Creating a Compiler for the Semi-Structured Language of Blazons

Michael Mulder
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
m.mulder-4@student.utwente.nl

ABSTRACT

A blazon is a textual, semi-structured description of the visual representation of a coat of arms. There is no formal tool to create such a visual representation given a blazon, yet there is a lot of demand for these drawings. In this project we research all steps needed to create a compiler that can turn blazons into their visual representation. Our focus during this research was creating a flexible and easily extendable compiler that can understand the different ways a blazon can be expressed.

Keywords

Blazonry, Semi-parsing, Semi-structured languages, Compiler Construction, Domain Specific Languages

1. INTRODUCTION

A *blazon* is a formal semi-structured textual description of a coat of arms, flag or similar emblem, from which the reader can reconstruct the appropriate image. *To blazon* is to create such a description. *Blazonry* is the art of creating such blazons, linking textual semi-structured descriptions of coats of arms to their visual representation.

There is quite an active online community of heraldry and blazonry enthusiasts. This community includes researchers researching medieval heraldry, people looking to learn more about family crests, and general enthusiasts. In this community there is a demand for the drawing of coats of arms. Many older coats of arms only have small, bad quality pictures, or even only a description. If someone want to get a better illustration they have to either learn how to (digitally) draw or they have to post a request on an online board in the hope that someone else will pick it up and provide them with an illustration. Our goal is to create an online tool which can help with this process. This will eliminate the need of users to have to learn image processing or having to ask someone to draw their shield. Instead the user will be able to enter a blazon into the tool and create their own drawing.

In this project we will research different aspects needed for the creation for this tool. To guide this research we have determined the following research questions:

34th Twente Student Conference on IT Jan. 29^{nth}, 2021, Enschede, The Netherlands.

1. How can we model the domain of blazonry as a domain-specific language, using software language engineering methods?
2. What are the rules of blazonry that could be seen as typing rules or its static semantics?
3. How can we construct a semi-structured, error-correcting parser of the blazon language?
   (a) How can we maximize the usefulness and informativeness of the parser's error messages?
   (b) How can we maximize the set of accepted inputs?
4. How can we create a code generator that can draw Scalable Vector Graphics [3] coats of arms given a parsed blazon?

We will start this paper be explaining the methods we used to get our results in section 2. The result we found using these methods will be discussed in section 3 until section 7. In section 8 we will discuss the effectiveness of our approach. section 9 will discuss possible future work and what obstacles we foresee. Lastly we will answer our research question in section 10.

All the code discussed in this paper can be found on GitHub.

2. METHODOLOGY

The goal of this research is to create a complete compiler. To create this we will follow general steps to create a compiler [1,5]. We will start by defining the language we wish to parse. Next we will write a scanner and parser, design an intermediate representation, and write a code generator. We will also introduce a static checker that checks the intermediate representation for typing rules. These steps closely correspond to our research questions and will also correspond to the structure of this document.

We will start with the language definition in section 3. The definition of the language is an important basis on which every next step will be built. To do this we analyzed the domain of blazonry using literary research. This step corresponds to our first research question.

Next we will create a scanner and parser, as detailed in section 4. We started by researching ways of semi-parsing, as can be read in subsection 4.1. After the research of the background we attempted to apply these principles to create our own parser, as detailed in subsection 4.2. The part of the process will answer the third research question.

Our parser has to parse the input to some form of intermediate structure, which is also used by the static checker and code generator. We designed and edited this intermediate

structure while working on the other parts of the compiler. This was a continuous process because as the other parts developed the requirements for the intermediate representation changed. Our final design will be discussed in section 5.

The final essential step of a compiler is the code generator. In our case we will generate an SVG, this will answer our fourth research question. To create a code generator we researched ways to describe different parts of the shield in the SVG language and looked for the best way to do so programmatically. The process of how we did this is detailed in section 6.

