP and Classroom Presenter and thus would require more maintenance when time for software upgrades. The performance of synchronous interpretation is negligible for a sentence or two of handwritten text, so we thought it more advantageous to keep ink interpretation separate from Classroom Presenter. The architecture, however, is modular enough to enable a switch to an asynchronous architecture in the future if desired.

In addition, the user does not see the recognition results because we do not want the user to worry about the accuracy of the interpretation. The aggregator [Smith 2006] takes into account interpretation errors by acknowledging the recognition confidence provided by the interpreter. Furthermore, the system is designed to give the instructor an overall picture of understanding in the classroom, and is not designed to give an exact account of how many students got a correct answer.

\(^1\) CLP is built on top of Classroom Presenter.
Shown in Figure 4.6 is the ink interpreter architecture, which leverages the power of the two recognizers mentioned above. The Ink Analyzer segments text and sketches and passes corresponding strokes to the appropriate recognizer. Semantic information then is extracted from partial results.

Shown in Figure 4.7 is a diagram we would like the system to interpret eventually: an environment diagram from the MIT introductory computer science course (6.001). The environment diagram has text and drawings intertwined. The challenge will be to differentiate between the drawings and text.
As our first text and sketch interpretation task, we will start with a simpler box and pointer diagram shown in Figure 4.8.

![Figure 4.8 Box and Pointer Diagram](image)

This task falls in line with current research being done on chemical structure recognition as well as some research done in vision. In his Phd thesis, Sajit Rao extracted the structure of a directed graph and rendered it, as shown in Figure 4.9. [Rao 1998]

![Figure 4.9 Directed Graph Recognition and Rendering](image)

Another challenge that the interpreter currently faces is to convert the ink format into some sort of intermediate language such that a diagram or text can be reconstructed or aggregated. In this thesis, I refer to the intermediate language with the term *semantic representation*.

The first step taken in implementing our ink interpreter was to implement the handwriting component, which interprets text and arrows.
Shown in Figure 4.10 is the architecture of the handwriting recognizer:

- The **Ink Segmentation** module segments ink into individual chunks. Chunks are elementary units that are supposed to be individual words or arrows.
- The **Chunk Error Correction** module attempts to fix common errors common of the Ink segmentation process such as splitting a word into two words, or combining two words into one.
- The strokes of each chunk then are passed to the **Microsoft English Recognizer** which outputs several hypotheses ranked by a confidence score.
- The hypotheses then are sent to the **Language Model** module, which uses a domain-specific dictionary and knowledge of expected exercise answer type to choose the best hypothesis.

### 4.5 Implementation

**Scenario**

The following example illustrates the interpreter in the context of a class exercise. The instructor asks students: “What is the type of the following Scheme expression: `(lambda (a b) (+ a b))`.” The answer to the question is: ‘number, number → number’. Other answers such as ‘`num, num → num’, ‘#, # → #’ or any of their derivatives are all considered correct. A student writes ‘`number, number → number’ on his/her tablet. The ink segmentation module identifies four chunks as shown in Figure 4.11:

- ‘`number,’ tagged as a word
- ‘`number’ tagged as a word
- → tagged as a drawing
‘number’ tagged as a word

Figure 4.11 Ink Segmentation Illustration

The Chunk Error Correction module identifies no errors and passes the chunks to the recognizer which outputs a list of results for each chunk, ordered by confidence. The output is fed to the language module, which outputs the following semantic representation:

```xml
<Answer Type="SEQUENCE">
  <Chunk Type="STRING" Confidence="Strong">number,</Chunk>
  <Chunk Type="STRING" Confidence="Strong">number</Chunk>
  <Chunk Type="ARROW" Confidence="Intermediate"/>
  <Chunk Type="STRING" Confidence="Strong">number</Chunk>
</Answer>
```

The renderer takes the semantic representation and outputs the string:

‘number, number → number.’

The remainder of this section focuses on the implementation details of the above example.

