

Writing a grammar and Parser for Blazon

Thomas Heyes

April 28, 2013

Abstract

Blazon is the semi-formal language of family crest and heraldry, dating back to the twelfth century. Using a well structured grammar to describe a coat of arms in a top down approach the language provides a robust yet flexible way to define a textual description of what is naturally a very graphical subject. Using a test-driven development model I have produced a project which is capable of parsing a large subset of Blazon. Encompassing several fields of Computer Science ranging from parsing to HTML5 graphics the application provides a platform that demonstrates how modern concepts and technologies can be used to represent a subject that pre-dates them by centuries.

Firstly giving a brief background and description of the language of Blazon, this report goes on to describe both how the Blazon parsing application works as well as and how the project was implemented and tested. From a simple shield of a single colour through partitioning into sections and sub partitioning as well as covering different line-types before heading onto geometric charges, honourable charges and the rule of tincture before finally discussing semi-formal charges.

After thoroughly describing the language I go on to discuss the implementation of the project using test driven development. Producing lots of iterations increasing the functionality of the project gradually and performing regression testing to ensure the soundness of my code base. Initially starting with a couple of Python script based prototypes I describe how the project was iteratively built into a fully fledged web application over the course of the academic year. Given more time I would attempt to expand the project further into parsing and drawing a larger subset of Blazon with features such as Counter Charging and combining shields with Quartering.

Contents

1	Introduction	3
1.1	Aims	3
1.2	Achievable	3
1.3	Limitations	3
2	What is Blazon?	4
2.1	The Seven Tinctures	4
2.2	Furs	5
2.3	Fields	6
2.4	How to Blazon a Coat of Arms: Part One	6
2.5	Partitioning a Field	7
2.6	How to Blazon a Coat of Arms: Part Two	9
2.7	Sub-Partitioning	11
2.8	How to Blazon a Coat of Arms: Part Three	12
2.9	Line Types	14
2.10	Charges	15
2.11	Attitudes	16
2.12	The Rule of Tincture	17
2.13	How to Blazon a Coat of Arms: Part Four	17
2.14	Counter-Charging	19
2.15	Directions and Sides	20
2.16	Overall	22
2.17	Variations	22
3	Lexical Analysis	24
3.1	What is Lexical Analysis?	24
3.2	Lexical Analysis of Blazon	24
3.3	More Involved Lexing	26

4	Defining a Grammar for Blazon	29
4.1	What is a Grammar?	29
4.2	Context free Grammars	30
4.2.1	An Example Context Free Grammar	30
4.3	Context Free Blazon	32
4.4	Formatting Input	34
4.5	Tokenizing Strings	34
5	Parsing	35
5.1	Parse Trees	35
5.2	Parsing Blazon	35
5.3	Parsing Example	36
5.4	Error handling	39
6	Graphical Representations	40
7	Design	41
8	Testing	42
8.1	Black Box Testing	42
8.2	Regression Testing	43
8.3	Blazon Seeking Missile	44
8.4	Application Level Testing	45
8.5	Testing for Completeness of the Blazon Subset	45

Chapter 1

Introduction

My project was to *Write a Grammar and Parser for Blazon*. Another student project by *Luke Torjussen* done the year before my own had attempted, very successfully, to produce an application that both parsed and drew representations of Blazoned coats of arms. This prior project used a parse generator to handle the grammar, lexing and parsing of the language. It was my objective to see if performing the parsing by hand in my own application could realise a more complete representation of Blazon.

Initially then my project was entirely based in tokenising, lexing, parsing and validating Blazon sentences. However my project grew later-on to also attempt to encompass producing a graphical representation of a parsed Blazon sentence as well as providing a plain English translation.

1.1 Aims

The aim of my project was to produce an application which upon receiving a Blazon sentence would firstly validate the sentence according to the rules of the Blazon language and then produce an English translation.

1.2 Achievable

1.3 Limitations

Chapter 2

What is Blazon?

Blazon is the language of heraldry and family crests, it dates back to the twelfth century and provides a strict set of rules about how to produce a coat of arms. There are several different versions of Blazon found around Europe however all follow the same behavioural rules regarding tinctures and charges. Most versions differ only in the set of tinctures and honourable ordinaries either having a more generous or conservative view on what is and isn't acceptable for example, African Blazon allows for an Orange coloured Tincture whilst English Blazon does not. For this project I focused on exclusively on a strict English Blazon.

Blazon is a powerful language for allowing limitless combinations and configurations of patterns and shapes to be *Blazoned* onto a shield in and recorded in a concise textual description.

What is impressive about the language is that it achieves this flexibility whilst reaming fairly formal and well defined, restricting the set of tinctures to total seven and maintaining a fairly low number of pre-defined honourable ordinaries.

Blazon, like any other language, has some unique terminology used to address certain aspects of heraldry. Anyone attempting utilize the language will need to familiarise themselves with this terminology.

2.1 The Seven Tinctures

The most fundamental elements of Blazon are the Tinctures. Tinctures are the set of colours allowed in coats of arms. English Blazon defines a set of seven tinctures and places them into to groups Metals and Colours.

Although at first this may seem overly restrictive Blazon overcomes having a limited set of valid tinctures by leaving interpretation of the tone and

The Seven Tinctures		
Tincture	Colour	Metal/Colour
Azure	Blue	Colour
Argent	Silver	Metal
Gules	Red	Colour
Or	Gold	Metal
Purpure	Purple	Colour
Sable	Black	Colour
Vert	Green	Colour

Table 2.1: Table of Tinctures found in English Blazon, the corresponding colours and whether each tincture is or isn't a metal.

shade of each Tincture's corresponding colour to the artist producing the shield. An Azure, blue, shield could be a light sky blue or a dark navy shade of blue depending upon the artists preference, as long as it is recognisably blue.

The Freedom of Interpretation is a great strength of Blazon vastly increases the different graphical variations of Blazoned shields without radically increasing the size of the language by exhaustively listing a set of valid tones and shades.

With regard to the two metallic Tinctures Or, gold, and Argent, silver, having freedom of interpretation allows for matt, non-metallic, colours to be placed onto a shield, although they will still behave as metal Tinctures. It is not uncommon to see yellow instead of gold and white instead of silver on many coats of arms.

2.2 Furs

Blazon incorporates several furs commonly used in coats of arms into the language. These behave in much the same way as tinctures but don't have the same metal or Colour property.

There are several pre-defined patterns for furs that are used directly instead of actual fur on a shield. These patterns generally consist of a repeating pattern on-top of a background colour. Each fur has a unique pattern although several are very similar, some simply being the inverse of another.

The colours of Furs are pre-defined but can be explicitly stated as differing from the normal colours by stating the Tinctures to be used instead. For example for example Ermine coloured silver and black would be defined as "*Ermine Argent and Sable*"

2.3 Fields

A Field is simply an area on a shield. Initially a shield has one implicit Field, which is the entire area of that shield. Blazon allows for two operations on fields, a field can be Tinctured, with a Tincture or a Fur, or a field can be Partitioned. Tincturing a Field dictates that the whole area incorporated in that Field be filled in the colour of that Tincture. Partitioning I will cover in much more detail later on.

2.4 How to Blazon a Coat of Arms: Part One

I have now defined a large enough subset of Blazon for a couple of basic examples.

In each example we are implicitly provided with a single Field which encompasses the entire body of the shield. Then we Tincture that Field by providing a Tincture and we have successfully Blazoned a shield.

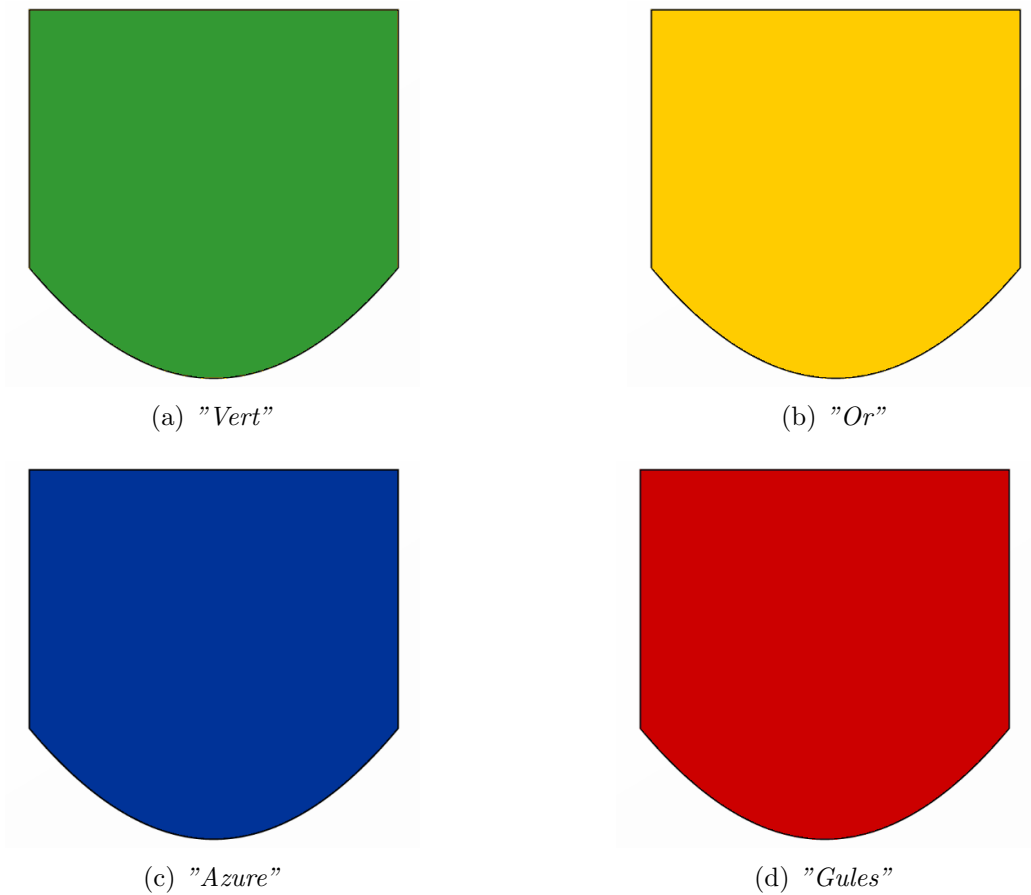


Figure 2.1: Basic Blazon examples.