In order to improve the usability of our tool, and to answer the second research question, we will also create a static checker that checks the intermediate representation to see if it follows the rules of blazonry. To answer this question we continued our analysis of blazonry into the domain of heraldry to find these rules. Next we defined these rules in a way that a static checker can check them and built this checker. These rules and the description of the checker can be found in section 7.

3. THE DOMAIN OF BLAZONRY

The language of blazonry is based on English, but with an uncommon word order in some places, and with French words dominating its vocabulary. We recognize 7 colors and 2 furs. The colors are divided into 5 tinctures and 2 metals. It is a general rule of heraldry that a shield is not supposed to have tinctures upon tincture or metals upon metals. So if the background is blue, anything placed on the background must be colored as a metal, anything that is in turn placed upon that must again be colored as a tincture. See Table 1 for an overview of the colors, also see Figure 6b and Figure 6c for depictions of the furs. We'll use color to mean any tincture, metal, or fur, unless specified otherwise. We'll write the names of colors with capital letters for clarity, this is not required in real blazons.

Heraldic name	English name
Metals	
Or	Gold / Yellow
Argent	Silver / White
Tinctures	
Azure	Blue
Vert	Green
Gules	Red
Sable	Black
Purpure	Purple
Furs	
Ermine	White with black spots
Vair	White and blue 'bells'

Table 1: The different colors

Each shield starts with a background, called the "field". A field can be a either be one color or divided into multiple colors. There are a few words that specifically denote a division. For example "Quarterly Azure and Or" is a field divided into quarters where the first and third quarter are blue and the second and fourth are yellow or gold. A division can also be expressed as "per <ordinary>". We'll go into ordinaries in the next paragraph. A bend is an ordinary that is a band from the top left to the bottom right of the shield. A division "Per band" is a field divided by a line from top left to bottom right. "Per bend Vair and Or", as shown in Figure 7a, shows this division. The word "party" can also be used to indicate a partition, this changes nothing in the meaning. So instead of "Per pale Vair Or" one could also say "Party per pale Vair Or" and achieve the exact same result.

After the field the ordinaries will be named. Ordinaries are a number of standardized shapes that can take a main position on the shield. We already mentioned the bend before, that is an excellent example of an ordinary. In Figure 8a we show the bend and in Figure 8c the pale. Figure 8b we also show the word "sinister", which is a modifier word that flips the figure that precedes it. We can also use this word to flip the division of the field, see Figure 7b. Furthermore it is also possible to specify special partition lines. Because there was no time to implement this in our code-generator we will not go into the different possibilities.

After the field and ordinaries figures can be added on top, we call these "charges". Charges can be smaller variants of the ordinaries, like a bendlet is a smaller variant of the bend, or figures, like flowers or animals. There are a number of standard figures that are widely recognized, but in theory the amount of figures in unlimited. In general weapons, animals, crosses and human arms are quite common. Each of these figures can be described into more detail if needed. Take animal figures for example, "displayed" means spread wings, "passant" means walking, etc [4, 6].

4. IMPLEMENTATION OF PARSING

4.1 Background on Semi-Parsing

In the domain of semi-parsing a number of different approaches have been developed [9]. For our application we decided to look into *iterative lexical analysis* [2] and *hierarchical lexical analysis* [7,8]. Both approaches use lexical analysis to generate source code models.

Lightweight Source Model Extraction (LSME), [7,8], uses hierarchical regular expressions to extract unit level models. Iterative Lexical Analysis (ILA) [2] also uses a hierarchical approach to regular expression matching. But while LSME was designed to build high level models, ILA is more focused on building low level models. Another point where they differ is that ILA requires the user to provide definitions for the base level tokens used in their scanner; while LSME breaks the input stream into two token types: identifiers and single characters. This means that a larger specification is needed for ILA, but this gives more control over the scanner. Lastly, the iterative nature of ILA allows it to find arbitrarily deeply nested constructs.

We based our idea of the parser on the approach used for ILA. So we'll summarize the basics of their approach.