**Software Architecture**

Figure 4.12 is the UML diagram for our implementation. The conceptual design shown in Figures 4.6 and 4.10 was implemented as four modules:

- **Recognition**: The recognition package was responsible for Ink Segmentation, Chunk Error Correction, and Recognition
- **Semantic Representation**: This package defined the semantic representation and XML formatting necessary. It was used to manipulate the semantic representation.
- **LM**: This package was responsible for the language model
- **Renderer**: This package was responsible for rendering the different semantic representation
Calling the handwriting recognizer

The handwriting recognizer is called via the Interpreter class:

```csharp
Interpreter interpreter = new Interpreter(Microsoft.Ink.Ink ink);
SemanticRepresentation semRep = interpreter.Recognize(ExpectedType.SCHEMEEXPRESSION);
```
The interpreter is initialized with the ink object containing the strokes. It then is called using the Recognize method. The expected type of the answer is used to help bias results and therefore improve accuracy. ExpectedType can take multiple values:

- TRUEFALSE: true-false answers such as ‘#t’ or ‘true’
- MULTIPLECHOICE: multiple choice questions where one letter or number is the answer such as ‘A’ or ‘1’
- MULTIPLEEXERCISE: this is for multiple exercises per slide (i.e., multiple ink objects), which has not yet been implemented
- STRING: the answer is expected to be one or two words such as ‘error’, usually put in one chunk.
- NUMBER: the answer is expected to a number such as ‘1000’
- SET – the answer is expected to be an unordered set of numbers of strings such as ‘1, 3, 2’
- SEQUENCE: the answer is expected to be an ordered set of number or strings (including arrows) such as ‘(1 2)’ or ‘boolean → string’
- SCHEMEEXPRESSION: A scheme expression is code in the Scheme Language such as ‘(lambda (a b) (+ a b))’.

More examples are available in Appendix A.

**Ink Segmentation**

The Ink Segmentation module attempts to segment words and drawings from handwritten ink. This module was implemented using the Ink Analyzer module from Microsoft. Chunking user input provided us with a divide and conquer approach, which enabled us to do recognition error correction in the Language Model. This module, however, had the following shortcomings:

1. Combining two words (or a word and a drawing) into one chunk. Shown in Figure 4.13, the third chunk consists of ‘# →’ where ‘#’ and the arrow should each be in their respective chunks.
2. The word chunks and drawing chunks were ordered by type; words were ordered in a separate list than drawings. The Ink Analyzer assumed that drawings were not part of text. Order is defined as to which word or drawing comes first in an English sentence.

3. Dividing one word into two chunks as is illustrated in Figure 4.14.

4. Classifying drawings as words and vice-versa

There are no details on how the internals of the Ink Analyzer from Microsoft operate but I believe that it attempts to combine ink strokes that form legal words in the dictionary.

**Chunk Error Correction**

I only dealt with some of the issues presented because the solution to all four of the above issues involves writing a new segmentation module. To deal with chunk ordering (issue 2), I wrote a sorting function by implementing the `IComparable` interface in C# and using the already available `Sort` method within `System.Collections`. I make the assumption that people write in quasi-straight lines, thus enabling the system to compare two chunks at a time. If order were defined in a more complicated way, then the proposed approach would fail.
The sorting function compares the location of strokes’ bounding boxes ignoring context. It returns 0 if the bounding boxes of the ink strokes superimpose each other, -1 if the first bounding box comes before the second, 1 otherwise.

```csharp
Rectangle thisBoundingBox = this.BoundingBox;
Rectangle otherBoundingBox = otherChunk.BoundingBox;

if (thisBoundingBox.Equals(otherBoundingBox))
// this compares if the boxes superimposes one onto the other
{  
  return 0;
}

//check that they are on the same line:
//-- check that upper left corner of first box is higher than lower left //corner of second box
//-- check that lower left corner of first box is lower than upper left //corner of second box
//--check that first box comes first:
//-- check that upper left corner of first box comes before than upper left corner of second box
//--remember that y coordinates increase in the downward direction
if (((thisBoundingBox.Top <= otherBoundingBox.Bottom) &&
     (thisBoundingBox.Bottom >= otherBoundingBox.Top)) &&
     (thisBoundingBox.Left <= otherBoundingBox.Left))
{  
  return -1;
}

// this the case where the first box is either the first word
//-- above the second box
else if (thisBoundingBox.Bottom <= otherBoundingBox.Top)
{  
  return -1;
}
return 1; //if all else fails, then the boundingBox is in front of the other
```

**Figure 4.16 Implementation of the CompareTo method of the IComparable Interface**
Figure 4.16 outlines the details of implementing the comparator function, which bundled with the quick sort implementation in C#, enabled me to sort through the different chunks. I only dealt partially with issue 3 by first sorting all the chunks and then combining all chunks with intersecting bounding boxes. The reason the solution is partial is because if a person does not write in cursive but in a rather detached form, it is still possible for the system to identify ‘car’ as two chunks, ‘c’ and ‘ar’ instead of one chunk ‘car’. For example, Figure 4.17 illustrates a case where the resorting algorithm fails.