Although the examples in Figure 2.1 are very basic each is a complete instance of a valid Blazoning of a shield.

2.5 Partitioning a Field

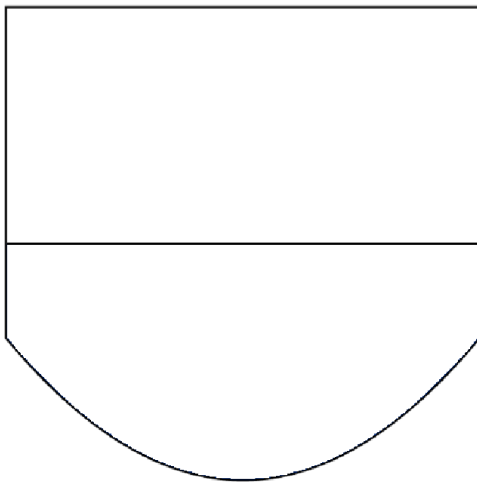
The Blazon I have defined so far is very limited, only allowing for single Tincture shields. The next natural area of the language to define is the operation of Partitioning a Field.

As stated previously a Field can be either Tinctured or Partitioned. To Partition a Field the key-word "*Per*" is used. It is obligatory that the word immediately after "*Per*" is a type of partition. Blazon has several pre-defined partitions ranging from the very simple "*Fess*" which divides a Field horizontally in half.

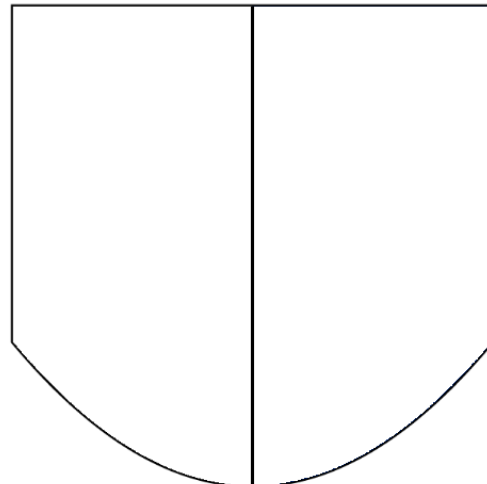
Partitioning a Field divides it up into several smaller fields the number and shape of which depend on the type of Partition used. After the Partition has been stated the Blazon sentence must go on to address the resulting new Fields.

A shield is Implicitly Blazoned from top left to bottom right with the top most Field taking priority firstly and if two or more Fields are adjacent at the same height the left most takes priority.

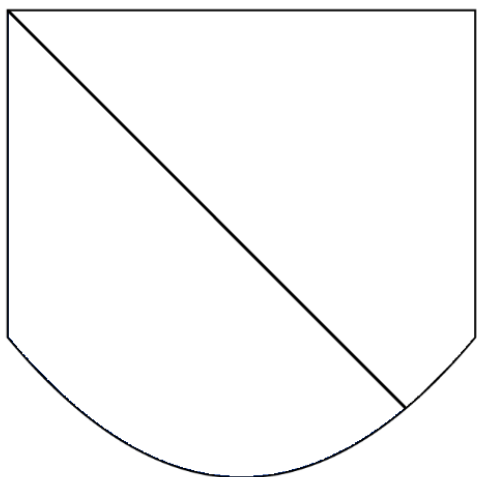
The Blazon sentence is only complete when all the Fields have been Tinctured. If a Blazon sentence Partitions a shield into two Fields and then provides only one Tincture that sentence is Invalid.



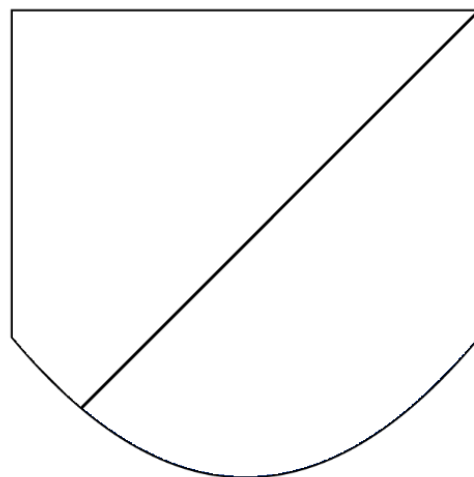
(a) "*Fess*"



(b) "*Pale*"



(c) "*Bend*"



(d) "*Bend Sinister*"

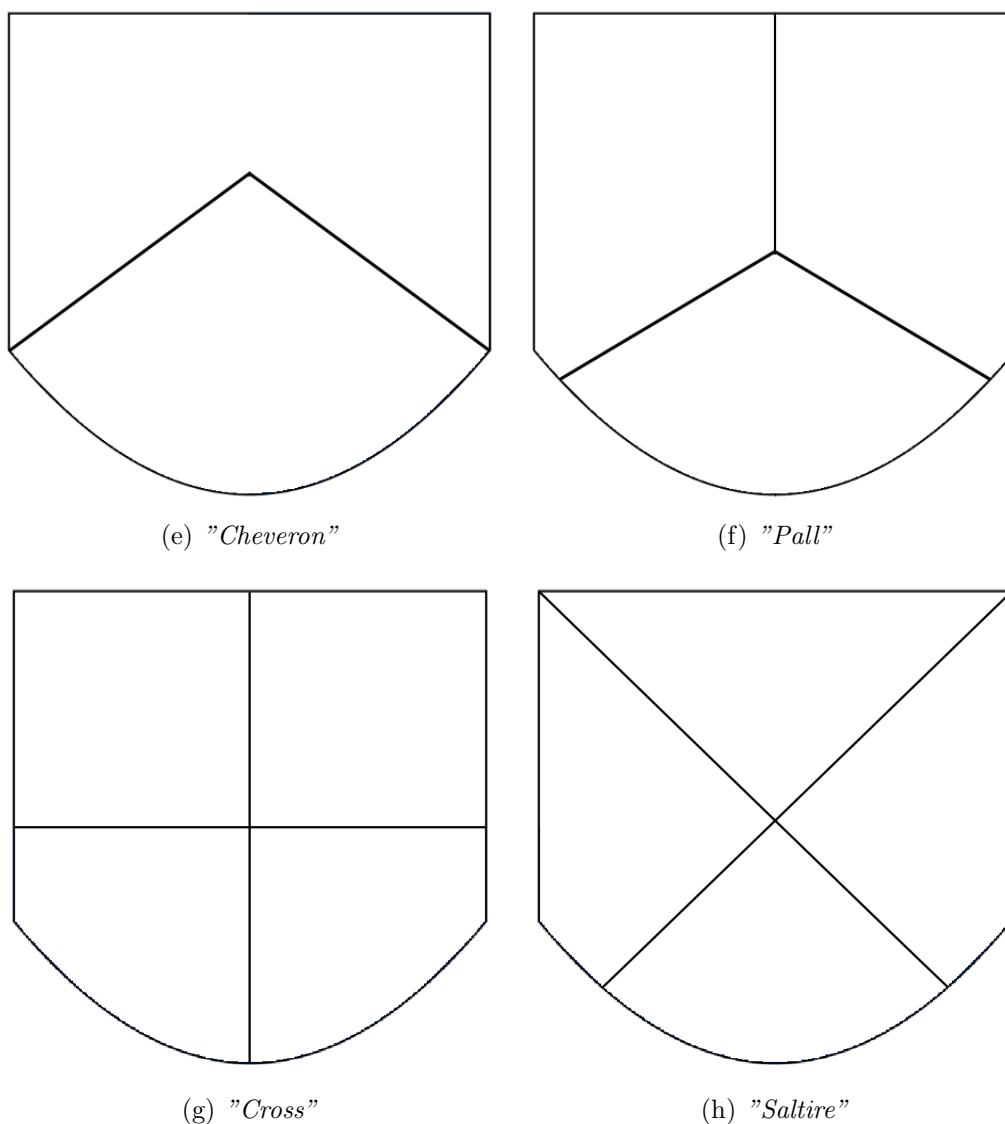


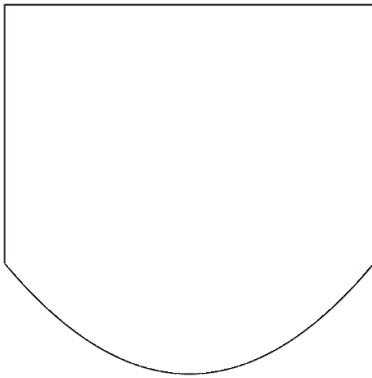
Figure 2.1: "Valid Partitions of a Field"

2.6 How to Blazon a Coat of Arms: Part Two

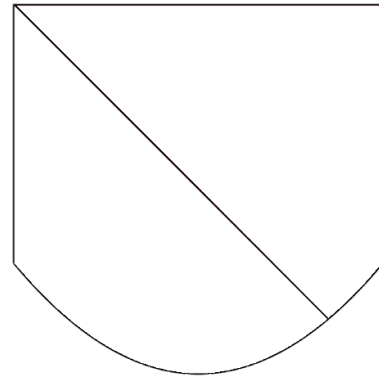
Partitioning is a very powerful aspect of Blazon and increases the number of possible shield designs immensely. A lot of very striking designs can be Blazoned onto a shield with very short Blazon sentences making use of Partitioning.