ILA encodes tokens as strings. It uses pattern matching to iteratively replace the basic tokens with generalizations until the stream has been completely parsed. In the paper by A. Cox and C. Clarke [2] a parser for C has been created, called ILA_C. In this implementation 8 different levels have been defined. The input goes through the levels one by one, sometimes being sent back. This process is visualized in Figure 2. Levels 1, 3, 4, and 5 use standard pattern matching, levels 2 and 6 are there for elements needing longer matching at level 1 or levels 3, 4, 5 respectively. Level 7 makes relatively low risk matches for ambiguous text and level 8 makes high risk matches for ambiguous text. Figure 1 shows an example of how ILA parses a function definition. In each iteration a part of the stream gets replaced by a higher order token until a function definition is found.

```
-1       int zero () { return (0); }
 0   int ident ( ) { return ( cnst ) ; }
 1  dspec name ( ) { return ( expr ) ; }
 2      dspec declr { return expr ; }
 3           dspec declr { stmt }
 4             dspec declr block
 5                  fundef
```

Figure 1: Example Substitution Sequence ILA_C [2]

4.2 Implementation

As said in subsection 4.1 we started by basing our implementation on the principles of Iterative Lexical Analysis [2]. This means that we will be parsing from the bottom up and we intended to mainly use regular expressions to find structures. Because we use bottom up parsing, any errors will be propagated to the top of the parse tree. Because the goal of this research is to research how semi-parsing might be used for a semi-structured language like blazonry, we choose our programming language based on personal experience instead of efficiency. The lead us to build our compiler in PHP 8.

Our first concern was to define our tokens and build a scanner. We programmatically create regular expressions from an associative array which we use to create our first list of tokens. This step mirrors what ILA_C does in line 0 in Figure 1, and can be seen as step 1 in Figure 3. All undefined words get saved as the token type `STRING`. This is to make sure that our scanner will always transform all words into tokens and nothing gets lost.

In order to keep our parsing rules simple, our next step is to generalize tokens. By this we mean that, for example, all colors get transformed into the token `COLOR`, or that both "per bend" and "quarterly" become get transformed into the token `PARTITION`. This step is also entirely done by regular expression matching.

Our next concern was to parse a field declaration. Because we do not want to have to define any parse rules directly as regular expressions we started to run into the limitations of our parsing approach. A regular expression for a field declaration would be '`(COLOR)|(PARTED? PARTITION SINISTER? PARTITION_LINE? SINISTER? COLOR AND? COLOR)`', this is without allowing for commas or other possible small deviations. So we started looking for another approach.

We will explain our approach using Figure 3. The gray squares indicate that the node did not get edited by this level, but they are repeated for clarity. Here we see that after the first two levels of parsing we are left with tokens `PARTITION COMMA COLOR AND COLOR`.

Our approach to parsing the field starts by reading the first token, `PARTITION` in this case. We have defined that a partition requires two colors, so the parser will continue until it has either found two colors or reaches the end of the blazon, in which case it will report that no field has been found. We have also defined that tokens like `COMMA` and `AND` hold no information, so it will not save these tokens in our parse tree.

The next step in parsing is parsing the ordinaries. An ordinary definition is in the shape of "`NUMBER? ORDINARY SINISTER? PARTITION_LINES? COLOR`". There are, however, a number of variations possible in defining ordinaries. For example, "`Azure, a bend and pale Or`" is the correct way to denote two ordinaries of the same color. This complicates our parser and makes it very hard to define a regular expression that encapsulates will match will all ordinary definitions while also catching all the information. Our current program is not able to parse these more complicated blazons. It uses matching on the regular expression expressed before.

5. INTERMEDIATE REPRESENTATION

Our parser creates an intermediate representation that gets passed on to the rest of the compiler. The goal of this representation is to have all the information that the parser extracted from the source in a way that the rest of the compiler can work with it. We choose to keep our intermediate representation simple. We created an abstract Node class which gets implemented by a classes called Term and NonTerm which are meant for terminal and non-terminal nodes, respectively.