![Figure 4.17 Illustration of Chunk Ordering Problem](image)

Figure 4.17 illustrates wrong chunking of the word ‘nil’ where each letter is placed in a different chunk and the dot on top of the letter ‘i’ is also placed in a separate chunk. The numbers in the boxes correspond to the ordering of chunks as performed by my algorithm, which poses a small problem for recognition. When recognition is performed, the interpreted result is: ‘. n * l’. A hypothesis as to why this happens is that the two mechanisms I can think of to avoid this problem fail in this specific example:

1. The ink segmentation module checks the chunks against a dictionary. Each letter is a valid entry of the dictionary, which justifies the chunking.

2. The example is written in non-cursive handwriting where the characters are not attached together. I assume that the program dynamically computes spacing between letters, adjusted to each person’s handwriting. When a person writes a sentence, and words are accurately recognized, the program dynamically computes the space between characters in order to find the separation between words. In this example containing one word, the program could not rely on results of remaining words to deduce a robust indicator of spacing between letters, and thus assumed that the user wrote individual chunks.
Dealing with issue 4, where the ink segmentation module classified words as drawings and vice-versa, relied on a bias of the expected answer type. Unless the type is a sequence, there is no reason to expect the handwriting to include any drawing. In addition, if the answer is anticipated to be short, such as numbers or true-false answers, the interpreter put all the strokes of the answer in the same chunk, thus side-stepping the possible problems of issue 3, when words are divided into two chunks.

**Recognition**

Once the chunks are identified, they are stored in a list of `RecognitionChunk` objects. This list of chunks is ordered according to the method described in Figure 4.16 and each chunk only contains the strokes of that chunk. The recognizer thus is isolated from the context of the sentence, leaving context handling to the language model. In addition, the recognizer is biased in two ways:

- By adding a list of words to the user dictionary that are not typically available with a standard recognizer such as: ‘set-cdr!’ or ‘caar’ which are only relevant in the current domain
- By biasing based on the expected answer type. Knowing about a true-false answer, I identified six possible answers: ‘#t’, ‘#f’, ‘true’, ‘false’, ‘t’, ‘f’ and coerced the recognizer in this direction. This list is not exhaustive and more user testing augments the dictionary. It was just brought to my attention, for example, that some students may indicate true and false using bits, 1 being true, and 0 being false. The recognizer is also biased using the Factoid capability of the Microsoft Recognizer which had multiple features, such as being able to bias towards numbers, which is useful when expecting a number.

The recognizer passes recognition results ordered by recognition confidence to the language model. Recognition confidence is an internal measure of the confidence the recognizer has in its results; in the Microsoft Recognizer API, only three values are provided: Poor, Intermediate, and Strong. A quantitative value based on probabilities would have been more useful in this case.
Language Model

The language model module used is basic and consists of a lexicon of words. The architecture allows for a future increase in complexity. The language module is only called only if the recognition results are poor. I turned off the Microsoft Recognizer Language Module because it was not well-suited for Scheme\(^1\) code: the language module had been developed for natural language and performed badly with out-of-dictionary words such as the ones in Scheme. When recognition is poor, the language model module compares the various hypotheses of each chunk against entries in our dictionary. Because the recognition hypotheses are ordered by recognition confidence, the program picks the highest ranked hypothesis in the list of hypotheses. In the case in which an arrow is expected (i.e., the answer is of type SEQUENCE), the program waits for all the recognition hypotheses to be compared against the dictionary. If none of them matches, the module checks that the potential string is less than 2 characters and hypothesizes that it’s an arrow. This method is not solid arrow recognition, but the architecture lends itself to allowing small drawings to be part of the text. This method eventually will be replaced with an object recognizer, but for the time being, this hack does reasonably well.