The Blazon sentence, "*Per Bend Gules and Azure*" Implicitly starts with

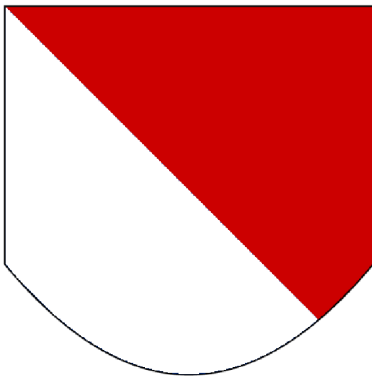
a single Field which encompasses the entire body of the shield before partitioning that Field into two smaller Fields with the keyword *"Per"* declaring a partition followed by the word *"Bend"* which is a diagonal division of a field from top left to bottom right. The Blazon goes onto Tincture the two new fields with the two Tinctures *"Gules"* and *"Azure"* respectively, following Blazon's rule about evaluating the topmost and then leftmost field first the upper section of the shield is Tinctured *"Gules"* and then the lower half is Tinctured *"Azure"*. There is no more Blazon reaming in the sentence and there are also no empty fields upon the shield, therefore this is a valid Blazon sentence and a striking red and blue shield has been produced.



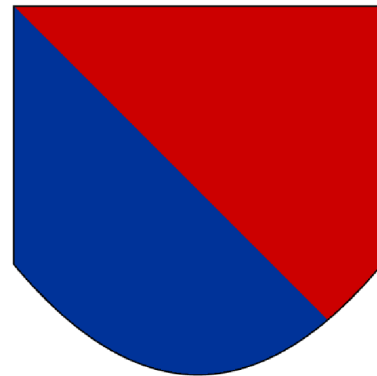
(a) *"A single empty Field."*



(b) *"The Field has been partitioned Per Bend"*



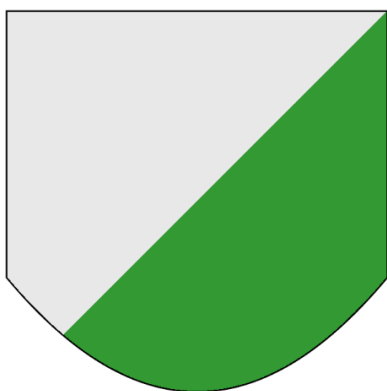
(c) *"The Topmost Field is Tinctured Gules"*



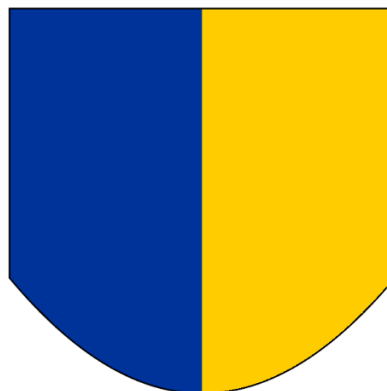
(d) *"The final Field is Tinctured Azure"*

Figure 2.2: *"Per Bend Gules and Azure"*.

Applying the same method as show above it is possible to validate the following Blazon sentences of similar complexity.



(a) *"Per Bend Sinister Argent and Vert."*



(b) *"Per Pale Azure and Or"*

Figure 2.3: *"Two more examples of valid Blazon sentences".*

2.7 Sub-Partitioning

The process of Partitioning a field produces two or more smaller Fields. This allows for some very striking, though simple, designs. Blazon allows however for fields to be Sub-Partitioned. The new Fields produced after a Partitioning can themselves be Partitioned.

Considerably more advanced patterns can be achieved buy using Sub-Partitioning. The complexity is potentially infinite as there is no limit enforced by the language as to how many Partitions are allowed. It is rare to see a historical shield Sub-Partitioned more than three times. Sub-Partitioning makes use of exactly the same set of Partition types as regular Partitioning.

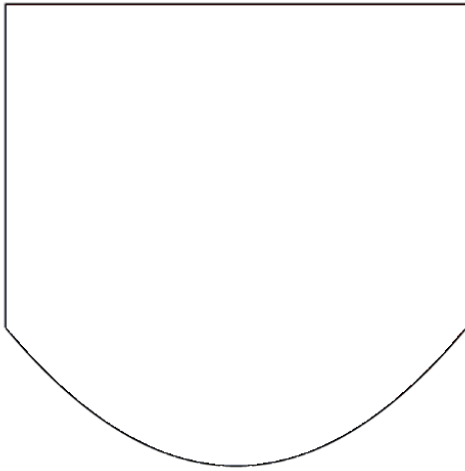
Sub-Partitioning is expressed in exactly the same way as partitioning. Once the initial field has been partitioned instead of providing a Tincture for one of the new fields the keyword *"Per"* is used to define a partition and then the type of partition is stated for example *"Fess"*. Then each of the new Fields need to be addressed either by being Tinctured or further partitioned.

The new Fields created by the Partition fully evaluated before any other Fields on the shield are addressed. The new sub-fields are addressed according to the same rules present in regular Partitioning, namely the top most Fields take Priority then the left most, once they have all been evaluated the next field according to partitioning rules is addressed.

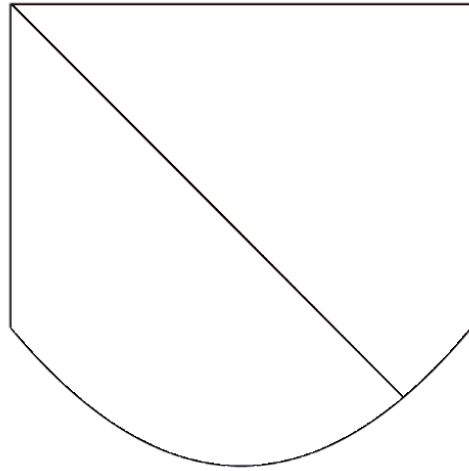
2.8 How to Blazon a Coat of Arms: Part Three

With Sub-Partitioning the set of valid Blazon sentences ceases to be finite, it is possible to repeatedly Partition a Field then Sub-Partition the new Sub-Fields and endlessly repeating this process. Very complex patterns can now be created with precisely defined Sub-Fields.

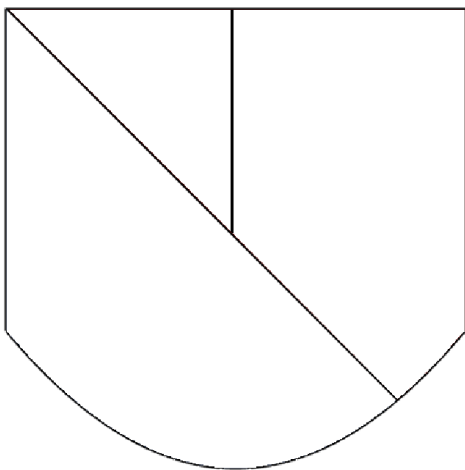
"Per Bend Per Pale Sable and Argent Per Fess Or and Gules" is an example a Blazon sentence which makes use of Sub-Partitioning. As before the the sentence is addressed from left to right and there is an implicit Field over the entire body of the shield. The sentence is valid if there are as many Fields defined as there are instances of Tinctures in the sentence.



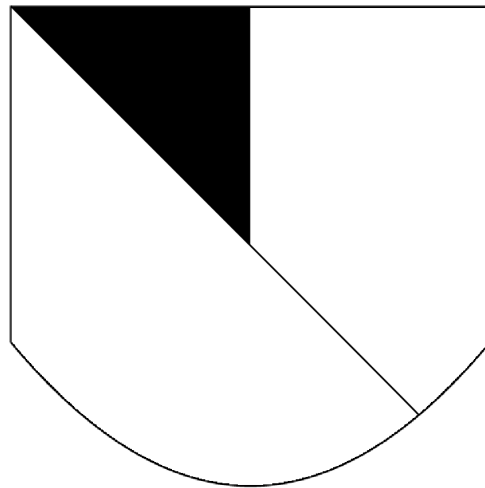
(a) *"A single empty Field."*



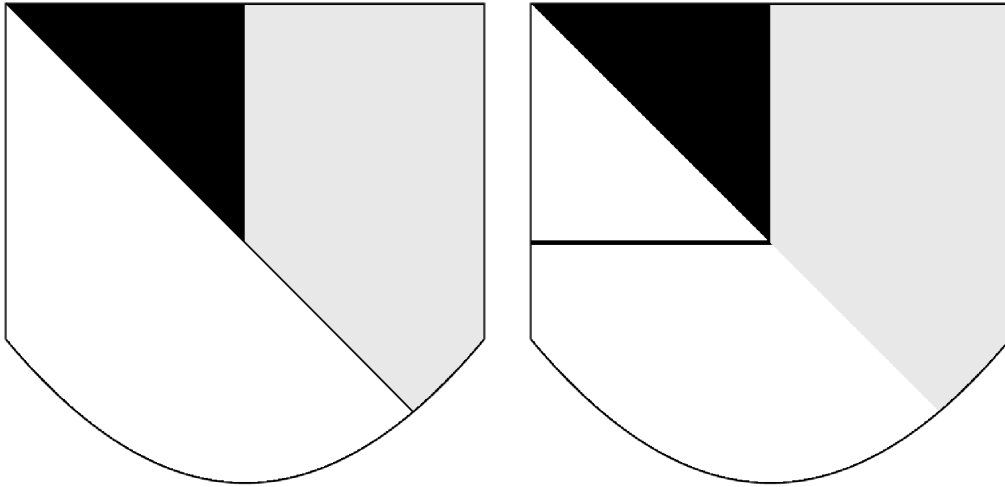
(b) *"The Field has been partitioned Per Bend"*