In Figure 4 the intermediate representation of the blazon 'Per bend, Azure and Or' has been depicted. If we compare this figure to the parse tree of the same blazon, depicted in Figure 3, we see that the intermediate representation follows the structure of the parse tree. It is, however, more compact and has left out the tokens that do not hold any new information, like the comma. Figure 5 depicts the intermediate representation of a longer blazon with a sinister division and an ordinary. This has been included to show how an ordinary, or any other extra part, is represented in the intermediate representation. This figure also shows how modifiers are stored, in this case the sinister modifier on the division of the field.

The intermediate representation class is also where we collect all error messages that get raised during parsing.

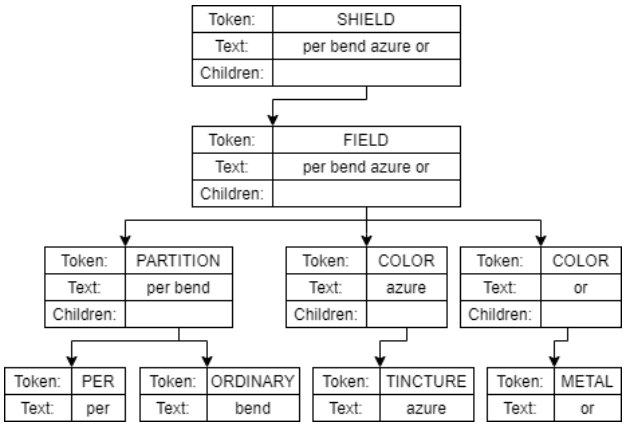


Figure 4: Intermediate Representation of the blazon 'Per bend, Azure and Or'

6. IMPLEMENTATION OF GENERATOR

We use SVG [3] to build the images of shields. This makes it possible to programmatically draw our shields layer for layer.

The generator builds the shield layer by layer. It start by creating the needed XML for an SVG of an empty shield. Next it will determine the background color of the shield. This can be all that is needed for the field, as in Figure 6a and the figures in Figure 8. If the background is a fur that means for the generator that the background is white and that it will have to add a fur pattern on top.

There are a number of approached to use a pattern on a field in SVG. The easiest approach for this application turned out to be to programmatically place the fur 'shape' on the shield using a loop to cover the field and adding a

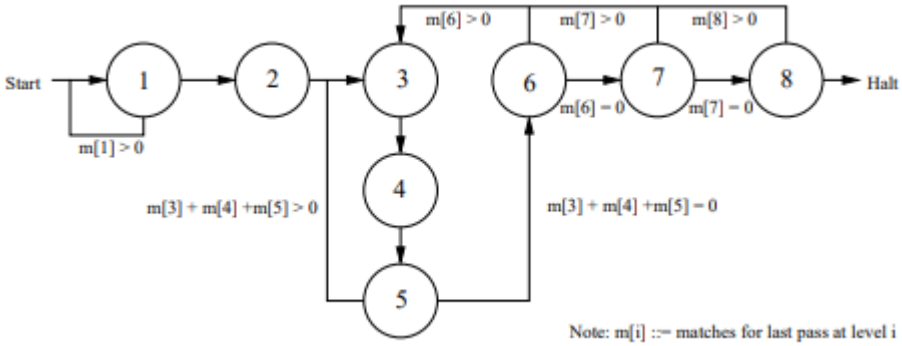


Figure 2: Iteration profile for ILA_C [2]

