Heraldry and Programming Languages: the Complexity of Natural Languages Examined through the Parsing of the Heraldic Blazon

Hans A. Mersinger

College of Saint Benedict/Saint John's University

Follow this and additional works at: http://digitalcommons.csbsju.edu/honors_theses

Part of the Computer Sciences Commons

Recommended Citation


http://digitalcommons.csbsju.edu/honors_theses/656

Available by permission of the author. Reproduction or retransmission of this material in any form is prohibited without expressed written permission of the author.
Heraldry and Programming Languages:
The Complexity of Natural Languages
Examined Through the Parsing of the Heraldic Language

A THESIS

The Honors Program

College of St. Benedict/St. John's University

In Partial Fulfillment

the Degree Bachelor of Arts

In the Department of Computer Science

by

Hans Mersinger

May, 1998
INTRODUCTION

The greater part of the world's troubles are due to questions of grammar.
--Michel de Montaigne, *Essays*

In most areas of oral and written human communication, grammar rules play an important role. While music and poetry are sometimes devoid of strict grammar, speech and written language usually depend on grammar to clarify the message being conveyed. When humans communicate with computers through programming languages, grammar becomes even more important. One reason it is difficult to create computer programs that truly interact with humans is that there are differences in the type and structure of the grammars that each program uses. An important challenge for programmers is to create a computer program which will convert a list of words (ex. a sentence) into a structure that defines the meaning of those words (ie. parsing).

I began this thesis because I wanted to complete a challenging project that would provide an enjoyable topic to program. I wanted it to involve research and theory computation that would be both informative and educational and I wanted the results to be useful to computer scientists as well as potential end users. The topic of parsing the heraldic language, a pseudo-natural language, fit these goals. I knew it would both challenge and educate me as I encountered, learned from, and dealt with, a number of the problems faced by computer scientists as they try to parse natural languages.

I initially planned to create a program to parse a sizeable subset of the heraldic language and ready the information to be passed on to a second program that would be able to graphically display the information. I soon discovered the importance of choosing a subset
of the heraldic language that would be feasible to try and parse within present time
constraints. As I realized the magnitude of this project and encountered unforeseen
difficulties, the size of the subset I used diminished considerably.

The parsing of the heraldic language is a project that could be useful in multiple ways.
Successful completion of this task would benefit people who want to learn what a blazon, the
written description of a coat of arms, looks like, and also would provide another source for
computer scientists to examine how others have attempted to parse natural languages.

Parsing the heraldic language, which is a kind of natural language, may not seem to be
significant to the casual observer. However, completion of this project will be very beneficial
because parsing the heraldic language will help me understand the parsing of other natural
languages. Natural languages have less strict rules than the programming languages which
computers deal with best. Completely converting the heraldic language, a pseudo-natural
language, into a grammar that is parsable by computers, would take me a long way toward
understanding how to create a language that is representative of a natural language and also
easily parsable.

When the topic of parsing natural languages arises, the problems of parsing natural
languages as compared to programming languages arise. These problems are a direct result of
the types of grammars that make up natural and programming languages. Natural languages
are defined by context sensitive grammar, while programming languages are defined by
grammars commonly called context free grammars. This means that programming languages
have very strict rules in regard to the construction of the strings in a language, making the
context irrelevant to determining meaning. In contrast, natural languages do not have such
ridged rules about construction of the strings, so the context in which the string is found plays
a very important part in determining meaning.

The differences between natural languages and programming languages cause
problems with parsing of natural languages by computers. When I attempted to code a
program that would be able to parse a subset of the heraldic language, I faced problems
dealing with how to handle the number of modifiers that heraldry uses, punctuation, word
choice, and other differences in grammar.

My desire to complete a computer program that would be able to interpret at least a
small subset of the heraldic language, prompted me to launch a study of how computer
scientists have attempted to parse both natural and programming languages. I found out that
the heraldic language is very similar to a natural language but has the benefit of not having
all of the complexity that natural languages have. I was then able to look at the types of
grammars that I would be dealing with before breaking the heraldic language itself down and
beginning programming.
COMPUTERS AND LANGUAGE

The idea of programming computers to understand human speech and written text has been the dream, even the "Holy Grail", of computer scientists and many others since before the first computers were created. Computers in countless movies and television shows have been able to understand, and therefore interact with, the humans around them, but until recently this possibility has been pure science fiction. Even today we do not have computers that can understand and interpret all human speech, but computers that run automated phone services do exist, as do various other voice recognition software programs. Although these are hopeful signs that one day it will be possible for computers to understand natural languages, that day is still far in the future.

Since computers were first conceived, computer scientists have been searching for ways to parse natural languages for one reason or another. Although many methods of parsing have been tried, some have had more success than others. There are two methods that are currently, and have been for some time, the most popular to research. The first is an attempt to achieve natural language understanding in computers by modeling the neural networks the brain uses to learn natural languages. The second method, which I attempted, is the hard coding of natural grammar rules in a complex algorithm.

Most research of this type is being done at large colleges and universities such as MIT, Georgia Tech, and New Mexico State which have whole departments devoted to natural language processing and understanding, among other cognitive sciences. There are also a number of associations, like the Association for Computational Linguistics, who draw together the leaders in the field to present papers at their yearly conferences about the advancements in
the field of natural language understanding.

Just as there are two major methods of researching the parsing of natural languages, there are two major and interconnected goals, theoretical and technological, that are motivating computer scientists to strive to create the perfect computer program that can understand a natural language (Dean 489). From the theoretical point of view, scientists have always been interested in just how we, as a species, function. The more precisely we are able to break down languages that are used to communicate, the better insight we will have into how the cultures which use those languages began and evolved. We will discover what the actual building blocks of the language are and might even learn some of the hidden fundamentals of natural languages.

The technological point of view deals with more “practical” goals. If computer scientists devise a way for computer programs to achieve true understanding of natural languages, they will be able to create computers which can carry out a large number of tasks for humankind.

The dream of creating computers which are able to understand natural languages will not be realized until scientists can understand and effectively model almost all aspects of human intelligence in a manner which can be programmed into a computer to help it understand natural languages. Unfortunately, there are a number of stumbling blocks on the path towards this goal of total natural language modeling. Programming languages are straightforward with little, if any, ambiguity. In contrast, natural languages are complex with a number of exceptions to grammar rules and much ambiguity built in. Also, the rules and points of ambiguity in natural languages are constantly changing. Drawing connections
between these languages is often difficult at best. To date, simply modeling a natural language with a less ambiguous grammar is the most valid option. Unfortunately this can often be a complex problem itself. (Dean 489).
CONTEXT FREE GRAMMAR/CONTEXT SENSITIVE GRAMMAR

The problem is that grammar defines language, and the grammar which defines one language often bears little resemblance to the grammar which defines another. Whether the languages in question are two spoken languages or a spoken language and a non-natural language, such as a programming language, large differences in grammar create large differences in the languages. This creates many problems when translating from one language to another. For this reason, it is helpful to look at how languages are broken down into categories based on their grammars before starting a discussion of specific languages.

There are, for the most part, two major types of grammars in use today; context free and context sensitive grammars. The grammar of almost every language will fit into one of these categories. Context free grammars produce context free languages. Context sensitive grammars produce context sensitive languages.

A context free grammar incorporates three concepts. The grammar must first have an alphabet of elements called terminals. These terminals are the elements used to build the strings in the language. Second, the grammar must include a set of symbols, called non-terminals, one of which is the start state, written as $S$. These non-terminals represent the productions used by the grammar to construct valid words of the grammar. Third, the grammar must have a finite set of productions of the form non-terminal -> finite string of terminals and/or non-terminals (Cohen 230). In most representations of context free grammars, upper and lowercase symbols are used to represent non-terminals and terminals, respectively. So, in a context free grammar the left symbol of the grammar can always be replaced with what is on the right side of the production. Context free grammars are not
context specific. This means that any place in a string where a non-terminal exists, it can be replaced with its equivalent terminals and/or non-terminals without regard to how the non-terminal relates to the syntax.

The derivation of a string can be graphically displayed by showing how each symbol is rewritten (Dean 495). For example, the following grammar,

\[
\begin{align*}
S & \rightarrow ACBC \\
A & \rightarrow a \\
B & \rightarrow b \\
C & \rightarrow A \mid B
\end{align*}
\]

is able to produce the following strings: aaba, aabb, abba, abbb. In this case the language that is defined by this grammar consists of only those four strings. A language is said to be defined by the strings that are accepted by its grammar. The derivation tree for this grammar is shown in figure 1. Parse trees, graphical representations of the productions used to arrive at any given string from the start state $S$ for each word created, is shown in figure 2.

Context sensitive grammars contain the same elements and rules as context free grammars, with one slight modification to the third rule. A context sensitive grammar allows terminals on either side of the grammar equation. For example, the grammar we looked at as a context free grammar could now be changed, resulting in a context sensitive grammar. One possibility, the grammar,

\[
\begin{align*}
S & \rightarrow ACBC \\
A & \rightarrow a \\
B & \rightarrow b \\
aa & \rightarrow C \\
C & \rightarrow A \mid B
\end{align*}
\]

would result in the following strings being accepted by the grammar: aaba, aabb, abba, abbb,
aba, abb, bba, bbb (figure 3). Allowing terminals on the left side of the equation creates more flexibility in the selection of elements to use in the grammar, although it makes it more difficult to check an input against the grammar to see if it is accepted.

Context free and context sensitive languages can both be categorized by two sub-divisions, ambiguous and non-ambiguous. Ambiguous grammars are those grammars which have more than one parse tree to derive a unique string. Figures 1 and 3 show examples of unambiguous grammars, while figure 4 shows an example of an ambiguous grammar. Having multiple parse trees producing the same word can cause problems with the interpretation of the language. The problem of having different paths that result in the construction of strings that look alike allows for confusion to enter into the language. It can be hard to tell which tree caused the construction of the word, an important fact to know because different derivations of a word can produce different meanings. Non-ambiguous grammars solve this problem by enforcing a one to one relationship between the parse tree taken and the string that is derived from that set of productions (Cohen 250).
PROGRAMMING LANGUAGES

Programming languages, unlike natural languages which will be discussed in the next section, have been specifically created for one purpose, to be easily understood by computers and programmers. For this reason, programming language grammars can be derived by productions of the form non-terminal goes to terminal and/or non-terminal. They have context free grammars so that they can be easily parsed. The grammars of most, if not all, programming languages are also non-ambiguous.