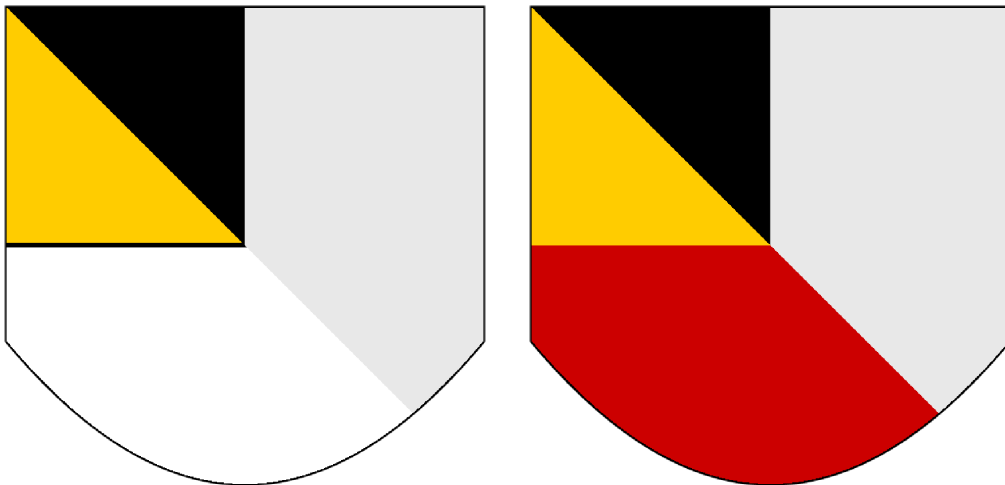
(c) *"The Topmost Field has then been Sub-partitioned Pale"*



(d) *"The Top Left most Sub-Field is Tinctured Sable"*



(e) *"The remaining Field is Sub-Partitioned Fess"* (f) *"The Top Left Most Sub-Field is Tinctured Or"*



(g) *"The remaining Sub-Field is Sub-Partitioned Fess"* (h) *"The final Field is Tinctured Gules"*

Figure 2.3: *"Per Bend Per Pale Sable and Argent Per Fess Or and Gules"*.

2.9 Line Types

Blazon allows for Partitions to be patterned with different Line Types. The line division through a Field when partitioning is implicitly a straight line however there are a set of pre-defined Line Types which can be used instead.

A Line Type is declared in Blazon simply by stating the name of the Line

Type desired immediately after a partition. For example *Per Fess Embattled Argent and Sable* is the Blazon sentence for a Silver and Black shield divided horizontally with an "Embattled" line, see figure 2.4.

Line Types can be applied to Sub-Partitions and its possible to have multiple Line Types in a single Blazon sentence.

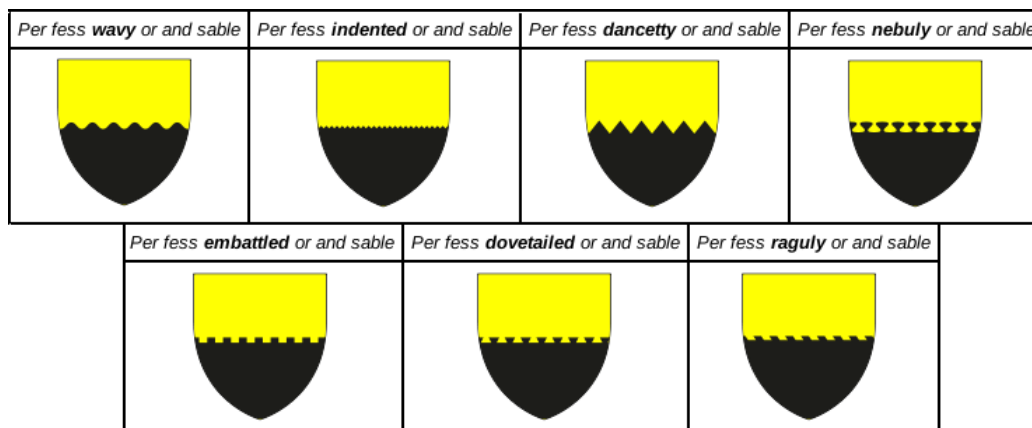


Figure 2.4: Different line types valid in Blazon.[1]

2.10 Charges

Another powerful aspect of Blazon is the ability to place pictures, called Charges, onto a shield. Charges can be placed onto any Field that has been Tintured. There are three subsets of Charge, the Honourable Charges, the Geometric Charges and Semi-formal Charges.

The Geometric Charges are the most simple, they are the set of basic shapes such as a square and a circle. The Honourable Charges are a set of predefined simple shapes some of which take after Partitions, such as "Bend". The set of semi-formal charges contains everything that can be described textually, common examples are Lions and Eagles some royal Blazons have a crown as a Charge.

A Charge is declared in a Field by using a Quantifier immediately after the Field has been tintured. The Quantifier is a pre-fix to the charge and determines how many instances of this Charge are present on this Field. For example the Sentence "Vert a Crown Or" Takes the implicit Field then Tintures it Vert before Charging the Field with a Crown Tintured Or, this would produce a green shield with a gold crown on it. Semi-formal Charges can be Tintured "Proper" which is their natural colours. Even if a Charge

is or a fictional creature it will have a proper colouring. Dragons are for example green when Tinctured "*Proper*".

The size and position of a Charge is defined so that a Charge should take up as much space as possible without obscuring any other Charge or intersecting the perimeter of the Field it is being placed onto.

Its possible to have several instances of the same charge on a Field. There are common ways to position such groupings of Charges however there is freedom for the artist to arrange groups of Charges differently especially if the area doesn't lend itself to the traditional placement. A very wide but short Field would better suit its Charges being placed in a straight line rather than the usual grouping as long as each Charge is visible.

The prefix keyword can be a number or amount to define how many types of this Charge are present in this Field, *Argent three Delfs Sable* would produce a silver shield with three black squares on it in a triangular pattern, one each in the top corners and one centrally placed in toward the bottom.

It is even possible to have multiple groupings of different Charges on a Field each Charge in the grouping is treated individually so that it is allowed to obscure any other Charge.

Keywords do exist for more specific positioning of Charges such as "*Beneath*" and "*Above*" these make it possible to position Charges relative to other Charges.

Finally, it is possible to place Charges upon other Charges. This can be done with the keywords "*in*" or "*upon*". For example, "*Vert a Bend Azure upon a Chief Argent*" would produce a green shield with a silver bar along the top which has a blue diagonal bar on top of it.

2.11 Attitudes

Blazon allows semi-formal Charges to be given an *Attitude* to add further description to the Charge. This normally takes the form of an adjective applied as a suffix, one of the features inherited from French. For example a lion can be described as rampant or as passant both of which alter the pose the lion will be depicted in. A lion rampant is depicted as rearing up on its hind legs claws raised while a lion passant is standing horizontally with a single claw raised.

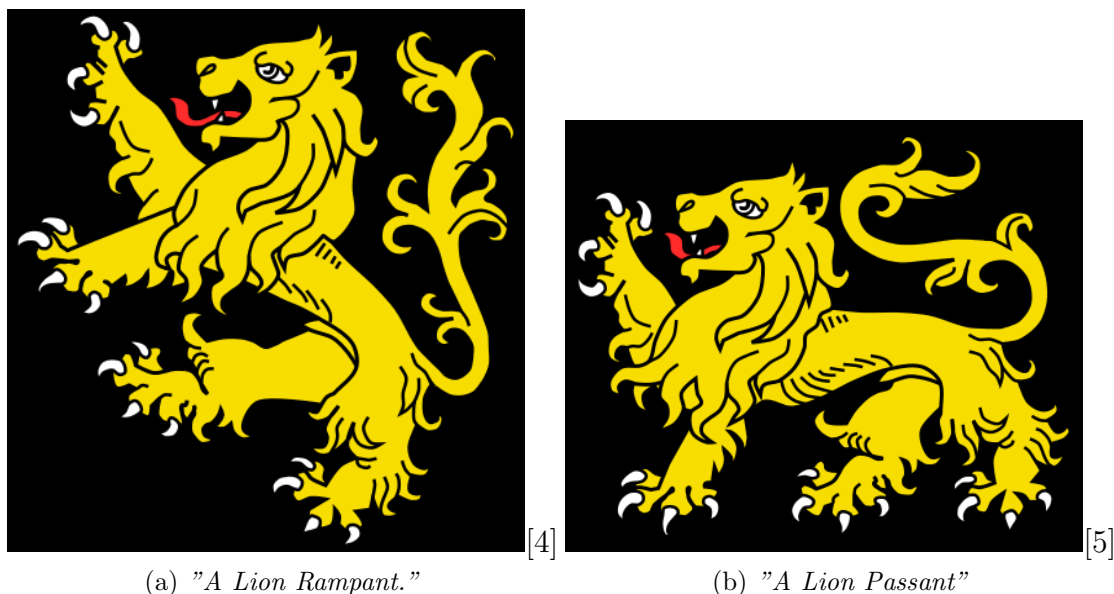


Figure 2.5: "An example of applying attitudes to a charge".