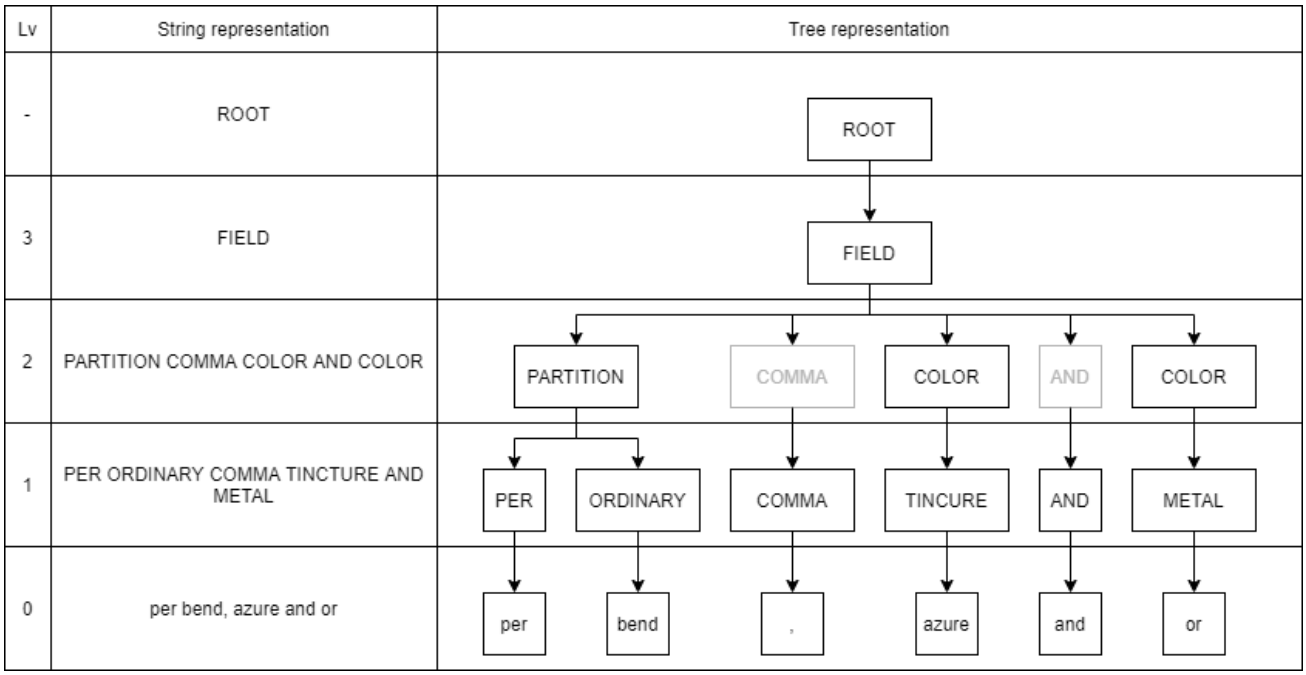


Figure 3: Parse tree for "Per bend, Azure and Or"

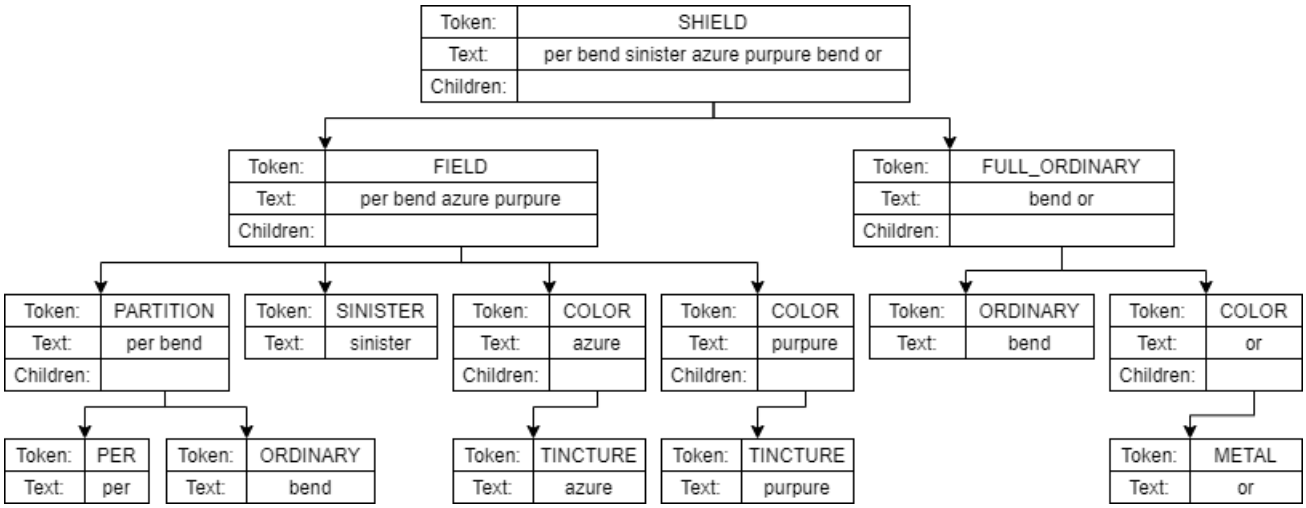
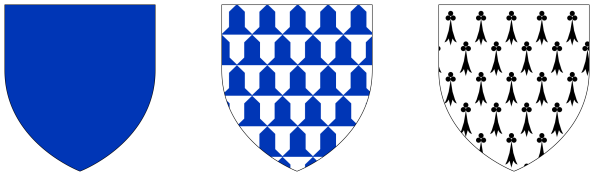