That a context free grammar has yet to be described in regards to its context is useful in programming languages because there are efficient ways to parse a context free grammar whereas there are not any for context sensitive grammars (Dean 495). This allows powerful compilers and interpreters to be written based on highly structured grammar rules. The more structured and defined the rules a grammar uses, the more like basic algebraic manipulation the parsing and interpretation of an inputed string becomes. This algebraic understanding is desirable because a computer functions best when dealing with algebraic manipulation.

While a non-ambiguous context free structure has benefits for programming, the problems that these languages face often occur precisely because they are so highly structured. The same non-ambiguous context free structure that allows for ease in parsing and interpreting programs also often makes these programs unreadable to anyone who has not thoroughly studied the language. This structure also makes some types of expressions more difficult to construct and therefore parse. The strict rules that a programming language’s grammar must follow in order to remain a context free grammar restrict flexibility in expressions. The valid expressions are commonly strict, straightforward elements which do
not allow much in the way of variation. This causes problems by forcing the person who is trying to express themselves in a programming language to conform their thoughts and views to the one way that something can be expressed no matter how it was expressed in the original natural language.

Another problem with the structure of programming languages is that complexity makes input errors more common. A computer does not understand what is being said in the same manner a person does. A person takes in a sentence and interprets it based loosely on familiar words and grammar rules, while taking into account the available context and non-verbal communication. A computer, on the other hand, reduces the input solely to the grammar rules that it knows. If there is an error with the input, the computer will usually not know what to do; the slightest variance in the expected grammar can throw it off.

This contrast in the way that computers and people understand language is only a surface symptom of a deep conflict in the way these languages are constructed. Looking at how natural languages are currently being studied gives a sense of the scope and complexity of the problems being faced in the attempt to create a fully functional grammar to describe them. In order to start the process of researching a natural language, it must first be broken down into a number of sub-categories. Some of the basic elements of a natural language that are important to look at include semantics, scope, and syntax (Dean 490).
NATURAL LANGUAGES

Natural languages, by definition, are those languages used by people to communicate and interact. While programming languages were specifically created to be easy for computers to understand, natural languages were some of the first languages used for communication after primitive gestures and other non-verbal communication. Natural languages, therefore, have slowly evolved and changed. These changes took place to meet the needs of the users whenever new situations which needed expression arose. Currently, natural languages can only be expressed by ambiguous context sensitive languages.

This is one of the major problems with trying to convert natural languages to something that can be parsed by a computer. For example, most natural languages have a built in vagueness factor that can be used to show uncertainty. In contrast, computers work well with polar values and have problems with vagueness. As natural languages evolve, exceptions to standing rules are a common result with the more common exceptions making their way into the formal grammar rules. The English language, for example, has the unofficial rule of "there is an exception to every rule." There is, in fact, the concept of "the exception which proves the rule." These exceptions to the fundamental grammar rules are eventually defined and clarified.

Although this can create a snarl of exceptions and loopholes, the basic elements of a language such as semantics and syntax, are not much affected. Semantics, the meaning or implied relationship of a set of one or more signs, and syntax, the method in which linguistic elements are constructed to form constituents, cause many of the problems with natural language parsing.
The necessity of determining semantics (i.e. the meaning of words) is, in fact, one of the most difficult problems encountered when trying to interpret natural languages through a vehicle such as a programming language. Natural languages often contain words and phrases that can have more than one meaning depending on the context of the sentence that they are in. This allows for a large amount of information to be implied within the context of the sentence. For example, there are at least forty definitions of the word ‘go’ in a good English dictionary (Dean 512). To illustrate this, look at the following sentences which all have different meanings for the word go; his hearing started to go, the motor won't go, a good rule to go by, and these books go on the top shelf. Each of these uses of the word ‘go’ has a different meaning which is determined by the context of the sentence. Another aspect of semantics which addresses multiple definitions is specifically identified as scope.

Scope is a common problem encountered when looking at natural languages. For example, consider the sentence: “Every boy loves a dog,” This is ambiguous because it is unknown if every boy loves the same dog, or if every boy loves a different dog(Dean 512). In a language’s common use, this ambiguity is normally solved contextually through an implied reference elsewhere. This is in contrast to programming languages where ambiguity has been purposefully excluded.

Syntax does not deal with the actual meaning of words as semantics does. Rather it addresses the order of the elements in the sentence. Important as it is, syntax alone is not relied on to form meaning in natural languages. In most natural languages, we see that syntax rules are only loosely followed at best. There are a vast number of ways that any one sentence can be constructed to achieve the same meaning in most natural languages. For
example, the sentence "I am going to take the test tomorrow morning before school" can also be written, "Before school tomorrow, I am going to take the test", "Tomorrow before school I am going to take the test", "Tomorrow I am going to take the test before school", or even "Tomorrow morning I have a test." This lack of a highly structured syntax in the grammar of natural languages can create problems in translation. When someone translates from one spoken language to another, they usually try to keep the meaning while disregarding the structure. However, when linguists attempt to translate between natural and programming languages, they use a finer scaled structure to solve some of these problems.

Currently the creation of sub categories to regulate and define the grammar is one method used to developed structure. Rather than working solely with nouns and verbs, most linguists use keys like noun_singular, noun_plural, noun_proper to allow greater flexibility in accepted input by creating a larger number of options. This is done to try to create stricter relationships between elements in a sentence so that some of the simple types of sentence reordering, such as those described in the last paragraph, are possible. When that doesn’t work successfully, the easiest way to solve problems with ambiguous elements of grammar, like pronouns, is simply to disallow them.

These methods that linguists use to model natural languages, give computer scientists an option to use to break a natural language’s grammar into a grammar that can easily be scanned and parsed by a computer. By looking at how others have attempted to solve the problems of interpreting natural languages, we can begin to understand what has worked and what hasn’t.
HERALDIC LANGUAGE

Heraldic languages became prevalent about the time of the First Crusade, circa 1095 AD. They were created to provide a standard form to describe the coats of arms of the various wealthy and important individuals of the time. As Neubecker says in his book A Guide to Heraldry, "Arms represent people or groups of people as though they themselves were present. The presence of a coat of arms acts as a substitution for the person, even after his death(10)." The concept of heraldic devices originated out of necessity. While engaging in a large field battle it was hard to tell friends from enemies since everyone was wearing similar armor. Trying to see a specific face or body through the milling hordes around you became a formidable task. Heraldic devices solved this problem by allowing each group of fighters on the battle field to wear a distinctive pattern on their shield, or elsewhere on their person, as a means of identification. As an increasing number of people created their own devices, a system to oversee these devices became necessary.

As this need for a formal language grew, three major standards emerged; German, French, and English. The English College of Heralds was created by King Richard III in 1484. It had total control over the heraldic devices used in all areas under England’s jurisdiction (College). In France and Germany similar systems developed, each allowing slightly different elements. For example, German heraldry allows red on black( i.e. a red object on top of a black one) which neither English nor French heraldry allows. Even with three languages that have some degree differences, heraldry’s common purpose of allowing ease in translating from a blazon (i.e., written description) to a coat of arms (i.e. useful representation) remained consistent. An English knight could be identified by a French lord
based on the design on his shield, even if that shield could not have been registered by France’s rules.

Since heraldry became prevalent it has passed through three major reforms. The first is the Heraldry of Warfare period. This is the period during the 12th and 13th centuries when heraldry was used for necessary identification purposes on the battlefield. The second is the Tournament Heraldry of the 14th and 15th centuries. This period developed because, after the invention of gunpowder, heraldry became superfluous on the battlefield. Heraldry had been developed to make suits of full plate armor more unique, as gunpowder rendered the full plate armor useless, heraldry died out. Heraldry was, however, still used on the tournament fields. Ornamental Heraldry is the last, and current, period of heraldry. In this period, the use of heraldry for identification purposes is almost extinct, but it is still sometimes used as a status symbol (Heim 11). Heraldry can also be seen today on national and state flags and seals. Each country in the world has their own unique national flag to represent them, and many have their own seal that is in effect the country’s coat of arms.

Compared to both programming and natural languages, heraldic languages have unique properties. They are pseudo-natural languages which have grown and developed over time like other natural languages. However, they differ from other natural languages because they were explicitly created for the purpose of describing heraldic devices. This creation of a language for a specific purpose, rather than the usual slow evolution undergone by natural languages, produced a strong set of grammar rules which are more structured than those of most natural languages. A heraldic language then, is a context sensitive language with considerable structure and little ambiguity.
When beginning this project, I decided to study only one of the heraldic languages, as they all served the same purpose. I chose the English heraldic language for two main reasons. One motivation for choosing this specific heraldic language to work with is that the Society for Creative Anachronism (SCA), an international re-creation organization of the middle ages, uses this standard to judge the validity of arms registered within the SCA. It is my desire that this project be useful as well as academic and it is within the SCA that I envision its usefulness. Another reason I chose this version is that although the English heraldic language would not be recognized by many people as English, it developed in England and is therefore written about in English, the language I speak. Henceforth the term, "the heraldic language" will refer to the English heraldic language.
THE LANGUAGE AND THIS PROJECT

Although the heraldic language is a natural language, it has strict grammar rules and semantics which are uncommon in a natural language. It is the uniqueness of the heraldic language which enabled me to consider this a feasible project. Most of the heraldic grammar rules have not changed much, if at all, since the original creation of a formal grammar style, a fact which significantly contributes to the parse-ability of the language. One reason the rules did not change much is that the language as a whole had a relatively short time to evolve. Heraldry became mostly obsolete for identification purposes by the late 18th century when the English College of Heralds was abolished as an executive institution (Boutell 307). Although a College of Heralds still exists in England to officiate over official government functions, it no longer officially controls the usage of arms.