2.12 The Rule of Tincture

One of the most important and widely followed rules of Blazon is the Rule of Tincture which states that *a colour may not be placed upon a colour nor a metal be placed upon a metal*[2, p.46]. This rule makes certain Tincture combinations of Charges and Fields invalid with each other. As stated above in Table 2.1, every tincture is either a Metal or a Colour in accordance with the Rule of Tincture, a Charge Tinctured with a Colour can only be placed upon a Field Tinctured with a Metal. Likewise, A Charge Tinctured with a Metal can only be placed upon a Fields Tinctured with a Colour.

Therefore, "*Azure a Cross Gules*" is an invalid Blazon sentence because both Azure and Gules are Colours and having a Colour Charged upon another Colour breaks the Rule of Tincture. The Rule also applies to Charges placed on Charges.

2.13 How to Blazon a Coat of Arms: Part Four

Charges are equally as important in Blazon as Partitioning, they allow the language a freedom to have any object placed upon them. With this extra

functionality comes a slightly more complex terminology as described above with new keywords and the need to observe the Rule of Tincture.

Taking, "*Per Pale Azure a Cross Argent and Or a Saltire Gules*" as an example of a Blazon sentence which uses charges it is possible to demonstrate how Charges are used in Blazon.

Starting with the implicit initial Field the Blazon sentence starts "*Per Pale*" which Partitions the Field Per-wards, vertically. The left hand field is then addressed as it is the Top-Left most Field not yet Tintured, the next word in the Sentence is a Tincture "*Azure*" which indicates this Field is to be blue. The next word in the sentence is "*a*" which is a quantifier for a single charge which the Sentence states is a "*Cross*" because the Charge is on a Field which has been Tintured with a Colour, the Charge must be a Metal for the Blazon to be valid in this case the Tincture is "*Argent*" which is indeed a Metal.

As the next word of interest in the the Blazon sentence is "*Or*" which as a Tincture and there have been no more charges declared we address the next Field so the right half of the shield should be gold in colour. Then there is another qualifier again its "*a*" indicating a single Charge which is stated as being a "*Saltire*" Tintured "*Gules*" this produces a Colour Charge on a Metal Field and thus respects the Rule of Tincture.

There are no more words remaining in the Blazon sentence, all the Fields and Charges have been Tintured and the Rule of Tincture has not been broken, therefore, "*Per Pale Azure a Cross Argent and Or a Saltire Gules*" is a valid Blazon sentence which contains two Fields each with a single Charge.

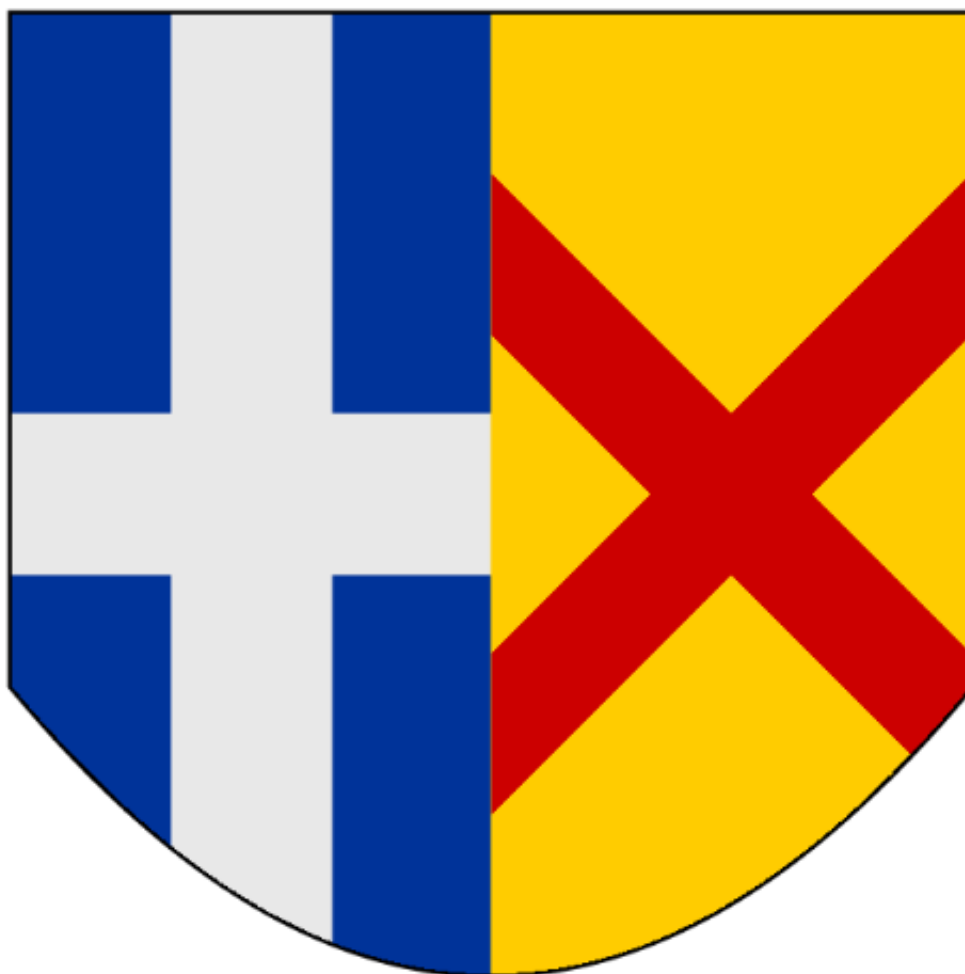


Figure 2.6: "*Per Pale Azure a Cross Argent and Or a Saltire Gules*".

2.14 Counter-Charging

Blazon also provides functionality for positioning Charges across Fields; this is achieved through *Counter-Charging*. If a Blazon sentence has two adjacent Fields, then a Charge can be Counter-Charged across the two fields, this would split the charge down the line of Partition between the Fields before Tincturing each half of the Charge inversely to the Field it lies in. Counter-Charging is performed using the keywords *Counter Charged* as a suffix operator to a charge and must have two tinctured fields defined before it.

In the example shown in figure 2.6, the Blazon sentence for the shield defines a Field Partitioned "*Per Pale*" then the two resulting Fields are Tinctured "*Argent*" and "*Gules*" then a Charge is defined "*a Bend*" which places a single diagonal bar upon the Field then the keywords *Counter Charged* are supplied instead of a Tincture which extends the bar across the two Fields and colours the section on the *Argent* Field in *Gules* and the section on the *Gules* Field in *Argent*.



Figure 2.7: "*Per Pale Argent and Gules a Bend Counter Charged.*"[3]

2.15 Directions and Sides

The Blazon language contains a set of words for describing positions and directions. These words are obviously useful for more accurately placing Charges on a Field. The top of the shield is refereed to as *Chief* and the bottom is refereed to as *Base*. The Left hand side of the shield, from the viewers perspective is called to as *Dexter* and the right hand side, again from the views perspective, is *Sinister*.

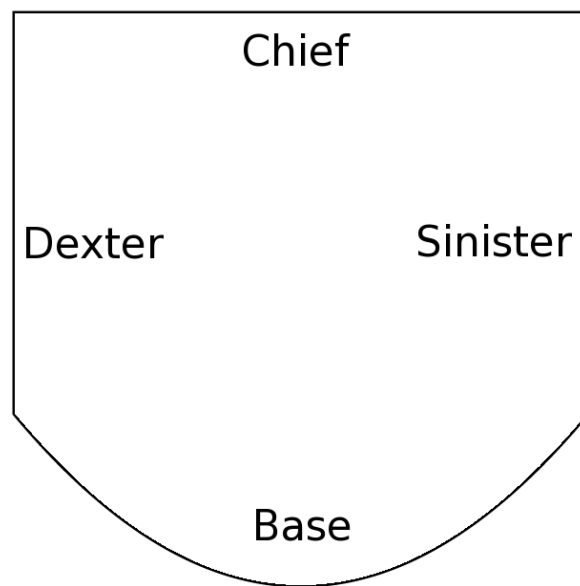


Figure 2.8: *The sides of the Field*

These keywords can be used to position Charges in a Field relative to each other. They can also be used to define direction, the most basic example of this is the Partition "*Bend Sinister*" which is a diagonal line from lower left to upper right. Semi-formal Charges can also be given a direction like so; "*A Purse Fesswise Argent Spilling Coins to Dexter Argent*" which would depict a purse lying horizontally with coins to the left hand side, from the viewers perspective.

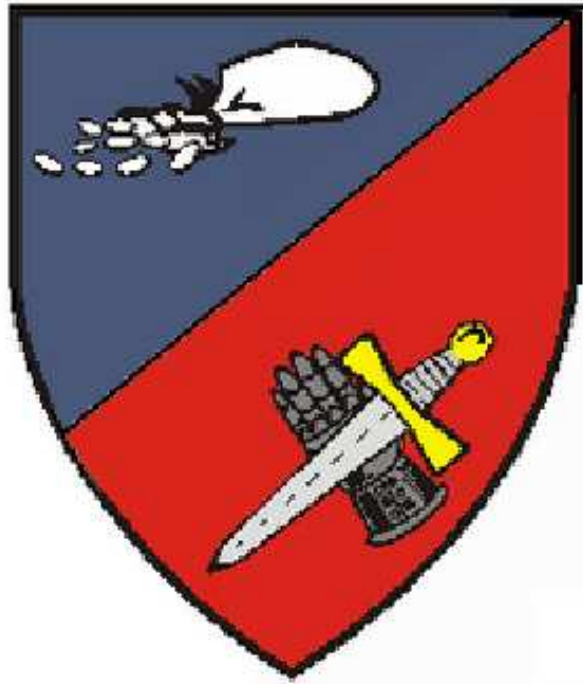


Figure 2.9: *Per Bend Sinister Azure A Purse Fesswise Argent Spilling Coins to Dexter Argent Gules a Gauntlet Bendwise appaumed (an attitude meaning open palmed) Argent Bendwise sinister inverted proper.*[6]

2.16 Overall

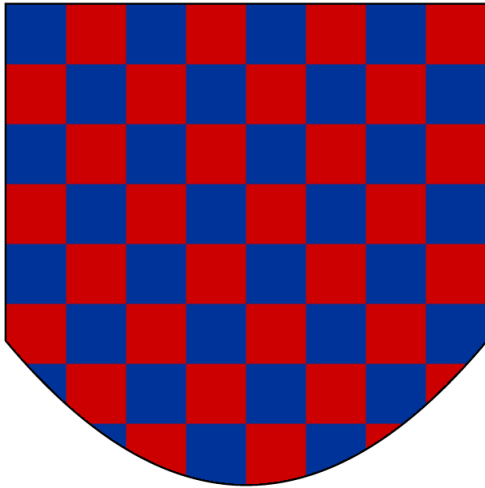
Blazon also encompasses a rudimentary system for layering. Charges can be placed on top of the entire shield as long as they don't break the Rule of Tincture or obscure another Charge. This is achieved with the keyword *Overall* which occurs at the end of a blazon sentence, when all the Fields have been Tinctured, and is then followed by a Charge.