Figure 5: Intermediate Representation of the blazon 'Per bend sinister Azure and Purpure a bend Or'

mask on top to hide any part that is outside the shield. The furs are depicted in Figure 6b and Figure 6c.



(a) Azure (b) Vair (c) Ermine

Figure 6: Three examples of generated fields of one `COLOR`

The next layer of the shield is still part of the field: the possible second half of the partition. At the moment of writing our generator recognizes the divisions "per bend" and "per pale", shown in Figure 7a and Figure 7c respectively. The partition is implemented as another shield on top of the first one, now using a mask to make sure only half of the original shield gets hidden. We have also defined the sinister modifier, which flips the way the shield is divided. See Figure 7b for the sinister version of Figure 7a. This is implemented using a transformation on the mask. Sinister has no effect on the division "per pale". In order to make sure that the sinister transformation did not get applied to this division we had to separately define which divisions can be flipped by sinister. If you were to apply the sinister transformation to "per pale" the colors would get flipped, which is not what is intended.

(a) Per bend Vair and Or (b) Per bend sinister Vair and Or (c) Per pale Vair and Or

Figure 7: Three examples of generated fields with divisions

After the complete field has been parsed we can start with the figures. The ordinaries are below the charges and are placed in the order that they are mentioned. We limited the ordinaries to the bend and pale to show the generation. They are depicted in Figure 8a and Figure 8c respectively. The sinister modifier is also available on ordinaries. We can use the same methods as we used for making the divisions sinister. The result can be seen in Figure 8b.

(a) Azure, a bend Or (b) Azure, a bend sinister Or (c) Azure, a pale Or

Figure 8: Three examples of generated fields with simple ordinaries

7. STATIC CHECKER IMPLEMENTATION

Heraldry has a number of rules of what is allowed on shield and what not. For each of these rules it is of course also possible to find an official coat of arms that breaks it. It is however generally considered bad design if a shield does. In order to improve the shield created with our compiler we set out to implement a static checker that would check our intermediate representation to see if it breaks any of these rules.

One of the most basic rules of blazonry is that a tincture cannot be placed on a tincture and a metal not on another metal. Furs, on the other hand. can be placed on either a metal or tincture, or even another fur. This also goes for anything that is emblazoned 'proper'. Proper in this context means that it is depicted in its original colors, or as close as possible. In designing a coat of arms it is however usual to treat a fur as either a metal of tincture, depending on its background. It is also rare to find a charge emblazoned as proper but depicted in a recognized heraldic color on a background of tincture, in such cased care should be take to emblazon the charge as proper and not the heraldic color.