Semantic Representation

The interpreter outputs the following semantic representation for the example shown in Figure 4.11:

\[
\text{<Answer Type="SEQUENCE"} > \\
\text{<Chunk Type="STRING" Confidence="Strong">number,</Chunk>} \\
\text{<Chunk Type="STRING" Confidence="Strong">number</Chunk>} \\
\text{<Chunk Type="ARROW" Confidence="Intermediate"/>} \\
\text{<Chunk Type="STRING" Confidence="Strong">number</Chunk>} \\
\text{</Answer>}
\]

---

\(^{1}\) The introductory computer science course uses the Scheme programming language.
The semantic representation contains the recognition result of each chunk and the confidence associated with the result. The representation potentially could be augmented to include the list of alternatives if an outside program such as the aggregator deemed it necessary. The confidence is included so the aggregator can determine the amount of flexibility needed in dealing with interpreter errors.

The following graph can be constructed from the XML representation for the semantic representation:

```
<Answer Type="BOXANDPOINTER">
  <Box id=1>
    <Text Confidence="Strong">X=0</Text>
    <Position x=0 y=0/>
  </Box>
</Answer>
```

![Figure 4.18 Graph for Semantic Representation](image)

The graph for a string answer is uninteresting. In a box and pointer diagram, however, it can provide a powerful tool for aggregation. Let’s take the simple diagram shown in Figure 4.19:

```
X=0  Y=0
```

![Figure 4.19 Simple Box and Pointer Diagram](image)

Here is the hypothetical semantic representation for the diagram:
The reconstructed graph would correspond to the one in the original diagram. This graph representation would simplify aggregation because classification of graphs is a well-studied problem. Furthermore, the representation would enable a renderer to reconstruct the interpreted input.

**Renderer**

The renderer displays the semantic representation in a human readable form. For the time being, it converts the XML representation into a simple string, but it eventually will be able to reconstruct an image from a text description.

**Alternatives**

I considered initially using the built-in Microsoft Recognizer and Analyzer. Early results, however, were very bad, prompting the alternative architecture with separate components for the ink segmentation and the language module.

### 4.6 Results

Recognition accuracy traditionally has been measured with a Word Error Rate. Due to the nature of our test cases, however, it was more appropriate to test the distance between the input and recognized strings in order to test for partial improvements. If the input string was “caar,” for example, and the subsequent recognition results were “cr” and “car”, it was important for our accuracy measure to take into account the partial progress made, something a word error rate measurement would not do. The *Levenshtein distance* [Atallah 1998], also called *edit distance*, accurately measures the distance between two
strings. The distance between two strings is given by the minimum number of operations needed to transform one string into the other, where an operation is an insertion, deletion, or substitution of a single character. In the above example, the edit distance between “caar” and “cr” is 2, while the distance between “caar” and “car” is 1.

The interpreter was tested in two experiments:

- A controlled experiment in which users were asked to ink predetermined answers, and the interpreted results were compared to the actual answers. This experiment was performed multiple times to test the different versions of the interpreter.
- In-class experiments in which I manually compared student submitted answers and their interpretations. Not knowing what the prior answers were only allowed for manual comparison. This experiment was used to understand user behavior and how the interpreter did in a real-world situation.

In the controlled experiment, users were asked to ink twenty-one representative answers (Appendix A); I collected 167 inked answers from several users. These sample answers were stored in a database. The inked answers were dynamically interpreted, then stripped of spaces, and converted to lower case. They then were compared to the input. Samples were collected early in the process in order to test each version of the interpreter against all these answers. This process provided a consistent way to check for improvements in recognition accuracy. The design of the interpreter sometimes involved tradeoffs, and measuring increasing accuracy on the same data set was a reliable way to ensure progress. The way I measured improvement was to check whether the total edit distance between interpreted answers and inputs decreased. The total edit distance between interpreted answers and inputs also could be regarded as the total number of character errors between interpreted answer and input, a character error being a character deletion, insertion or substitution. I present the results of recognition of the current version of the interpreter. The experiment used the examples shown in Appendix A., which illustrate four of the seven answer types present in the database. The following types are subsets of other types and necessary for operation:

- MULTIPLE EXERCISE: this type identifies that the exercise contains multiple ink objects; it is a collection of strings and has not been implemented yet.
- **SET**: this type is a subtype of sequence because it is an unordered list, which the aggregator distinguishes from ordered sets (i.e., sequences). For the purposes of the interpreter, a set is the same as having a sequence.