2.17 Variations

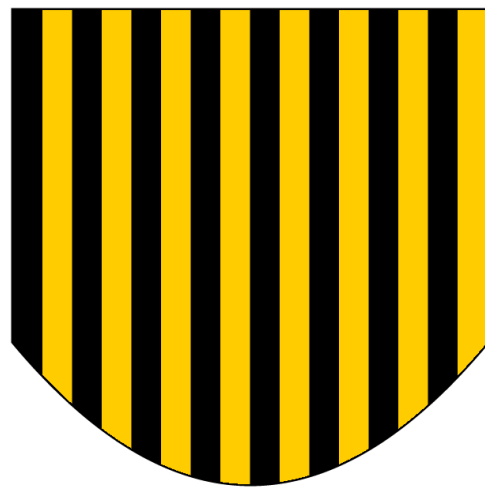
Finally Blazon has several pre-defined short hands for commonly used patterns known as variations. These patterns are all achievable through sub-partitioning but in a very verbose manner. "*Checky Argent and Gules*" produces a silver and red grid pattern which for all intents and purposes is the same as placing a large number of *Delfs* Tinctured Gules onto a Field

Tinctured Argent.

By default Blazon will produce the pattern in quantities of six. The number of repetitions can be stated by adding the suffix *"of"* and then providing a quantity.



(a) *"Checky of eight Azure and Gules".*



(b) *"Paly of sixteen Sable and Or"*

Figure 2.10: *Two examples of short hand being used.*

Chapter 3

Lexical Analysis

Before parsing a Lexical analysis must be performed upon the input. The aim of Lexical Analysis is present an input in a way that makes it easier to parse. This involves taking an input and splitting it into a series of tokens using keywords as identifiers.

3.1 What is Lexical Analysis?

Lexical Analysis is more concerned with classifying sections of input as instances of things than it is understanding them; as such there are generally several kind of tokens produced by a lexical analysis, determined by the context of the application. Tokens can range from being a single letter to being a series of words, again this is context dependant.

In traditional compilation all of the basic programming control statements would have their own Token; if, else, while, etc. Tokens provide a basic abstraction level as in the sense that you can have a token representing an instance of a Tincture but the actual data about the Tincture is irrelevant at this stage, although it must be stored for later.

3.2 Lexical Analysis of Blazon

Lexically Analysing Blazon sentences is fairly straightforward, this is due to the large number of pre-defined elements present in the language. All the Tinctures, Partitions and Line Types are pre-defined as well as all the Geometric Charges and Honourable ordinaries.

Blazon also heavily utilises keywords as prefixes, "*Per*" always indicates the next word is a partition and any quantifier indicates a charge.

A simple Blazon sentence such as *Per Bend Embattled Or and Argent* would be translated by lexical analysis into the following tokens *Partition, Line Type, Tincture, Tincture*.

Storing the data for each of the tokens is simply handled by each token being a separate instance of an object of that type. So upon encountering the word "*Gules*", an instance of a Tincture object is created representing Gules.

For the sake of simplicity Geometric Charges and Ordinaries are to be considered in the same set of pre-defined Charges, whilst a Semi formal is to be treated as an edge case.

The only difficulty in lexing is dealing with Semi-Formal Charges. As they can be of any length and involve a large number of descriptive attitudes and positioning. A Semi Formal Charge declaration ends when either; the end of the sentence is reached, another quantifier and charge are declared or finally if two tinctures in a row are declared, one being for the charge and the other for the next field.

```

1: Lexically Analyse a given array of Strings which form a Blazon Sentence
2: Words(anarrayofstringseachofwhichisawordinablazonsentence)
3: Tokens[]
   {For every word}
4: for  $i = 0; i < Words.length; i++$  do
5:   if  $Words[i] = \text{"per"} \ \&\& \ Words[i+1]$  is a Partition then
6:     Create a new instance of a Partition using  $Words[i+1]$ 
7:     Add this instance of a Partition to the Tokens Array
8:     increment  $i$  by 1 to avoid lexing it twice.
9:   else if  $Words[i-1]$  is a quantifier  $\ \&\& \ Words[i]$  is a Charge then
10:    Create a new instance of a Charge using  $Words[i]$ 
11:    Add this instance of a Charge to the Tokens Array
    {If a quantifier is not followed by a pre defined charge then it should
    be treated as a semi formal}
12:   else if  $Words[i-1]$  is a quantifier then
13:    Create a new instance of a semi formal Charge
14:    Proceed through the array of string until the end of the charge dec-
    laration incrementing  $i$  each time.
15:    Add this instance of a Charge to the Tokens Array
16:   else if  $Words[i]$  is a quantifier then
17:    Create a new instance of a Quantifier using  $Words[i]$ 
18:    Add this instance of a Quantifier to the Tokens Array
19:   else if  $Words[i]$  is a LineType then
20:    Create a new instance of a LineType using  $Words[i]$ 
21:    Add this instance of a LineType to the Tokens Array
22:   else if  $Words[i]$  is a Tincture then
23:    Create a new instance of a Tincture using  $Words[i]$ 
24:    Add this instance of a Tincture to the Tokens Array
25:   end if
26: end for

```

Figure 3.1: Pseudo code that produces a series of Tokens by performing Lexical analysis on a given array of Strings.

3.3 More Involved Lexing

While the above pseudo code does provide an adequate and valid Lexical Analysis of a Blazon sentence, it is quite naive.

It is not particularly necessary to have an operate token for each LineType and Quantifier as both only occur in relation to another kind of token so

can not exist in isolation. Quantifiers only occur before a Charge so can be encompassed into an instance of a Charge and LineTypes only occur directly after a Partition or a Charge so can also be incorporated into the representation of those objects. It is also possible to reduce the occurrences of tinctures by wrapping Tinctures of Charges into the Charge object.

This would leave a system with only three different kinds of token which is considerably simpler than a system with five obviously it is *forty percent less complex*. This improvement isn't very hard to implement either although it does increase introduce a potential lookahead cost, where the Lexer needs to check words further on in the sentence then that of the one currently being index.

```

1: Lexically Analyse a given array of Strings which form a Blazon Sentence
2: Words(an array of strings each of which is a word in a blazon sentence)
3: Tokens[]
   {For every word}
4: for  $i = 0; i < Words.length; i++$  do
5:   if  $Words[i] = \text{"per"} \ \&\& \ Words[i+1] \text{ is a Partition}$  then
6:     Create a new instance of a Partition using  $Words[i+1]$ 
7:     Check for a Line Type and add to Partition instance
8:     Add this instance of a Partition to the Tokens Array
9:     increment  $i$  by 1 to avoid lexing it twice.
10:  else if  $Words[i] \text{ is a quantifier} \ \&\& \ Words[i+1] \text{ is a Charge}$  then
11:    Create a new instance of a Charge using  $Words[i+1]$ 
12:    Check for a Line Type and add to Charge instance
13:    Find the tincture for this charge and add an Tincture object to the
    charge instance.
14:    Add this instance of a Charge to the Tokens Array
    {If a quantifier is not followed by a pre defined charge then it should
    be treated as a semi formal}
15:  else if  $Words[i-1] \text{ is a quantifier}$  then
16:    Create a new instance of a Charge
17:    Proceed through the array of string until the end of the charge dec-
    laration incrementing  $i$  each time.
18:    Set the description of this charge to be the concatenation of all the
    words between the Quantifier and the ending Tincture.
19:    Add this instance of a Charge to the Tokens Array
    {By looking for Tinctures last it ensures that Tinctures for Fields
    are never mistaken for Tinctures for Fields}
20:  else if  $Words[i] \text{ is a Tincture}$  then
21:    Create a new instance of a Tincture using  $Words[i]$ 
22:    Add this instance of a Tincture to the Tokens Array
23:  end if
24: end for

```

Figure 3.2: Pseudo code for a slightly more complex lexical analysis that produces a simpler output with less tokens.

Although the algorithm is slightly more complex it simplifies parsing. By building line types and tinctures directly into the tokens it saves having to construct the more complicated data structures later on.

Chapter 4

Defining a Grammar for Blazon

Now that the language of Blazon has been fully defined the task of Parsing it can begin. Parsing a language is a fairly complicated task however there are several standard tools found throughout the field that simplify the problem by breaking it down into smaller more manageable tasks. The first goal is to express Blazon as a Grammar.