A charge that is composed of a tincture and a metal may be placed upon a field of either. In these shields it is however common to ensure that the first color of the composite charge is in opposition of the field that it is placed on. For example "Or, a fess chequy Azure and Argent", a golden shield with a horizontal line blocked in silver and blue, is a correct blazon. To blazon, or depict this shield as "Or, a fess chequy Argent and Azure" would however be incorrect. In the second blazon the blocks of silver and blue would be switched. An ordinary placed on a background of both metal and tincture, can be of metal, tincture, or fur. This is not considered an infraction on the basic rule of not having a tincture on tincture or metal on metal.

If the field is divided into an equal number of pieces it may be composed of two metals or two tinctures. This is because all the pieces are equal and of equal number so none can be considered as placed upon another. There are also some special cases, the fields emblazoned 'landscape', 'water', and 'masoned'. These are said to fall outside of the categories mentioned.

We have implemented a static checker that checks the color combinations that are possible with our current parser and generator implementations. It adds an error to the intermediate representation if it finds an invalid combination.

8. EVALUATION

One of the biggest challenges is writing a compiler for a semi-structure language is the parsing step. In this paper we have tried two different approaches. We based our first approach on ILA [2] and we have tried extending that approach by writing a specific parser for one of the steps. There are benefits and disadvantages of both approaches, which will be discussed now.

We deviated from the original approach of ILA because we felt that we didn't have enough control over the matching and a possible rollback. We also did not like the idea of directly writing regular expressions. The lexical component is very suited for matching simple expressions, like recognizing a tincture as a color. A downside of this approach is that it makes it harder to create helpful error messages because the regular expression either matches or not, there is no indication of an almost match. The main issues with this approach could possible be solved by creating a custom regular expression matcher, in which you

could have more control on what matches and on what doesn't match and gives an error.

For our specific parser we attempted to write a smart parser that could find and parse any field declaration. This approach gave us some flexibility that we missed in our ILA implementation, but it also introduced new problems. In writing a custom parser with the variability needed for semi-structured languages it can be quite hard to account for all possibilities. Especially all the ways a wrong input can be structured. A benefit of this approach is that we were able to make our error messages more specific.

Another part of our compiler is our static checker. This checker is intended to check the intermediate representation generated by the parser to check if the rules of heraldry a being complied to. Our checker checks the color of the field in combination of the color of the ordinary. These rules can become quite difficult to implement when the shield gets more difficult. Think for example of ordinaries on top of ordinaries, or charges that are partly on an ordinary and partly on a field. Our approach works perfectly for our relatively small set of options and could be extended for all future rules. Anyone who wishes to do so might however consider extending the intermediate representation to reduce some of the complexion of the checker.

Lastly, there is the code generator. Seeing as the goal of blazonry is to describe the coat of arms step by step, layer by layer, it is not surprising that it lends itself beautifully to this kind of code generation. The main obstacle would be the massive amount of different shapes that can be used. We envision that in extending the code generator it would be wise to design a maintainable structure to save all these shapes. 1

9. FUTURE WORK

Our goal of creating a flexible compiler with good usability features is still a long way ahead. In this study we have collected a lot of knowledge on blazonry as well as on what parsing methods work or don't work at all.

In order to achieve our ultimate goal we obviously need to add charges to our compiler and extend our library of divisions and ordinaries. There is, however, also a lot of research that still can be done on the parts that have been implemented.

A lot more research can be done on the parsing of blazons. We currently use a different parsing method for field than ordinaries. The field has his own specialized class, while the ordinaries use a regular expression to match. The specialized class gives more freedom in defining the behavior of the parser, but is harder to correctly implement. In the basics it is very easy to implement, but it is also very bug-prone. A regular expression on the other hand is more difficult to write initially. It requires quite some initial knowledge on regular expressions and it is very easy to make typing or spelling errors. ILA [2] shows us however that it is possible to achieve complicated patterns using regular expression matching. This will likely involve the implementation of a system that has more control over the regular expressions and how they match than our current implementation has.