- **NUMBER**: the handwriting recognizer has been built with this type in mind, therefore, using it presents no problems. Number and character recognition are the traditional problems handwriting recognition research has been solving. Current handwriting recognizers have 95% accuracy (i.e., 5% Word Error Rate) when recognizing numbers or characters. [MacKenzie and Chang, 1999]

- **MULTIPLE CHOICE**: this type is simply single character recognition.

Table 4.1 presents the results grouped by answer type and sorted by length in ascending order.

<table>
<thead>
<tr>
<th>Type</th>
<th>Input Text</th>
<th>Input Length</th>
<th>Mistakes</th>
<th>Total Char</th>
<th>Error Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheme Expression</td>
<td>(1 2)</td>
<td>5</td>
<td>2</td>
<td>32</td>
<td>6%</td>
</tr>
<tr>
<td>Scheme Expression</td>
<td>(* 1 2)</td>
<td>7</td>
<td>8</td>
<td>40</td>
<td>20%</td>
</tr>
<tr>
<td>Scheme Expression</td>
<td>(caar seq)</td>
<td>10</td>
<td>15</td>
<td>72</td>
<td>21%</td>
</tr>
<tr>
<td>Scheme Expression</td>
<td>(define x 3)</td>
<td>12</td>
<td>5</td>
<td>80</td>
<td>6%</td>
</tr>
<tr>
<td>Scheme Expression</td>
<td>(eq? id1 id2)</td>
<td>13</td>
<td>15</td>
<td>88</td>
<td>17%</td>
</tr>
<tr>
<td>Scheme Expression</td>
<td>(lambda (a b) (+a b))</td>
<td>21</td>
<td>17</td>
<td>119</td>
<td>14%</td>
</tr>
<tr>
<td>Scheme Expression</td>
<td>(car (quote (quote a)))</td>
<td>23</td>
<td>18</td>
<td>160</td>
<td>11%</td>
</tr>
<tr>
<td>Scheme Expression</td>
<td>(set-cdr! (last-pair x) x)</td>
<td>26</td>
<td>17</td>
<td>184</td>
<td>9%</td>
</tr>
<tr>
<td>Scheme Expression</td>
<td>(cons (cdr seq) (cddr se))</td>
<td>27</td>
<td>15</td>
<td>184</td>
<td>8%</td>
</tr>
<tr>
<td>Scheme Expression</td>
<td>(cons (cons x (+ 1 (+ 1 (seq-length seq))))</td>
<td>42</td>
<td>58</td>
<td>272</td>
<td>21%</td>
</tr>
<tr>
<td>Scheme Expression</td>
<td>(define x (let (( two '(2))) (list (cons 1 two) (list 1) two )))</td>
<td>65</td>
<td>82</td>
<td>416</td>
<td>20%</td>
</tr>
<tr>
<td>Scheme Expression</td>
<td>#, #, # -&gt; #</td>
<td>12</td>
<td>17</td>
<td>56</td>
<td>30%</td>
</tr>
<tr>
<td>Scheme Expression</td>
<td>boolean -&gt; string</td>
<td>17</td>
<td>4</td>
<td>112</td>
<td>4%</td>
</tr>
<tr>
<td>Scheme Expression</td>
<td>nbr, nbr, nbr -&gt; nbr</td>
<td>20</td>
<td>15</td>
<td>120</td>
<td>13%</td>
</tr>
<tr>
<td>String</td>
<td>Nil</td>
<td>3</td>
<td>0</td>
<td>24</td>
<td>0%</td>
</tr>
<tr>
<td>String</td>
<td>Error</td>
<td>5</td>
<td>0</td>
<td>40</td>
<td>0%</td>
</tr>
<tr>
<td>String</td>
<td>double-tree</td>
<td>11</td>
<td>3</td>
<td>88</td>
<td>3%</td>
</tr>
<tr>
<td>True-False</td>
<td>#t</td>
<td>2</td>
<td>0</td>
<td>16</td>
<td>0%</td>
</tr>
<tr>
<td>True-False</td>
<td>#f</td>
<td>2</td>
<td>2</td>
<td>16</td>
<td>13%</td>
</tr>
<tr>
<td>True-False</td>
<td>True</td>
<td>4</td>
<td>0</td>
<td>32</td>
<td>0%</td>
</tr>
<tr>
<td>True-False</td>
<td>False</td>
<td>5</td>
<td>0</td>
<td>40</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 4.1 Interpretation Results ordered by Type and Input Length
Error Rate was measured by dividing the number of mistakes over total number of characters. Figure 4.20 shows a graph of Table 4.1 results.
Interpretation Results