4.1 What is a Grammar?

A grammar is best defined as a series of formal rules that define the syntax of a formal language. Blazon is only a semi-formal language as charges provide components that are not predefined, however they are only a sub set of the language and can be handled as an edge case which can be included into the grammar.

A sentence of a Language must conform to that Language's grammar. To this end a grammar can be used to both produce sentences for a language but can also be used to validate that sentences conform to a Language. This is exactly what is needed for parsing.

There are four parts to a grammar. Firstly every grammar has a set of terminal or atomic symbols which are the building blocks of the language. Secondly a grammar needs to have a set of non-terminal symbols which represent a series of terminal symbols. Thirdly a grammar will have a series of production rules which will take a terminal symbol and turn it into a either terminal or non-terminal symbols or a combination of both. Finally a grammar must have a defined non-terminal symbol as a start symbol.

There are two types of grammar, context free and context sensitive the difference between the two is in the production rules. A context sensitive grammar can have production rules where there are both terminal tokens

and non-terminal tokens on both sides where context free grammars always uniquely have non-terminals on the left hand side of every production rule.

Grammars are also useful from an iterative development perspective as they can be built up over many iterations, which is exactly what I did.

4.2 Context free Grammars

Context free grammars are very simple but powerful tools for concise syntax definition. They form a top down hierarchy of non-terminal and terminal symbols which together can be used to generate every possible valid sentence in a language. Non-terminal symbols are symbols that have still to be fully evaluated whilst terminal symbols are atomic and can not be evaluated further.

Sentences are crafted by defining a series of production rules for the language. In a context free grammar a production rule takes a non-terminal symbol and translates it into a different symbol or series of symbols which may be a combination of terminal or non-terminal.

A Context has a designated start non-terminal symbol which provides the highest level of abstraction. The start symbol is evaluated by translating it using any of the production rules that take the start symbol. The output from the production rule now needs to be evaluated. This is done by taking the left most, head, non-terminal symbol, using a production rule to evaluate it and then repeating until no non-terminal symbols remain. Non-terminal symbols allow for a basic level of abstraction in a grammar as they may evaluate into any number of possible other symbols defined by the production rules of the language. The start symbol will be the most abstract and the terminal symbols will be well defined instances.

4.2.1 An Example Context Free Grammar

To demonstrate the power of a context free grammar I'm going to generate a few sentences for a language using a trivial grammar. I'm going to define a Context Free Grammar which will produce sentences for the language of Meals.

The start symbol for this grammar is *Meal*. There are three non-terminal symbols in the grammar, *Meal*, the start symbol, *Food* and *Drink*. The *|* symbol is used as an exclusive or so that multiple production rules can be compressed. *Drink* may evaluate to Either *Tea* or *Wine*.

An example of producing a sentence of the *Meal* language:

$$Meal \rightarrow Food + Drink$$

$$Food \rightarrow Steak \mid Potato \mid Fish \mid Salad$$

$$Drink \rightarrow Tea \mid Wine$$

Figure 4.1: Context Free Grammar of a Meal

Starting with the start symbol:

$$Meal$$

Using the production rule for Meal:

$$Meal \rightarrow Food + Drink$$

Results in:

$$Food + Drink$$

Evaluating the left most non-terminal using the production rule for *Food* which results in *Steak*:

$$Food \rightarrow Steak$$

Results in:

$$Steak + Drink$$

Finally evaluating *Drink* using the production rule resulting in *Wine*:

$$Drink \rightarrow Wine$$

Results in:

$$Steak + Wine$$

As there are no more non-terminal symbols remaining the production is complete.

Figure 4.2: Producing a Meal (easier than cooking)

4.3 Context Free Blazon

Obviously Blazon is a considerably more complex language than Meal. As such it requires a larger grammar to fully express it.

The grammar's start symbol is *Field* which represents the single Field that encompass the entire shield. The grammar needs production rules that allow a Field to be either Partitioned, with or without a line type, producing a Partition as well as a number of fields or Tinctured which produces a Tincture.

Partitions are all pre-defined to produce a set number of sub-fields depending on the number produced a partition can be evaluated into any of the appropriate terminals.

Tinctures can be evaluated into any of the valid Tinctures defined in Blazon which are all terminal symbols.

A Field can also evaluate into a Tincture and a Quantifier, which in turn evaluates into a Charge. It is possible to have more than one charge per field so a production rule is defined that evaluates a Quantifier into a Charge and another Quantifier, for the next charge. This can be used repeatedly to generate any number of charges.

Charges can belong into one of three categories, Ordinaries, Geometrics or semi formal. The Geometric charges are all pre-defined as are the Ordinaries. Semi-formal charges are the exception in that they can not be pre-defined exhaustively. However they follow the same syntactic pattern as regular charges defining the image first and ending in a tincture.

Line types are all predefined so evaluate into terminal symbols.

Blazon as a context free grammar

$Field \rightarrow PartitionOfTwo + LineType + Field + Field$
 $Field \rightarrow PartitionOfThree + LineType + Field + Field + Field$
 $Field \rightarrow PartitionOfFour + LineType + Field + Field + Field + Field$
 $Field \rightarrow PartitionOfTwo + Field + Field$
 $Field \rightarrow PartitionOfThree + Field + Field + Field$
 $Field \rightarrow PartitionOfFour + Field + Field + Field + Field$
 $Field \rightarrow Tincture$
 $Field \rightarrow Tincture + Quantifier$
 $Quantifier \rightarrow Charge$
 $Quantifier \rightarrow Charge + Quantifier$
 $Charge \rightarrow Charge + Charge$
 $Charge \rightarrow Ordinary + Tincture$
 $Charge \rightarrow Geometric + Tincture$
 $Charge \rightarrow Semi - formal + Tincture$
 $Ordinary \rightarrow Fess|Bar|Bend|BendSinister|Pale|Chief$
 $Ordinary \rightarrow Base|Cross|Saltire|Chevron$
 $Geometric \rightarrow Delf|Roudle$
 $Tincture \rightarrow Azure|Gules|Vert|Sable|Argent|Purpure|Or$
 $PartitionOfTwo \rightarrow Pale|Fess|Bend|BendSinister|Chevron$
 $PartitionOfThree \rightarrow Pall$
 $PartitionOfFour \rightarrow Cross|Saltire$
 $LineType \rightarrow Embattled|Wavy|Indented|Dancetty|Nebuly|Dovetailed$
 $SemiFormal \rightarrow ???$

Figure 4.3: Blazon as a context free grammer

- 4.4 Formatting Input
- 4.5 Tokenizing Strings

Chapter 5

Parsing

After Lexical Analysis is complete and the input string has been tokenised the next step is to generate a parse tree. Parsers produce a data structure that takes into account all the details of the language defined in its grammar and also attempt to perform error handling.

5.1 Parse Trees

A parser takes the output from lexical analysis and turns it into a data structure, as implied by the name this generally takes the form of a tree.

5.2 Parsing Blazon

Parsing Blazon involves taking the tokens generated from Lexical analysis and producing a well formed data structure that checks whether the input conforms to the defined grammar or not and attempts to report the reason behind why the input may be erroneous.

Blazon lends it self very well to a top down, left to right parsing method because of the way the language is defined. Fields are defined and Tinctured from the top-left most point downwards and mirroring this approach in parsing makes for a nice parallel.

The output from the lexical analysis is of a fairly high level because of the more advanced lexing approach taken which wrapped Line Types, Quantifiers and Tinctures into other objects when appropriate.

To build the data structure itself each of the objects to be instantiated have inbuilt class variables to allow them to act as nodes in a tree.

Partitions all have an array, the size of which is determined by the number of fields to be created for the particular type of partition. Tinctures have an

array of Charges so that they can have multiple charges placed upon them.

As Blazon sentences all implicitly start with an empty field that encompass the entire shield this is where the parser and parse tree will start. In the implementation the implicit field is represented by a partition which produces a single field, an *Escutcheon*.

The parser then takes the first token, according to the grammar the only valid operations that can be performed upon a field are Tincturing it or Partitioning it, therefore the parser checks whether this token is either a Tincture or a Partition and if it either of the above adds it to the initial field accordingly.

This process is repeated until either there are no more tokens left to parse or there is no more space in the tree structure or ideally both.

5.3 Parsing Example

The following demonstrates the parsing of the Blazon sentence *Per Bend Azure and Or a Cross Vert*.

The tokens generated by Lexical analysis would be

Tokens:

Partition (Bend)
Tincture (Azure)
Tincture (Or)
Charge (Cross)

Escutcheon
+Partition type = Escutcheon
+Number of Fields = 1
+Tree[] = An array of fields

Figure 5.1: *"The tree is full and there are no tokens left."*

Tokens:

Tincture (Azure)
Tincture (Or)
Charge (Cross)

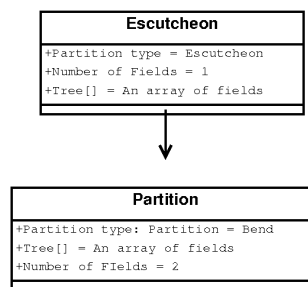


Figure 5.2: "The tree is full and there are no tokens left."

Tokens:

Tincture (Or)
Charge (Cross)

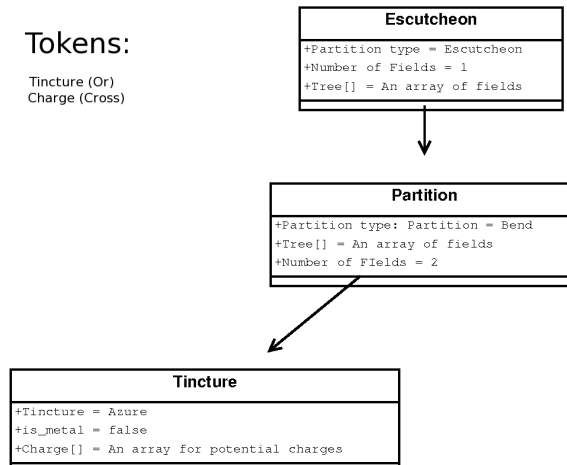


Figure 5.3: "The tree is full and there are no tokens left."