Another important reason for heraldry's stability is that the rules were carefully considered before being set into place. They clearly met all of the requirements which the language was addressing, and they fit into a God and King centered world view very well. In addition to this, the heraldic language is mainly a written language. Written languages are often called frozen because they don't change much over time. Once someone registered their arms, the written description of their arms was “frozen” in the record books. Years later, if anyone wanted to know what those arms looked like, they could look it up and find the grammar and spelling of the blazon.

A blazon is the name given to the written description of a heraldic picture. The blazon consists of three main elements; tinctures, charges, and modifiers. In heraldry, colors are called tinctures. Each tincture has its own heraldic name. Black, for example, becomes
“sable”, while silver becomes “argent.” Charges are the actual pictures or design elements that are put on a shield. In a blazon, the color of a charge comes after the charge to which it is referring. Modifiers are the last of the basic elements that are found in a blazon. Modifiers are any of the number of words that are used to describe or detail how a charge looks or is placed. Modifiers can change the location of a charge or change how the charge looks.

(Figure 5)

Currently, the heraldic language can be loosely broken down into a few basic grammar rules: 1) blazons are almost completely unpunctuated. 2) the only exceptions are that tinctures and charges begin with capital letters. 3) when referring to any one charge, any adjectives that modify the charge follow it, with numeric modifiers coming before the charge. 4) some charges have assumed modifiers as default. 5) tincture is the last item listed for any given charge and it is optional if the next charge listed is the same color. 6) the background of a device is always blazoned first, followed by the principal charge, secondary charge, and then any charges that overlap either the principal or secondary charges in order of position (Frial).

(figure 6)

When parsing a blazon of a Heraldic device to see if it is valid, color conflicts are one of the most difficult elements to check without experience. When heraldry was most widespread, there were a number of color rules that were used to create a device. These rules were in place to make certain that from a distance the device could be identified. These rules involved the placement of the “metals”, the light yellow(gold) and white(silver), and the “colors,” the much darker blue, green, purple, red, and black. A third variation, furs, which consist of patterns on a solid color, were also commonly used. Placing small patches of color
adjacent to or on top of another color or doing the same with metals created visual problems at a distance which is why these combinations were disallowed. In order to check for this type of conflict, the computer would have to keep track of every charge that was described, and also its relationship to all other background colors and charges that it rests on.
DESCRIPTIONS OF BASIC ELEMENTS TO BE USED IN THIS PROJECT

Although the heraldic language is a natural language, it has many aspects that are typically found in a programming language. For example, it has rules that limit semantics. With this in mind, I decided to attempt to create a computer program that would parse a heraldic blazon as input. Although I believe that parsing the whole language will eventually be possible, in order to create a reasonable scope for my thesis I decided to concentrate on one subset of the heraldic language.

The subset of the heraldic language which I chose to use is fairly straightforward. It consists of charges, colors, and some lines of division. I feel that restricting my working set to these elements has allowed me to set up an acceptable and fairly comprehensive language, in which the grammar is easy to read and understand. Some of the basic blazon formats I included are: a background with one or more charges on it, a background with a line of division, and a background with a line of division and a charge, or charges, on either side of it, or on it.

When determining what subset of the language to use, I decided to exclude from my grammar any of the basic modifiers of a charge that affect the way the charge itself graphically would look. These are cosmetic changes and could be a fairly simple addition to the code at some point in the future. I decided to leave charge modifiers out because they would clutter up the language making it more difficult for someone who wasn't completely familiar with both heraldry and programming languages to understand. Additionally, I did not think modifiers would add significantly to the project.

Another major element that I did not include is alternative descriptive naming. Like
most natural languages, the Heraldic language has a number of different ways to say the same thing. For example, in English you can say that the glass is half empty or the glass is half full, the sky is partly cloudy or partly sunny. I chose not to accept these because, although it would be possible to have conditions for these built into my program, I feel at this time that it would merely add unnecessary complexity. I think that the handling of metaphors would be better implemented as a front end program to pre-scan the input for common metaphors and replace them with their recognized equivalents.

Another aspect of heraldry that I did not include in my project is charges placed in a specific design, a line, for example. In American English, we say that people stand in a line. This refers to the position of a group of people. Heraldry also allows this type of special reference which I chose not to use in order to keep my code as simple as possible and to facilitate the ability of others to understand what I was doing and how I managed to do it, as well as to allow me to complete the project in the allotted time.

My choice to work solely with a subset of the heraldic language allowed me to concentrate effort on the core essences of the heraldic language. I chose the most basic concepts of the language to use, those concepts that have the most static grammar. I believed that trying to interpret the entire heraldic language at once would be a monstrous task, but that by breaking down the language I would be able to create the code to parse a small part of the grammar. My goal was to parse a subset and slowly grow from there. Reduction of the valid heraldic language grammar set reduced the complexity of the code that I needed to parse, without reducing the significance of the work that I was doing. I believe that with minimal work, any individual concept of the language which I chose to ignore can later be
added to my grammar.

In order to accomplish the parsing of the chosen subset of the heraldic language, I looked over many examples of the heraldic language and determined a basic set of rules that generally held true. I compared this set to the blazoning rules which I found in heraldic resources. The grammar that I created after accomplishing this accepts the charges that it has been told are valid charges, and most modifiers of charges that affect the physical location of the charge on the blazon.

The basic format that the grammar accepts is a color followed by one or more charges. For my subset the background color must be a solid color and the charges may be any of the defined charges or lines of divisions. A charge is in the format of a number if there is more than one, or an “an” or an “a” followed by the name of the charge. After the name of the charge comes any location information, followed by charge modifiers. The only exception that I used in my grammar is the modifier “between”. In this case it is the format of a charge between charge and charge. (figures 7 and 8)

After the grammar was finalized for my subset, I created a program that takes in a line of blazon and scans it for valid words. All words in the blazon must be recognized by the program in order for it to be considered a possible valid blazon. Then the blazon, along with the tokens which describe what type of linguistic element each word in the blazon is, is passed to the parsing part of the program.

In the parsing part of the program, I attempted to reduce the valid grammar to a limited number of functions. The inputted grammar enters into these loops, its direction being defined by the tokens attached to the various words of the input.
PROBLEMS FACED WHILE CODING

One of the major problems that I faced in this project was to try to develop a grammar for both the heraldic language as a whole, and for my subset. Because I was not well acquainted with the specifics of the language, I chose to use a two step approach in forming these grammars. First, I researched the heraldic language in order to develop a general idea of its format and fundamental rules. Once I felt that I understood the general fundamentals of the heraldic language, I looked at examples from the SCA armorial to search for patterns to back up these grammar rules.

After examining these patterns, I discovered a problem with the descriptions I had received concerning the fundamentals of the Heraldic language. Traditionally there should not be any punctuation, such as commas, in a blazon. Even so, I found that a comma was often placed in a blazon to make it more readable. Commas were also sometimes used to separate different blazons or groups of blazons. While much of the comma usage was inconsistent throughout the sample, the usage of a comma to separate the background blazon from that of the charges to be placed on it was consistent. I felt that this usage appeared to be a standard useful enough to include in my grammar.

One of the next problems that I encountered, and subsequently thought that I had successfully solved, was deciding which programming language I should use. I considered using either C++ or JAVA and my first choice became JAVA. One of the great selling points of JAVA has always been the fact that it does not have pointers. Although I, unlike many programmers, like pointers, I thought it would be nice not to have to worry about the hassle of keeping track of them in JAVA. Eventually though, I decided that JAVA's references are
worse than C++'s pointers.

Realizing that I had misunderstood JAVA's references was one of several frustrations I encountered while programming due to my misinterpretation of how JAVA worked. I was able to track down most of the problems and solve some of them. Unfortunately, I was unable to solve all of them in time to finish this project in JAVA.

After spending some of fall quarter and an intense J-term coding in JAVA, I still did not have a working program. As I started spring semester, I came to the conclusion that it would not be worth my while to continue struggling with the programming and run the risk of not finishing my paper in time. To this end, I decided to stop writing the JAVA and instead to push to get the paper done. My hope was that if writing went well I would have time to work on re-coding the project in C++ and develop a working project by the conclusion of my thesis.

When I started re-coding in early April, I encountered remnants of problems I had faced in JAVA, which continued to affect my program. I first tried to search out all the JAVA specific code and replace it with the C++ equivalent in order to reuse the code I had completed. This was not as hard as I first thought it would be, but at times, the fact that I had just finished programming in JAVA created problems. Switching from one language to another became frustrating when I found that it was difficult to remember how to do simple tasks in C++ without review.

The problem of reference values/pointers was present in the programming of both JAVA and C++. In JAVA, when a dynamically allocated memory variable is declared, a reference value is assigned to it referring to the memory address where the variable is located. Because
of my belief that JAVA did not use pointers, I did not worry about them. I believed they would be correctly taken care of. I was wrong. When I tried to copy a variable's value to another variable, JAVA copied the reference value instead of the actual value. I tried to work around the vagueness of the documentation but was unable to find a successful way to have the reference value/reference location identities work for me the way that I wanted. It seemed unclear when a variable was being passed by a reference to its location rather than passed by value, and this caused problems when trying to assign values in arrays.

The attempt to address this problem using pointers in C++ after most of the code was converted was more successful than my attempt to use reference values in JAVA. This can be attributed to my familiarity with and knowledge of exactly how pointers work in C++. After multiple attempts at implementing pointers, I was able to develop a scheme that worked well for me and the project.

Learning how to dynamically declare and then define non-square multi-dimensional arrays in JAVA was another challenge. This problem was frustrating because of the lack of available information. When the books that I used mentioned non-square multi-dimensional arrays, they only briefly touched on them with examples using static declarations. In spite of the lack of information, I was eager to try to use multi-dimensional arrays because of the large advantage they have over square multi-dimensional arrays in that they only use as much memory as is needed. I wanted to use them to store all of the valid words of my grammar along with their respected tokens in a manner that would allow a relatively quick access time.