Figure 4.20 Graph of Interpretation Results ordered by Type and Input Length
Finally, Table 4.2 summarizes the accuracy of the interpreter per type.

<table>
<thead>
<tr>
<th>Answer Type</th>
<th>Error Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheme Expression</td>
<td>15.30%</td>
</tr>
<tr>
<td>Sequence</td>
<td>12.50%</td>
</tr>
<tr>
<td>String</td>
<td>1.97%</td>
</tr>
<tr>
<td>True-False</td>
<td>1.92%</td>
</tr>
<tr>
<td><strong>Total &gt;&gt;</strong></td>
<td><strong>13.37%</strong></td>
</tr>
</tbody>
</table>

Table 4.2 General Accuracy of the Interpreter

CLP has been deployed three times at the time of writing. Results of recognition were used to augment the dictionary. In addition, the interpreter did not do very well with messy handwriting. Knowing that even humans have trouble with messy handwriting, the result was not surprising. CLP will be deployed next academic year 2006-2007 again in introductory computer science classes.

4.7 Discussion

In general, the system performs well, and the deployments were a success because the aggregator was able to group interpreted ink into groups and display the results, despite interpretation errors. 13% error rate (or 87% accuracy rate) is not yet excellent but is acceptable for a first iteration. I expect to reach 5% error rate, the industry standard, if all the changes proposed in the Future Work section are implemented.

When taking a closer look at the results by type, the accuracy for string and true-false questions is 98%. This result, combined with experiments performed on number and character recognition, clearly demonstrate that our system can rival wireless polling systems such as PRS, in which a student submits an answer to a multiple-choice or true-false question using a transmitter. Furthermore, CLP’s ink interpreter allows student submission and aggregation of more than multiple-choice or true-false questions.

Difficulties arise when interpreting Scheme expressions and sequences. If we take a closer look at Figure 4.18, accuracy is not closely correlated with the length of the input. I hypothesize that it has to do with the nature of the input. The interpreter has more trouble, for example, with the shorter input ‘#, #, # → #’ than with ‘boolean → string’. In the rest of this section, I discuss the most commonly observed errors.
Incorrect Chunking

Figure 4.21 Illustration of Incorrect Chunking
In addition to the previously shown example in Figure 4.11, the ink segmentation incorrectly chunks the input, thus, causing the arrow not to be identified.

Ambiguity in the writing

Figure 4.22 Illustration of Ambiguity in Writing
The rendered interpreter result for this exercise was: ‘ubr,nbr,nbrsubr.’ To credit the interpreter, the ‘n’ in ‘nbr’ looks like a ‘u.’ It is only because we know the context of the question that we know that it was an n that was used not a u. Furthermore, connecting the arrow with the last word misleads the analyzer into thinking that it is part of the same word.

Incorrect Recognition

Figure 4.23 Illustration of Incorrect Recognition
The rendered interpreter result for this exercise is: ‘lee? id At id2)’. It correctly chunked the input but poorly recognized it.
Confusing letters and punctuation

The interpreted result for this exercise was: ‘#i#i#7#’. Besides incorrectly chunking the result, the recognizer confused the commas with the letter ‘i’.

Confusing numbers and parentheses

The interpreted result for this exercise was: ‘(define x 31’. In this example, the interpreter mistakes the last parenthesis for the number 1.

Space Elimination

The interpreted result for this exercise was: ‘(12)’. This elimination creates a problem in sets because the machine thinks that the user input a list containing the number 12 rather than a list containing the two numbers 1 and 2.

The improvements proposed in the Future Work section attempt to solve the problems outlined in this section.