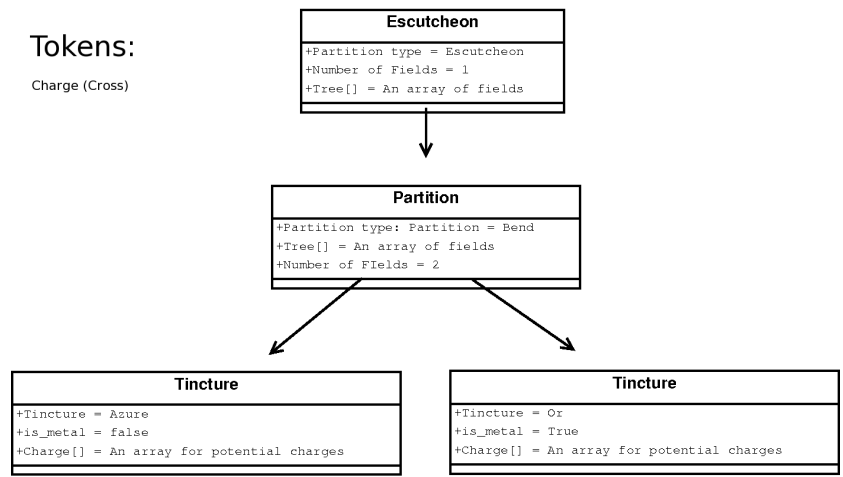


Figure 5.4: "The tree is full and there are no tokens left."

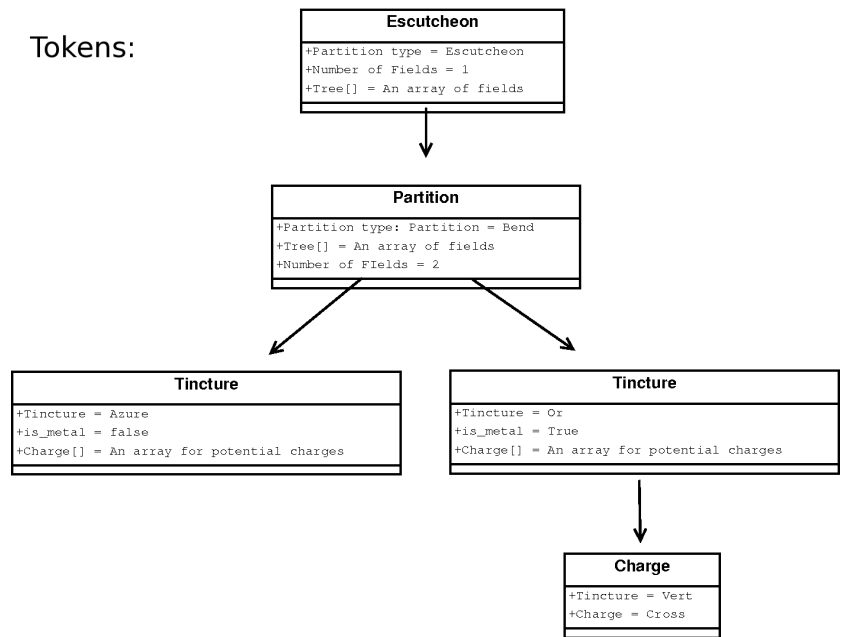


Figure 5.5: "The tree is full and there are no tokens left."

5.4 Error handling

If there are no more tokens but the tree has empty nodes in it then the Blazon sentence is invalid because at least one field has yet to be tinctured. If the inverse is true and there is no space in the tree for the next Token then too many field have been defined.

Chapter 6

Graphical Representations

Chapter 7

Design

Chapter 8

Testing

Testing is one of the most important factors of application design allowing developments to be thoroughly checked and regulated before being fully integrated into the application. To test the code behind the Blazon parser two main types of testing were used, Black Box testing and Regression Testing.

8.1 Black Box Testing

Black Box Testing involves testing an application without being able to view the source code. Obviously this is tricky to achieve as I was both writing and testing the application. Pseudo Black Box testing can be achieved by writing test cases before implementing any functionality.

Black Box Testing is very simple in practice. Before even starting to write code for a new feature of the application a series of test cases describing the expected behaviour for that feature must be fully defined.

Next a series of test are defined according to the expected behaviour for example, *"The application must warn the user when not enough tinctures are declared in a Blazon sentence for the number of fields"*.

Initially this seems fairly simple to test, the feature would be tested by providing the application with a Blazon sentence to parse that didn't tincture all of the fields it declared to ensure that this case would occur and that when it did the warning was presented correctly.

It is also necessary to test the feature to ensure the warning is not shown inappropriately. When the Blazon sentence is valid so all the fields are tinctured the warning should not show so a valid Blazon sentence should be another test case for this feature.

Finally the warning should also not show if the sentence is invalid for another reason, so a series of invalid Blazon sentences which are incorrect

but for a different reason should be provided to determine with a high degree of accuracy that the new feature does indeed work as intended.

After all the test cases have been determined, the feature is implemented. Once completed each of the defined test cases should be run in turn. If the feature achieves the intended behaviour as stated for a test case then the feature passes this test case. If the feature doesn't pass the intended behaviour for a test case then the feature hasn't been implemented correctly and needs to be altered so that it does pass this test case, altering the implementation of the feature means that all the test cases need to be ran again. This process is repeated until the feature has been implemented in such a way that it passes all the desired test cases.

Black Box testing lends itself very well towards an agile like iterative development approach to an application. Progressively adding more functionality in small quick iterations and fully testing each one will allow quick progress and make debugging a relatively painless experience.

8.2 Regression Testing

Regression Testing involves checking the integrity of the rest of an applications code base once a new feature has been implemented.

Implementing a new feature tends to change the way old code worked. These changes are normally minor, an extra parameter in a function call or another method in an object to help it interface with the new feature. It is just as important to test these little changes as it is to test the new feature. *Small changes break things*. Regression testing involves testing parts of the application that were all ready implemented and ensuring that a minor change hasn't broken anything.

An example of Regression testing being useful during the development of this application was when I added a *"hex"* property to my tincture objects. Because I was writing my project in JavaScript I had to implement my own clone functions which simply created a new instance of a Tincture and set all its properties to that of the original. Sadly I had changed the constructor of Tincture to take a hex value but forgot to change the clone function to incorporate this change and if you call a constructor and don't provide enough parameters in JavaScript the method still gets called and the missing values as null. The result was my cloned Tinctures having a null hex value and therefore when it came to actually drawing the colour a null pointer error was thrown but the error was not in my new code.

Fortunately I noticed that the tincture being passed to my drawing functions had a null value for element and re-tested my cloning functions which

now, were not producing the desired behaviour and therefore found my mistake.

Like Black Box testing, Regression Testing is very useful if not necessary for an application being developed in an agile manner. If used on a regular basis then it will defiantly help in the development of an application especially in regards to feature integration.

8.3 Blazon Seeking Missile

During the development of the project my flat mate *Johnny Houston* took it upon himself to write a python script lovingly named "*Blazon Seeking Missile*". The sole aim of this script was to break my project by producing very very large blazon sentences as test data.



Figure 8.1: A rendering of a very complicated Blazon sentence

This was achieved by utilising the context free grammar defined in Chapter two. Starting with a single field the script takes an integer as a command line argument and proceeds to build a balanced tree of the arguments depth, representing a Blazon sentence, treating tintured fields as leaves and partitions as nodes. The script randomly selects which partitions and tinctures to use.

This tool was very useful to me in testing my web application, even if it originally had more of a malicious intent. Essentially if my application can handle Blazon sentences containing 2¹¹ partitions then it will probably be able to handle more realistic input. It also proved the case that more

complex designs are not always better by producing a lot of horrible colour schemes and designs.

8.4 Application Level Testing

It is also important to test the application as a whole as well as individual components. Does it properly parse Blazon into a rendered Image? How is it known that the application does indeed handle the subset of Blazon it is supposed to? What happens if part of the language that application is not able to handle is presented?

8.5 Testing for Completeness of the Blazon Subset

The subset of the language that the application can parse includes:

1. Tinctures
2. Partitions
3. Geometric Charges
4. Ordinaries
5. Semi Formal Charges

The completeness of the parsing can be tested by examining the data structures outputted to the debug console in a browser. While the completeness of the drawing can be checked by simply looking at the result drawn on the HTML5 Canvas.

Bibliography

- [1] Mark E. Shoulson The pyBlazon wiki, <https://code.google.com/p/pyblazon/wiki/PartyPer> , Febuary 4th 2010
- [2] Michel Pastoureau. Heraldry, Its Origins and Meaning. New Horizons. Published in 1996, translated in 1997.
- [3] The language of Helaldry. <http://www.obcgs.com/charge1.jpg>
- [4] Author: Sodacan, The Wikimedia Commons, Description: A Heraldic Lion in Rampant Attitude http://upload.wikimedia.org/wikipedia/commons/7/75/Lion_Rampant.svg 13 April 2009
- [5] Author: Sodacan, The Wikimedia Commons, Description: A Heraldic Lion in Passant Attitude, http://upload.wikimedia.org/wikipedia/commons/f/f3/Lion_Passant.svg , 13 April 2009
- [6] Adapted from: Phelan and Vanora, March 1 1989, <http://khevron.tripod.com/armorial/awards.html>