I talked to a number of computer aficionados who informed me that it was possible to create non-square multi-dimensional arrays, but none of them knew, off hand, how to
accomplish the task. I decided to experiment on my own and, after much trial and error, I was able to figure out how to declare non-square multi-dimensional arrays. I was then able to use these arrays in both the JAVA and C++ version to easily pass information to the different parts of the program.

Unfortunately, I was unable to find a decent method to break apart either a string into substrings or a character array into substrings of arrays using JAVA. In all of my research I found only the most basic functions for this task. In order to create substrings I was required to look at each letter in the array of characters and add them together to form words that made up strings which I then was able to place in an array.

The final, and never ending, problem that I faced while programing was how to create the flow control loops on the program that actually take in the input and then extract the resulting charges and information. This becomes a never ending process because until the whole language is converted into a grammar that then is subsequently converted into code, there will always be more work that will need to be done with it. My major problem with doing this in C++ for the subset of the heraldic language that I chose stemmed from the fact that I transferred much of my code from the JAVA coding attempt. I had to remove a number of unnecessary variables and functions that were included in the code to work around problems in the JAVA program that now functioned in just the opposite manner.
CONCLUSION

Despite the problems I faced while programming, I believe that this project was a wonderfully successful learning experience. I attained my goal of creating a program that successfully parsed a subset of the heraldic language. In doing so I learned about the process involved in hard coding natural grammar rules in an algorithm. I learned that creating a grammar which is sufficiently restricted to begin coding is a complex job, and that it may be benafitual to work on coding a language in stages. Although the heraldic language is a pseudo-natural language, it offers some insight into one of the potential processes of parsing a natural language. This method of hard coding the grammar rules explicitly into the program may also work with true natural languages.

In order to make this project feasible, from the start I had to limit my grammar to a small subset, which included only some of the basic elements of heraldry. When I decided to re-code my program in C++, I further reduced the grammar to some of the truly fundamental elements in order to produce a working program by my deadline. These basic elements included a background and a background with a charge. Once the basic program was working, I was able to slowly add more grammar. The grammar that I added was originally part of the subset of the heraldic language I had chosen to incorporate in the JAVA version. I first added the concept of elements being "between" other elements, which involved keeping track of where on the shield the elements were. Once that was completed, I was able to add the concept of having a charge "on" another charge along with some of the positional modifiers.

In this manner I was able to begin with a fundamental subset and begin adding more
complex subsets to the program as I progressed. I believe that this supports my theory that it is not necessary, or desirable, to tackle an entire language in one chunk. Although my program parses a smaller subset than the one that I initially visualized, the validity of using a subset remains for the same reasons. The subset may be small, but it is a strong foundation to which additional information can be added, as I have been doing, without exceptional effort. I feel it would be possible to continue adding subsets to the program until the language is complete, or at least nearly so.

My decision to switch from JAVA to C++ in my attempt to program was a wise decision. Although my JAVA program had seemed to reach a dead end, I was able to take it the next step in C++. After switching languages, I was able to complete a C++ program that takes in a limited subset of the heraldic language and breaks it down into its components. The program takes in a blazon, and then outputs to the screen a block of text for each charge that it finds in the input along with the different attributes that deal with that charge. This not only shows that it is possible to parse a subset of the heraldic language, it provides a useful analysis of a blazon containing the accepted elements.

The fact that I am dealing with a sub-set of the heraldic language instead of the whole language is one flaw in the work that I have done. The relatively small size of the sub-set of the heraldic language that I ended up using does not provide a complete picture of the language. Because I did include the entire language it is not completely certain that it would be possible to do anymore than I have already accomplished. However, as I have already started adding subsets in the manner in which other subsets will be added, I find this an unlikely possibility.
This flaw is one that was built in to my proposal since in spite of it I reached my goal by completing a program that is able to parse a subset of the heraldic language, as I had set out to do. I feel that this was successful because most of the items that I chose to exclude can easily be added to my program, either as a pre-parser, or within the parser code itself. I feel that the subset of the heraldic language that I chose was not overly difficult or hard in and of itself. I have come to the conclusion that starting with an even smaller subset and building upon it toward the final grammar would be the best method to use to write this program.

This project has started to reach my goal of being useful to end users, although this aspect should be further improved. I believe it is also useful to other computer scientists in multiple ways. Those studying natural language parsing may simply be interested in seeing how I attempted this project and how my finished program was coded. Additionally, a benefit to any of the computer programmers within the SCA who may wish to take this project farther than I, would be the ability to see not only what worked in C++ but also what I attempted in JAVA, as well as how and what I transferred from JAVA to C++. This could especially be useful to anyone who wishes to try creating this program in a language other than the C++ which I plan to continue pursuing as stated below.

If given the chance to do this over, I would have started to program immediately in C++ instead of attempting to use JAVA. One reason is that after giving up on JAVA many of the problems that I faced when programming in C++ were due to the time constraints that I had based on the late start date of the C++ programming. I believe that the majority of the problems I faced dealing with Java were due to the basic programming language I do not believe that it was lack of skill in the JAVA language that caused problems. I did run into
some problems based on my C++ assumptions, but most of them were easily fixed by looking in SUN's Core Java book, or talking with my fellow classmates and professors. The language itself created many of the problems because the main use of JAVA has been to add thrills to web pages and most of the functionality that has been refined in it leans toward these goals. In contrast to the relative simplicity of adding thrills, I routinely tried to perform complicated recursion and repetition to solve problems with large multi-dimensional arrays of information that were constantly changing. Although my attempts here were unsuccessful, I believe that JAVA is capable of handling this if enough time and energy is spent doing so. I do not, however, think that JAVA is an efficient means of doing so.

While writing the program, one of my major priorities was to create code that would be relatively easy for myself, or someone else, to expand and improve on. The list of valid key words and the tokens that go along with them are read from a file, so it is relatively easy for them to be added to, changed, or updated at any time. The rest of the program was created using an object oriented design approach so that it would be evident what each part of the program did as each part interacted with the others. I also tried to keep the variable names that I used for the same elements consistent throughout the program to further ease the understanding of the actual code.

Although my sub-set does not include all of the grammar rules that define the heraldic language, I feel that through the use of the subset, it is possible to show how one would go about parsing the heraldic language as a whole, one new rule at a time, starting with the most basic rules. I believe that another programmer, familiar with C++ and heraldry, could pick up this project where I left off and continue with it. In order to add the rest of the language, the
programmer would need to list all the grammar rules and create a hierarchy list of sub-sets of the language where each subset contains all the subsets below it in the chain. Then they would then start with the smallest subset, and work up the list of subsets adding the new grammar rules into the algorithm.

I plan to continue this project in my free time this summer. I feel that it is a worthwhile and achievable goal. The graphical display of the elements of the blazon is also something that is on my list of things to accomplish. I did not worry about that aspect of the program for this project because I was interested in the methods of parsing the heraldic language rather the various methods of graphically display data, but I feel that this is something that would be worth looking into.

Overall, I feel that this project was successful. I achieved the goals that I had set including my goals to complete a working program and learn more about natural languages and the heraldic language in particular. While coding the program to parse the heraldic language, I gained much information about the makeup of natural languages, what some of the methods that are currently used to try and parse them are, and how hard coding a natural language works.
The total language tree of the grammar:

- **S**: $\rightarrow$ ACBC
- **A**: $\rightarrow$ a
- **B**: $\rightarrow$ b
- **C**: $\rightarrow$ A | B

Resulting in the strings of:

- aaba
- aabb
- abba
- abbb
The parse trees for the grammar:

\[ S \rightarrow ACBC \]
\[ A \rightarrow a \]
\[ B \rightarrow b \]
\[ C \rightarrow A \mid B \]

Resulting in the string of: \textit{aaba}

Resulting in the string of: \textit{aabb}

Resulting in the string of: \textit{abba}

Resulting in the string of: \textit{abbb}
The total language tree of the grammar:

- $S \rightarrow ACBC$
- $A \rightarrow a$
- $B \rightarrow b$
- $aa \rightarrow C$
- $C \rightarrow A \mid B$

Resulting in the strings of:

- $aaba$, $aabb$, $abba$, $abbb$, $aba$, $bba$, $abb$, $bbb$. 
The total language tree of the grammar:

\[
\begin{align*}
S & \rightarrow AB \\
A & \rightarrow a \\
B & \rightarrow b \\
B & \rightarrow bbb \\
bb & \rightarrow C \\
C & \rightarrow b
\end{align*}
\]

Resulting in the strings of: \textit{ab, abbb, abb}.
Notice that there is three ways to create the word \textit{ab} and two ways to create \textit{abb}.
In the world of a map there are four cardinal directions: north, east, south, and west. Similarly, the world of a shield also has four cardinal directions.

is blazoned "Sable, a mullet argent."