In section 7 we have detailed some of the static rules of heraldry and blazonry. These are only the rules on which color can be placed where, as one might imagine there are a lot more. We would like to do more research on these rules, which is maybe more in the domain of history or vexillology. It would also be interesting to research the usability.

A big part of the goal we envision for this tool is its usability and guiding error messages, together with helping the user create the best shields possible. Research into how to best present any compiler messages would make this task a lot easier. Combining that with research into how people use blazonry nowadays and what are common mistakes that cause the result to be different than they envisioned would greatly improve the usability of our envisioned tool.

10. CONCLUSION

To structure the conclusion we will recall the research questions we set out to answer.

1. How can we model the domain of blazonry as a domain-specific language, using software language engineering methods?

We have started modeling the domain of blazonry into a domain-specific language, as we detailed in subsection 4.2. Since this is a large undertaking, we have limited our current model to the field and a simplification of the ordinaries.

2. What are the rules of blazonry that could be seen as typing rules or its static semantics?

We have found a number of rules of blazonry and heraldry that could be seen as typing rules our its static semantics. For our limited model there are only rules about which colors can be combined, for which we have implemented a checker, as described in section 7. As the model gets extended, the static checker should also be extended to include the added possibilities for color combinations next to new rules that will come into play.

3. How can we construct a semi-structured, error-correcting parser of the blazon language?

(a) How can we maximize the usefulness and informativeness of the parser's error messages?

(b) How can we maximize the set of accepted inputs?

In order to maximize the set of accepted inputs of the parser a choice has to be made to create a very flexible parser, which is error-prone in developing, or create special rules for every edge case, which is a lot of work to write and most likely will not be enough to catch all edge cases. We have chosen to try to create a parser flexible enough to catch everything. Some headway has been made, but no optimal solution has yet been found. Our considerations are in subsection 4.2. Error-correction of blazons is quite complicated, in order to fully achieve this more research, as detailed in section 9 is needed.

4. How can we create a code generator that can draw Scalable Vector Graphics [3] coats of arms given a parsed blazon?

The model of blazonry lends itself quite well to the creation of a code generator that build SVG coat of arms. This is because a blazon is meant to describe the coat of arms layer by layer and SVG allows us to follow this approach for building the image.

11. REFERENCES

[1] K. Cooper and L. Torczon. *Engineering a Compiler*. Elsevier Ltd, 2012.

[2] A. Cox and C. L. A. Clarke. Syntactic approximation using iterative lexical analysis. In *11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA*, pages 154–164. IEEE Computer Society, 2003.

[3] E. Dahlström, P. Dengler, A. Grasso, C. Lilley, C. McCormack, D. Schepers, J. Watt, J. Ferraiolo, J. Fujisawa, and D. Jackson. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. W3C, Aug. 2011. https://www.w3.org/TR/SVG11/.

[4] A. C. Fox-Davies. *A Complete Guide to Heraldry*. T.C. & E.C. Jack, London, 1909.

[5] D. Grune, K. van Reeuwijk, H. E. Bal, C. J. H. Jacobs, and K. G. Langendoen. *Modern Compiler Design*. Addison-Wesley, second edition, 2012. https://dickgrune.com/Books/MCD_2nd_Edition/.

[6] The Heraldry Society. *Historic Heraldry Handbook*, 2018.

[7] G. C. Murphy and D. Notkin. Lightweight source model extraction. In G. E. Kaiser, editor, *SIGSOFT '95, Proceedings of the Third ACM SIGSOFT Symposium on Foundations of Software Engineering, Washington, DC, USA, October 10-13, 1995*, pages 116–127. ACM, 1995.

[8] G. C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Trans. Softw. Eng. Methodol.*, 5(3):262–292, 1996.

[9] V. Zaytsev. Formal foundations for semi-parsing. In S. Demeyer, D. W. Binkley, and F. Ricca, editors, *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014*, pages 313–317. IEEE Computer Society, 2014.