The first word, "Sable" describes the field, indicating that it is entirely black. The remainder of the blazon, "a mullet argent" describes a group of charges on the field: 1."a" indicates that there is one charge in the group 2."mullet" indicates that charges in the group are stars 3."argent" indicates that the tincture of the group is silver/white.
Two examples of blazons from actual arms.
The grammar of the subset of the heraldic language I used

S -> E | EK
A -> and
B -> JCE
C -> name of a charge
D -> on
E -> name of a color | nothing
F -> in
G -> fess | cheif
H -> to
I -> dexter | sinister
J -> FG | HI | nothing
K -> B | BAB | BDB
JAVA CODE

import java.io.*;
import GetInput;
import Parse;
import Valadate;
/**
 * Test.java
 * @author Hans Mersinger
 * @version Pre-Beta
**/

public class test {
    public static void main(String argv[]) throws IOException{
        GetInput.GetInput();
        String[][] TEST = GetInput.SENTENCE;
        System.out.println(TEST[0][0]+"+TEST[1][0]+"+TEST[2][0]);
        int LENGTH = GetInput.Length();
        for (int i = 0; i < LENGTH; i++){
            TEST[i][1]=Parse.Parse(TEST[i][0]);
        }
        GetInput.Print();
        Valadate.Valadate();
        Valadate.Print();
    }
}

VAladate.java

import java.io.*;
import Element;
/**
 * Valadate.java
 * @author Hans Mersinger
 * @version Pre-Beta
**/
public class Validate {
    // array.getLength() should return the length of the array
    static Element[] TEMPBLAZEN = new Element[50];       
    //array of all the charges and colors
    there of.
    static Element[] BLAZEN;     //Actual blazen to be used
    static int BL=0;             //CURRENT length of the TEMPBLAZEN
    array
    static int LENGTH;          //number of elements in the input
    static String[][] INPUT;    //the array of strings of the input
    static String[][] REST;     //temp array. Stores the TEMPBLAZEN that has
    not been parsed
    static int TEMP=0;          //yet. TMP is the number of element
    in REST
    static int CURRENT=0;       //CURRENT location in REST
    static int LEFT=0,RIGHT=100, TOP=100,BOTTOM=0;
    public static void Validate(){
        for (int i = 0; i < 50; i ++){
            TEMPBLAZEN[i] = new Element();
        }
        LENGTH = GetInput.Length();
        INPUT = GetInput.SENTENCE;
        ParseColor();
        while (TEMP > 0){
            ParseRest();
        }
        BLAZEN = new Element[BL+1];
        for (int i = 0; i <= BL; i ++){
            BLAZEN[i] = TEMPBLAZEN[i];
        }
    }
    private static void ParseColor(){
        if (INPUT[0][1].equals("color")){
            TEMPBLAZEN[0].SetColor(IntColor(INPUT[0][0]));
            TEMPBLAZEN[0].SwitchColor();
            TEMPBLAZEN[0].SetName("background");
            if (INPUT[1][1].equals("comma")){
                TEMP = LENGTH - 2;
                System.out.println(TEMP+" + LENGTH");
                System.out.flush();
                REST = new String[TEMP+1][2];
                for (int i = 0; i < TEMP; i ++){
                    REST[i][0] = "";
                    REST[i][1] = new String[0];
                    REST[i][1][0] = " ";
                }
            }
        }
    }
}
REST[i][0]=INPUT[i+2][0];
REST[i][1]=INPUT[i+2][1];
}
REST[TEMP][0]="";
REST[TEMP][1]="";
}
else { // the TEMPBLAZEN does not start with a color
    System.out.println("Error, no color");
}

private static int IntColor(String Color)
{
    if (Color.equals("sable")){
        return 1;
    }
    else if (Color.equals("gules")){
        return 2;
    }
    else if (Color.equals("azure")) {
        return 3;
    }
    else if (Color.equals("vert")) {
        return 4;
    }
    else if (Color.equals("purpure")) {
        return 5;
    }
    else if (Color.equals("argent")) {
        return 6;
    }
    else if (Color.equals("or")) {
        return 7;
    }
    else {
        return 0;
    }
}

private static void ParseRest()
{
    System.out.print("ParseRest");
    BL++;
    if (REST[CURR][1].equals("in")){
        System.out.print("in");
        CURRENT++;
        ParseLocation();
    }
}
}
if (REST[CURRENT][1].equals("number")) {
    System.out.print(" number");
    CURRENT++;
    ParseCharge();
    if (REST[CURRENT][1].equals("on")) {
        System.out.print(" on");
        CURRENT++;
        ParseRest();
    }
    if (REST[CURRENT][1].equals("between")) {
        System.out.print(" between");
        int t = TOP;
        int b = BOTTOM;
        int l = LEFT;
        int r = RIGHT;
        TEMPBLAZEN[BL].SetX(((RIGHT-LEFT)/2)+LEFT);
        TEMPBLAZEN[BL].SetY(((TOP-BOTTOM)/2)+BOTTOM);
        TEMPBLAZEN[BL].SetSize(TOP-BOTTOM);
        CURRENT++;
        Split(3,1,t,b,l,r);
        ParseRest();
        if (REST[CURRENT][1].equals("and")) {
            System.out.print(" and");
            CURRENT++;
            Split(3,3,t,b,l,r);
            ParseRest();
        }
        else { //error, something is not right
            System.out.println("got lost looking for the and");
        }
    }
    Split(1,1,t,b,l,r);
}
else {
    TEMPBLAZEN[BL].SetX(((RIGHT-LEFT)/2)+LEFT);
    TEMPBLAZEN[BL].SetY(((TOP-BOTTOM)/2)+BOTTOM);
    TEMPBLAZEN[BL].SetSize(TOP-BOTTOM);
}
}

TEMP = TEMP - CURRENT;
String[][] TEMPS = new String[TEMP+1][2];
for (int i = 0; i <= (TEMP); i++) {
    TEMPS[i][0] = REST[i+CURRENT][0];
    TEMPS[i][1] = REST[i+CURRENT][1];
}
```java
TEMPS[i][1]=REST[i+CURRENT][1];
}
REST = TEMPS;
CURRENT = 0;
}
private static void ParseCharge()
{
    System.out.println(" ParseCharge");
    if (REST[CURRENT][1].equals("charge")]
        TEMPBLAZEN[BL].SwitchColor();
        TEMPBLAZEN[BL].SetName(REST[CURRENT][0]);
        CURRENT++;
        if (REST[CURRENT][1].equals("modifier")]
            System.out.print(" modifier");
            SetModifier();
        }
    if (REST[CURRENT][1].equals("color")]
        System.out.print(" color");
        for (int i = BL; i > 0; i--)
        {
            SetColor(IColor(REST[CURRENT][0],i);
        }
        CURRENT++;
    }
}
else {
    ERROR();
}

private static void ParseLocation()
{
    System.out.println(" ParseLocation");
/*
 * if (CURRENT >= TEMP){}
 * else {
 * if (REST[CURRENT][0].equals("dexter")]
 *     TEMPBLAZEN[BL].SwitchDexter();
 * }
 * else if (REST[CURRENT][0].equals("sinister")]
 *     TEMPBLAZEN[BL].SwitchSinister();
 * }
 * else if (REST[CURRENT][0].equals("chief")]
 *     TEMPBLAZEN[BL].SwitchChief();
 * }
 * else if (REST[CURRENT][0].equals("fess")]
 *     // it defaults to the center
 * else if (REST[CURRENT][0].equals("base")]
 * ```
TEMPLAZEN[BL].SwitchBase();
}
else {
    CURRENT--;
}
*/
CURRENT++;
}
private static void SetColor(int c, int i){
    System.out.println(c + " " + i);
    if (TEMPLAZEN[BL].Color(i)) {
        TEMPLAZEN[i].SetColor(c);
        TEMPLAZEN[i].SwitchColor();
    }
    System.out.println(c + " " + i);
}
private static void SetModifier()
    System.out.print(" " SetModifier");
/*
    if (REST[CURRENT][1].equals("inverted")) {
        TEMPLAZEN[BL].SwitchInverted();
    } else if (REST[CURRENT][1].equals("reversed")) {
        TEMPLAZEN[BL].SwitchReversed();
    } else {
        CURRENT--;
    }
*/
CURRENT++;
}
private static void ERROR(){
private static void Split(int n, int p, int t, int b, int l, int r) {
    // (number of splits, what split we are at, top, bottom,
    left, right)
    TOP = t;
    BOTTOM = b;
    LEFT = (l + ((p - l) * (t - l) / n));
    RIGHT = (l + (p * (t - l) / n));
}
public static void Print() {
    System.out.println(BL);
    for (int i = 0; i <= BL; i++) {
```java
System.out.println(i);
BLAZEN[i].Print();
}
}
```

```java
import java.io.*;

/**
 * Parse.java
 * @author Hans Mersinger
 * @version Pre-Beta
 **/

public class Parse {
static int NUM = 52; // total number in the array
public static String[][] DATA; // [categories][data][type]
    // [x][y] where
    // x[1-51] (odd)=alpha listing;
    // x[2-52] (even) = token ids for the corresponding (x-1) y's
    // y is either the key word, or the corresponding token;

public static String Parse(String VALUE){
    int TRY = -1; // the search path to be tried
    int FIRST = Character.getNumericValue(VALUE.charAt(0));
    if (((FIRST > 9) && (FIRST <= 36)) {  
        TRY = (((FIRST - 9)*2)-1);
        int T = Integer.parseInt(DATA[TRY][0]);
        for (int i = 1; i <= T; i++){
            if (VALUE.equals(DATA[TRY][i])){
                return DATA[TRY+1][i];
            }
        }
        return "ERROR";
    } else if (((FIRST < 10) && (FIRST >=0)){
        return "number";
    } else if (VALUE.equals(",\")){
        return "error\";
    } else {  
        TRY = DATA[0][0];
        return TRY;
    }
}
```
return "comma";
}
return "ERROR";
}

static {
    DATA = new String[53][];
    String WORD = null;
    File FILE = new File("data.dat"); // creates the file object
    if (!FILE.exists()){
        System.out.println("File 'data.dat' could not be
found. please make sure
that it is in the current directory");
    }
    if (!FILE.canRead()){
        System.out.println("File 'data.dat' could not be
read. please make sure
that it is in the current directory");
    }
    BufferedReader in = null;
    try {
        in = new BufferedReader(new FileReader(FILE));
    } catch(Exception e){}
    for (int i = 1; i < 53; i++){
        try {
            WORD = in.readLine();
        } catch(Exception e){
            System.out.println("Opps, couldn't read line
at spot 1");
        }
        int VALUE = Integer.parseInt(WORD);
        DATA[i] = new String[VALUE+1];
        DATA[i][0] = WORD;
        for (int ii = 1; ii <= VALUE; ii++){
            try {
                DATA[i][ii] = in.readLine();
            } catch(Exception e){
                System.out.println("Opps, couldn't
read line at spot 2");
            }
        }
    }
    i++;
DATA[i] = new String[VALUE+1];
DATA[i][0] = WORD;
for (int ii = 1; ii <= VALUE; ii++){
    try {
        DATA[i][ii] = in.readLine();
    }
    catch(Exception e){
        System.out.println("Opps, couldn’t read line at spot 3");
    }
}

GetInput.java

import java.io.*;
import Parse;

/**
 * GetInput.java
 * @author Hans Mersinger
 * @version Pre-Beta
 **/

public class GetInput {
    private int LENGTH;
    //private String[] TEMP;
    public static String SENTENCE[][];
    private static String NULL = " ";
    private static int COUNT;
    private static int END;
    private static int START;//will hold the starting place of a word
    public static void GetInput() throws IOException {
        // Get set up to read lines of text from the user
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Blazen: "); // Print a prompt
        String line = in.readLine()+" "; // Read a line
        if (line.equals("")) { // If empty line then throw exception
            throw new IOException("ERROR Nothing entered");
        }
    }
}
int LENGTH = (line.length()-1);
String[] TEMP= new String[LENGTH];
int COUNTING = -1; // number of words, starting at zero
for (END = 0; END <= LENGTH; END++) {
    if (line.charAt(END)==(" ")){
        if (START == END){
            START=END+1;
        } else {
            COUNTING++;
            TEMP[COUNTING]=line.substring(START,END);
            START=END+1;
        }
    } else if (line.charAt(END)==(',')){
        if (START < END){
            COUNTING++;
            TEMP[COUNTING]=line.substring(START,END);
        } 
        COUNTING++;
        TEMP[COUNTING]=line.substring(END,END+1);
        START=END+1;
    }
}
SENTENCE = new String[COUNTING+1][2];
for (int i = 0; i <= COUNTING;i++){
    SENTENCE[i][0] = TEMP[i];
    SENTENCE[i][1] = "ERROR";
}
COUNT = COUNTING+1;
System.out.println(LENGTH+"+COUNT+"+COUNTING);
}
public static int Length(){
    return COUNT;
}
public static void Print(){
    for(int i = 0; i < COUNT;i++){
        System.out.println(SENTENCE[i][0]+""+SENTENCE[i][1]);
    }
}
import java.io.*;

/**
 * Element.java
 * @author Hans Mersinger
 * @version Pre-Beta
 **/

public class Element {
    boolean INVERTED = false; // These values are set while parsing
    boolean REVERSED = false; // the current element of the blazen
    boolean DEXTER = false; // They are set to 'true' only if the
    boolean SINISTER = false; // current element sets it.
    boolean CHEIF = false;
    boolean BASE = false;
    boolean COLOR = false;
    int MYCOLOR = 0;
        // color values are:
        // 0 = none 1 = sable (Black)
        // 2 = gules (Red) 3 = azure (Blue)
        // 4 = vert (Green) 5 = purpure (Purple)
        // 6 = argent (Silver/White) 7 = or (Gold/Yellow)
    String NAME = "NONE"; // The name of the object of the current
element.
    int X=50; // X,Y, and SIZE, keep track of the location
    of the
    int Y=50; // element, along with the size of the
    element.
    int SIZE=0; // Using a standard 100x100 grid with sizes
0-2.

    public void Element(){};
    public void Element(Element E){
        INVERTED = E.Inverted();
        REVERSED = E.Reversed();
        DEXTER = E.Dexter();
        SINISTER = E.Sinister();
        CHEIF = E.Cheif();
        BASE = E.Base();
    }
COLOR = E.Color();
MYCOLOR = E.MyColor();
NAME = new String (E.Name());
X = E.X();
Y = E.Y();
SIZE = E.Size();
}
public boolean Inverted(){
    // These calls return the set value of the
    return INVERTED; // private boolean attributes flags.
}
public boolean Reversed(){
    return REVERSED;
}
public boolean Dexter(){
    return DEXTER;
}
public boolean Sinister(){
    return SINISTER;
}
public boolean Cheif(){
    return CHEIF;
}
public boolean Base(){
    return BASE;
}
public boolean Color(){
    return COLOR;
}
public int MyColor(){
    // Returns the set color for the current object.
    return MYCOLOR;
}
public String Name(){
    // Returns the set name for the current object.
    return NAME;
}
public int X(){
    // Returns the x value for the current object.
    return X;
}
public int Y(){
    // Returns the y value for the current object.
    return Y;
public int Size(){    // Returns the size of the current object.
    return SIZE;
}

public void SwitchInverted(){    // Switches the set value to the
    INVERTED = !INVERTED;    // Other boolean value.
}

public void SwitchReversed(){
    REVERSED = !REVERSED;
}

public void SwitchDexter(){
    DEXTER = !DEXTER;
}

public void SwitchSinister(){
    SINISTER = !SINISTER;
}

public void SwitchCheif(){
    CHEIF = !CHEIF;
}

public void SwitchBase(){
    BASE = !BASE;
}

public void SwitchColor(){
    COLOR = !COLOR;
}

public void SetColor(int NEWCOLOR){    // Set the current object to
    MYCOLOR = NEWCOLOR;    // the new
    // color, NEWCOLOR.
}

public void SetName(String NEWNAME){    // Sets the name of the
    NAME = NEWNAME;    // current object
    // to NEWNAME.
}

public void SetX(int NEWX){    // Set the current object to
    X = NEWX;    // the new
    // x, NEWX.
}

public void SetY(int NEWY){    // Set the current object to
    Y = NEWY;    // the new
    // y, NEWY.
}

public void SetSize(int NEWSIZE){    // Set the current object to
    // the new

}
SIZE = NEWSIZE;            // size, NEWSIZE.
}

public void Print(){
    System.out.println("INVERTED "+INVERTED);
    System.out.println("REVERSED "+REVERSED);
    System.out.println("DEXTER "+DEXTER);
    System.out.println("SINISTER "+SINISTER);
    System.out.println("CHEIF "+CHEIF);
    System.out.println("BASE "+BASE);
    System.out.println("COLOR "+COLOR);
    System.out.println("MYCOLOR "+MYCOLOR);
    System.out.println("NAME "+NAME);
    System.out.println("X "+X);
    System.out.println("Y "+Y);
    System.out.println("SIZE "+SIZE);
}
C++ CODE

#include "GetInput.h"
#include "Element.h"
#include <iostream.h>
#include "Parse.h"
#include "Word.h"
#include "Valadate.h"

/**
 * Test.cc
 * @author Hans Mersinger
 * @version Pre-Beta
 **/

void main() {

Word * TheInput;
GetInput INPUT;
const int Length = INPUT.Length();
TheInput = new Word[Length];
INPUT.Setup(TheInput);
Parse(TheInput,Length);
Valadate(TheInput,Length);
}

Word.cc

#include "Word.h"
#include "misc/MyString.h"

/**
 * Word.cc
 * @author Hans Mersinger
 * @version Pre-Beta
 **/
Word :: Word()
    Key = "";
    Token="";
};
Word :: ~Word();

Word & Word :: operator = (const Word & source) {
    Key = source.Key;
    Token = source.Key;
    return *this;
}

="/*************************/
Word.h
="/*************************/

#include "misc/MyString.h"

#ifndef WORD_H
#define WORD_H

class Word {
public:
    Word();
    ~Word();
    MyString Key;
    MyString Token;
    // Operator : assignment from MyString
    // Precondition : source != NULL
    // Postcondition : *this[i] == source[i], 0 <= i < Length
    ()
    // Length () == source.Length ()
    // Return value : *this
    Word & operator = (const Word & source);
};
#endif

#include "Element.h"
#include "misc/MyString.h"
#include <iostream.h>
* Element.cc
* @author Hans Mersinger
* @version Pre-Beta
**/

Element :: Element(){
    INVERTED = 0;    // These values are set while parsing
    REVERSED = 0;   // the current element of the blazen
    DEXTER = 0;     // They are set to '1' only if the
    SINISTER = 0;   // current element sets it.
    CHEIF = 0;
    BASE = 0;
    COLOR = 0;
    MYCOLOR = 0;
    // color values are:
    // 0 = none           1 = sable (Black)
    // 2 = gules (Red)    3 = azure (Blue)
    // 4 = vert (Green)   5 = purpure (Purple)
    // 6 = argent (Silver/White) 7 = or (Gold/Yellow)
    NAME = "NONE";       // The name of the current element.
    X=0;                 // X,Y, and SIZE, keep track of the location
    Y=0;                 // element, along with the size of the
    SIZE=0;              // Using a standard 100x100 grid with sizes
    0-2.
}

Element :: ~Element(){
void Element :: copyElement(Element E){
    INVERTED = E.Inverted();
    REVERSED = E.Reversed();
    DEXTER = E.Dexter();
    SINISTER = E.Sinister();
    CHEIF = E.Cheif();
    BASE = E.Base();
    COLOR = E.Color();
    MYCOLOR = E.MyColor();
    NAME = E.Name();
    X = E.Xvalue();
    Y = E.Yvalue();
    SIZE = E.Size();
}
int Element :: Inverted() { // These calls return the set value
    return INVERTED; // private inten attributes flags.
}
int Element :: Reversed(){
    return REVERSED;
}
int Element :: Dexter(){
    return DEXTER;
}
int Element :: Sinister(){
    return SINISTER;
}
int Element :: Cheif(){
    return CHEIF;
}
int Element :: Base(){
    return BASE;
}
int Element :: Color(){
    return COLOR;
}
int Element :: MyColor(){ // Returns the set color for the
current object.
    return MYCOLOR;
}
MyString Element :: Name(){ // Returns the set name for the
current object.
    return NAME;
}
int Element :: Xvalue(){ // Returns the x value for the
current object.
    return X;
}
int Element :: Yvalue(){ // Returns the y value for the
current object.
    return Y;
}
int Element :: Size(){ // Returns the size of the current
object.
    return SIZE;
}

void Element :: SwitchInverted(){ // Switches the set value to
the

INVERTED = !INVERTED; // Other int value.
}

void Element :: SwitchReversed()
    REVERSED = !REVERSED;
}

void Element :: SwitchDexter()
    DEXTER = !DEXTER;
}

void Element :: SwitchSinister()
    SINISTER = !SINISTER;
}

void Element :: SwitchChief()
    CHEIF = !CHEIF;
}

void Element :: SwitchBase()
    BASE = !BASE;
}

void Element :: SwitchColor()
    COLOR = !COLOR;
}

void Element :: SetColor(int NEWWCOLOR) { // Set the current
    MYCOLOR = NEWWCOLOR; // the new color,
    NEWWCOLOR.
}

void Element :: SetName(MyString NEWWNAME) { // Sets the name of
    NAME = NEWWNAME; // current to
    NEWWNAME.
}

void Element :: SetX(int NEWX) { // Set the current
    X = NEWX; // to the x, NEWX.
}

void Element :: SetY(int NEWY) { // Set the current
    Y = NEWY; // to the y, NEWY.
}

void Element :: SetSize(int NEWSIZE) { // Set the current
    SIZE = NEWSIZE; // the new size,
    NEWSIZE.
}
void Element :: Print()
{
    cout << "INVERTED" << INVERTED << "\n";
    cout << "REVERSED" << REVERSED << "\n";
    cout << "DEXTER" << DEXTER << endl;
    cout << "SINISTER" << SINISTER << "\n";
    cout << "CHEIF" << CHEIF << "\n";
    cout << "BASE" << BASE << endl;
    cout << "COLOR" << COLOR << "\n";
    cout << "MYCOLOR" << MYCOLOR << "\n";
    cout << "NAME" << NAME << endl;
    cout << "X" << X << "\n";
    cout << "Y" << Y << "\n";
    cout << "SIZE" << SIZE << endl << endl;
}

void Element :: Copy(Element E){
    INVERTED = E.Inverted();
    REVERSED = E.Reversed();
    DEXTER = E.Dexter();
    SINISTER = E.Sinister();
    CHEIF = E.Cheif();
    BASE = E.Base();
    COLOR = E.Color();
    MYCOLOR = E.MyColor();
    NAME = E.Name();
    X = E.Xvalue();
    Y = E.Yvalue();
    SIZE = E.Size();
}

#include "misc/MyString.h"

#ifndef ELEMENT_H
#define ELEMENT_H

class Element {
public:
    Element();
    ~Element();

```
void copyElement(Element E);
int Inverted();
int Reversed();
int Dexter();
int Sinister();
int Cheif();
int Base();
int Color();
int MyColor();
MyString Name();
int Xvalue();
int Yvalue();
int Size();
void SwitchInverted();
void SwitchReversed();
void SwitchDexter();
void SwitchSinister();
void SwitchCheif();
void SwitchBase();
void SwitchColor();
void SetColor(int NEWCOLOR);
void SetName(MyString NEWNAME);
void SetX(int NEWX);
void SetY(int NEWY);
void SetSize(int NEWSIZE);
void Print();
void Copy(Element E);

private:

int INVERTED;    // These values are set while parsing
int REVERSED;    // the current element of the blazen
int DEXTER;      // They are set to '1' only if the
int SINISTER;    // current element sets it.
int CHEIF;
int BASE;
int COLOR;
int MYCOLOR;
    // color values are:
    // 0 = none                     1 = sable (Black)
    // 2 = guules (Red)              3 = azure (Blue)
    // 4 = vert (Green)              5 = purpure (Purple)
    // 6 = argent (Silvент/White)    7 = or (Gold/Yellow)
MyString NAME;    // The name of the object of the current element.
int X;       // X,Y, and SIZE, keep track of the location of the
int Y;       // element, along with the size of the element.
int SIZE;    // Using a standard 100x100 grid with sizes
             // 0-2.

};
#endif

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Parse.cc
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "Parse.h"
#include <fstream.h>
#include <iostream.h>
#include <stdlib.h>

/**
 * Parse.cc
 * @author Hans Mersinger
 * @version Pre-Beta
 **/

Parse :: Parse(Word TheInput[],int NUMBER){
    LoadData();
    int VALID;
    for (int i = 0; i < NUMBER; i++){
        if (TheInput[i].Key == ","){
            TheInput[i].Token = "comma";
        }
        else {
            VALID = 0;
            int TEMP = (int(TheInput[i].Key[0])-48); //
            getting the ansi value
            // of
            the first char in the
            if ((TEMP <= 9) && (TEMP >= 0)){  //
                string is it a number
                TheInput[i].Token = "number";
            }
            else {
                TEMP -= 17;  // 'A'
            }
            will now equal 0
        }
    }
}
if (TEMP > 30)
    TEMP -= 32;  // 'a'

will now equal 0

if (TEMP > 25)
    VALID = 0;  // not a

else {
    for (int ii = 0; ii <
INDEX[TEM]; ii++)
    {
        if (TheInput[i].Key
== DATA[TEM][ii].Key)
            TheInput[i].Token
= DATA[TEM][ii].Token;

            // MATCHED INPUT

            // EXIT LOOP

            ii = 9999999;
    }

    if (!VALID)
        // if !VALID then
didn't match it.

        TheInput[i].Token = "ERROR";

    }

}

}

void Parse :: LoadData() {
    MyString WORD;
    // The current word that has
    been read in.
    int Index = -1;  // the current letter we are
    on.
    int GO = 1;  // not to the end yet
    int MyNum = 0;  // number of elements before
    next letter.
    int KEY = 0;  // is this the word, or the
    key?
    IN.open("data.dat");  // opening the data file to
read
    if (!IN.good())
        // (!IN.good()) is true if
can’t read the file
        cout << "Error reading data file - 'data.dat'" <<
endl;
        exit(1);
    }
while ((GO) && (IN >> WORD)) { // Do loop while there are words, and GO == 1
    if (MyNum == 0){ // the there are no more words for the current letter
        Index ++;  // inc. index and start next letter
    
    if (Index == 26){  // stop, we have read in all the letters
        GO = 0; // we are done, stop
    }
    else {
        INDEX[Index] = WORD.ToInt();
        MyNum = INDEX[Index];
        DATA[Index] = new Word[WORD.ToInt()];
    }
    else {
        if (KEY) {
            DATA[Index][MyNum-1].Token = WORD;
            MyNum --;
            KEY = 0;
        }
        else {
            DATA[Index][MyNum-1].Key = WORD;
            KEY = 1;
        }
    }
}

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Parse.h
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

/**
* Parse.h
* @author Hans Mersinger
* @version Pre-Beta
.
```cpp
#ifndef PARSE_H
#define PARSE_H

#include "misc/MyString.h"
#include <fstream.h>
#include "Word.h"

class Parse {
public:

    Parse(Word TheInput[], int NUMBER);
    // VAue is the array of MySTRings that contain the words of
    // Blaen that were inuput. NUMBER is the number of elements
    // in the array. The array's [x][1] locations are changed to
    // of the tokens in the [x][0] locations.

    int Index(int Number);
    // the number of elements that are
    // Not used, not needed

private:
    void LoadData();
    int NUM;    // total number in the array
    Word * DATA[26];
    int INDEX[24]; // list the number of elements for each
letter
    ifstream IN;
};
#endif
```

GetInput.cc

GetInput.cc

/**
 * GetInput.cc
 * @author Hans Mersinger
```
* @version Pre-Beta
**/

#include "GetInput.h"
#include "misc/MyString.h"
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <stdlib.h>

GetInput :: GetInput(){

    /* std input is read for the blazen. Read in as an array of
chars  */
    /
    int place = 0;
    int GO = 1;
    char TEMP[256];
    cout << endl << "Blazen: ";
    cout.flush();
    while ((GO) && (TEMP[place] = getchar())){
        if (int(TEMP[place]) == 10){
            GO = 0;  //break out of the loop
            //pad the end of the input
            TEMP[place] = ' ';
            place ++;
        } else {
            place ++;
        }
    }
    if (!place) {
        exit(1); // exit if there was no input
    }

    /* now the array of chars is broken down into an array of
MyStrings */
    /*
int templength = 0;
int tempcount = 0;
TempArray = new MyString[place];
MyString currentword = "";
for (int i = 0; i < place ; i++){
    if (TEMP[i] == ','){
        if (templength > 0){
            TempArray[tempcount]=currentword;
            tempcount++;
            templength = 0;
            currentword = "";
            TempArray[tempcount]=",";
            tempcount++;
        }
    else{
        TempArray[tempcount]=",";
        tempcount++;
    }
    }
    else if (TEMP[i] == ' '){
        if (templength > 0){
            TempArray[tempcount]=currentword;
            tempcount++;
            templength = 0;
            currentword = "";
        }
    else {
        currentword = currentword +
        MyString(TEMP[i]);
        templength ++;
    }
    }
    cout << "LENGTH " << tempcount << endl;

    /***************************************************************************/
    /* Copies the array of words into array SENTENCE of size LENGTH */
    /***************************************************************************/
    LENGTH = tempcount;
    /***************************************************************************/
    /* for (i = 0; i < LENGTH; i++){
       TheInput[i].Key = MyString(TempArray[i]);
    }******************************************************************************/
*/
    delete [] TEMP;
}

GetInput :: "GetInput()
    delete [] SENTENCE;
}

void GetInput :: Setup(Word TheInput[])
    for (int i = 0; i < LENGTH; i++)
        TheInput[i].Key = TempArray[i];
    delete [] TempArray;
}

int GetInput :: Length()
    return LENGTH;
}

void GetInput :: Print()
    for (int i = 0; i < LENGTH; i++)
        cout << SENTENCE[i][0] << " " << SENTENCE[i][1]
        << endl;
    }

GetInput.h
GetInput.h
#include "misc/MyString.h"
#include "Word.h"

#ifndef GETINPUT_H
#define GETINPUT_H

class GetInput {
public:
    GetInput(); // Gets a array of chars from std input, returns
        array
                  // of MyStrings containing the words.
    ~GetInput(); // Deletes the array of words
    int Length(); // Returns the number of words in the array

```
void Setup(Word TheInput[]); // sets up the array
void Print(); // prints out all the words and the tokens.
MyString * * SENTENCE; // SENTENCE[LENGTH][2]

private:
    int LENGTH; // Number of words in the array
    MyString * TempArray; // temp holder of words found.
};
#endif

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Validate.cc
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

/**
* Validate.cc
* @author Hans Mersinger
* @version Pre-Beta
**/

#include "Validate.h"
#include <iostream.h>
#include <stdlib.h>

Validate :: Validate(Word TheInput[], int NUMBER){
    TOP = 100;
    BOTTOM = 0;
    LEFT = 0;
    RIGHT = 100;
    LENGTH = NUMBER;
    BL = 0; // counter of the current element that we are working with
    VALID = 0; // counts the number of passes through 'ParseRest()'
    cout << "Number of elements in the input = " << LENGTH << endl;
    TEMPBLAZEN = new Element[LENGTH];
    ParseBackground(TheInput);
    while (CURRENT + 2 < LENGTH){ // takes care of the rest of the input
        cout << "Parsing rest" << endl;
        ParseRest();

}  
BL ++;  
cout << "BL = " << BL << endl;  
BLAZEN = new Element[BL];  
for (int i = 0; i < BL; i++) {  
    // copies all the elements to the  
    // final array of elements  
    BLAZEN[i].Copy(TEMPBLAZEN[i]);
}
Print();
}
Valadate :: ~Valadate(){};

void Valadate :: ParseBackground(Word TheInput[]){
    if (TheInput[0].Token == "color"){
        TEMPBLAZEN[0].SetColor(IntColor(TheInput[0].Key));  
        //TEMPBLAZEN[0].SwitchColor();  
        TEMPBLAZEN[0].SetName("background");
        if (LENGTH == 1){
            TEMP = 0;
        }
    }
    else if (TheInput[1].Token == ("comma"){  
        TEMP = LENGTH - 2;  
        REST = new Word[TEMP+1];
        for (int i = 0; i < TEMP; i++){
            REST[i].Key = TheInput[i+2].Key;  
            REST[i].Token = TheInput[i+2].Token;
        }
        CURRENT = 0;
    }
    else {  // the TEMPBLAZEN does not start with a color  
        cout << "Error, no color" << endl;  
        ERROR();
    }
    cout << "finished color" << endl;
}

int Valadate :: IntColor(MyString Color){
    if (Color == "sable"){
        return 1;
    }
    else if (Color == "gules"){
        return 2;
    }
else if (Color == "azure"){
    return 3;
}
else if (Color == "vert"){
    return 4;
}
else if (Color == "purpure"){
    return 5;
}
else if (Color == "argent"){
    return 6;
}
else if (Color == "or"){
    return 7;
}
else {
    return 0;
}
}

void Validate :: ParseRest()
{
    VALID++;
    if (REST[CURRENT].Token == "ERROR"){
        cout << "Error in input" << endl;
        cout << "at word[" << REST[CURRENT].Key << "]
location number " << CURRENT << endl;
    }
    else {
        cout << "ParseRest" << REST[CURRENT].Token << "" <<
REST[CURRENT].Key << endl;
        BL++;
        cout << CURRENT << endl;
        if (REST[CURRENT].Token == "in"){
            CURRENT++;
            ParseLocation();
        }
        if (REST[CURRENT].Token == "number"){
            CURRENT++;
            ParseCharge();
            //if (REST[CURRENT].Token == "on"){
            //    CURRENT++;
            //    // ParseRest();
            //    //}
            if (REST[CURRENT].Token == "between"){
cout << "I got here, right?" << endl;
cout.flush();
int t = TOP;
int b = BOTTOM;
int l = LEFT;
int r = RIGHT;
Split(3,2,t,b,l,r);
TEMPBLAZEN[BL].SetX(((RIGHT-LEFT)/2)+LEFT);
TEMPBLAZEN[BL].SetY(((TOP-BOTTOM)/2)+BOTTOM);
TEMPBLAZEN[BL].SetSize(RIGHT-LEFT);
CURRENT++;
Split(3,1,t,b,l,r);
cout << " TOP " " BOTTOM " " " <<
LEFT << " " RIGHT << endl;
ParseRest();
if (REST[CURRENT].Token == "and") {
    CURRENT++;
    Split(3,3,t,b,l,r);
    ParseRest();
}
else { //error, something is not
right

}
Split(1,1,t,b,l,r);
}
else {
    TEMPBLAZEN[BL].SetX(((RIGHT-LEFT)/2)+LEFT);
    TEMPBLAZEN[BL].SetY(((TOP-BOTTOM)/2)+BOTTOM);
    TEMPBLAZEN[BL].SetSize(RIGHT-LEFT);
}
if (VALID > LENGTH) {
exit(1);
}
}

void Validate :: ParseCharge(){
cout << " ParseCharge" << endl;
if (REST[CURRENT].Token == "charge"){
    TEMPBLAZEN[BL].SwitchColor();
    TEMPBLAZEN[BL].SetName(REST[CURRENT].Key);
    CURRENT++;
    if (REST[CURRENT].Token == "modifier"){
        cout << " modifier" << endl;
    }
}
SetModaifier();
}

if (REST[CURRENT].Token == "color"){
    cout << " color" << endl;
    SetColor(IntColor(REST[CURRENT].Key),BL);
    CURRENT++;
}

else {
    ERROR();
}

void Valadate :: ParseLocation()
// this function will change the location
}

void Valadate :: SetColor(int c, int i){
    TEMPBLAZEN[i].SetColor(c);
    TEMPBLAZEN[i].SwitchColor();
    if (TEMPBLAZEN[i-1].Color()){
        SetColor(c,i-1);
    }
}

void Valadate :: SetModaifier(){
// this function will change the modafires

void Valadate :: ERROR()
// exit(1);
}

// this function is to do something when errors happen

void Valadate :: Split(int n, int p, int t, int b, int l, int r){
    // (number of splits, what split we are at, top, bottom,
    left, right)
    TOP = t;
    BOTTOM = b;
    LEFT = (1+((p-1)*((r-l)/n)));
    RIGHT = (1+(p*((r-l)/n)));
}

void Valadate :: Print()
    cout << BL << endl;
for (int i = 0; i < BL; i++)
  cout << i << endl;
  BLAZEN[i].Print();
}

Validate.h

/**************************

/***** Validate.h
* @author Hans Mersinger
* @version Pre-Beta
***/

#ifndef VALDATE_H
#define VALDATE_H
#include ".\misc\MyString.h"
#include "Element.h"
#include "Word.h"

class Validate {
public:  // array.getLength() should return the length of the array
  Validate(Word TheInput[], int NUMBER);
  ~Validate();
  void ParseBackground(Word TheInput[]);
  int IntColor(MyString Color);
  void ParseRest();
  void ParseCharge();
  void ParseLocation();
  void SetColor(int c, int i);
  void SetModafier();
  void ERROR();
  void Split(int n, int p, int t, int b, int l, int r);
  void Print();
private:
  Element * TEMPBLAZEN;  //array of all the charges and colors there of.
  Element * BLAZEN;      //Actual blazen to be used
  int BL;                //CURRENT length of the TEMPBLAZEN array
  int LENGTH;
  Word * REST;           //temp array. Stores the TEMPBLAZEN that has
not been parsed
int TEMP ; //yet. TMP is the number of element in REST
int CURRENT ; //CURRENT location in REST
int LEFT,RIGHT, TOP, BOTTOM;
int VALID;

};

#endif
GENERAL GLOSSARY

->: A substitution marker. The expression on the right side can replace the expression on the left side.

algorithm: A predetermined series of instructions for carrying out a task in a finite number of steps.

alphabet: The finite set of symbols which makeup the words of a languages.

context free grammar: A grammar with a finite set of productions of the form: One Terminal -> a finite string of terminals and or non-terminals.

context sensitive grammar: A grammar with productions unrestricted to having only a terminal on the left side of the production.

dynamic: Term referring to a variable that does not have a set value, value changes depending on user input.

keys: The main identifying element in a record which associated information is attached.

language: A certain specified set of strings of characters from the alphabet.

natural language: A language that is defined by a context sensitive grammar.

non-natural language: A language explicitly created without the need for semantic interpretation and based largely on strict symbolic expression.

non-terminal: A symbol used in a grammar to represent a production.

parse: A linguistic term for breaking up a sentence in a natural language into its syntactical components.

pointer: An indirect method of referencing a variable.

production: The set of grammar rules followed to create(or parse) a word by a grammar.

programming language: A non-natural language created for ease of parsing.

reference value: The value of the variable that a pointer is referring to.

reference location: The memory location at which the variable that a pointer is referring to is located.
**semantics:** The field of linguistic analysis concerned with the meaning component of language.

**syntax:** The field of linguistics concerned with the grammatical relations of words in a sentence.

**static declaration:** A declaration where the value is always declared to be the same when run.

**terminal:** A symbol that cannot be represented by anything else.
Glossary of Heraldic Terms

ARGENT: Silver

AZURE: Blue.

BASE: The lower part of the shield.

BORDURE: One of the ordinaries, being a narrow border around the edge of the field.

CHIEF: One of the ordinaries, being the upper third of the shield; also used to indicate the position of a charge, e.g. 'and in chief two...'.

DEXTER: The right side. When applied to a shield it refers to that part which would be towards the right side of the man carrying it, thus the portion on the viewer's left.

FESS: One of the ordinaries, being a broad horizontal band across the middle of the shield.

GULES: Red.

OR: Gold. Can be depicted as yellow or gold.

ORDINARY:

PURPURE: Purple

SABLE: Black.

SINISTER: The side of the shield toward the left of the man carrying it, thus to the right when viewed from the front.

VERT: Green.
WORKS CITED AND REFERENCED


http://www.glendrakon.midrealm.org/~dragon/heraldry/howtoblazon.html (17 June 97)


"Heraldry and Programming Languages: the Complexity of Natural Languages Examined through the Parsing of the Heraldic Blazon"

Approved by:

[Signature]
Assistant Professor of Computer Science

[Signature]
Assistant Professor of Computer Science

[Signature]
Instructor in Geology

[Signature]
Chair, Department of Computer Science

[Signature]
Director, Honors Thesis Program

[Signature]
Director, Honors Program