Joakim Omberg Lier

Applying Language Technology in Error Detection for Optical Music Recognition

Master's thesis in Computer Science

Supervisor: Øystein Nytrø

June 2020

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Norwegian University of
Science and Technology

TDT4900 - Computer Science, Master's thesis

Applying Language Technology in Error Detection for Optical Music Recognition

Joakim Omberg Lier

Advisor
Øystein Nytrø

June 10, 2020

Problem Description

The objective of this work is to explore how language theory and technology can improve the quality of current methods in optical music score recognition. The scope of the thesis ranges from review of relevant language models and theory to concrete experiments in parsing, error detection and translation.

A concrete goal is to validate output from from the Optical Musical Recognition system of Calvo-Zaragoza and David Rizo using the PRiMuS dataset for testing, characterize areas of potential improvement and propose, implement and test selected solutions. Finally, a study of different kinds of formal languages is to be conducted in order to see if the entire language definition may be embedded in a formal grammar.

Summary

Etter fremveksten av teknikker som benytter dyp læring har Optical Music Recognition-løsninger som forsøker å lese og forstå noter sett store fremskritt. Teknikker som benytter dyp læring er ofte vanskelig å validere, noe som er et viktig skritt for å sikre korrektheten til en løsning. Det er vanlig å validere slike løsninger ved å sammenligne resultatene med håndlagde eller datagenererte fasiter. Dette krever at datasettene er av høy kvalitet, og at alle fasitene er korrekte.

Ved å benytte veletablert teknologi som språkteori og kompilatorkonstruksjon kan en tilnærming som modellerer problemet være mer nøyaktig og mindre arbeidskrevenede enn den vanlig tilnærmingen innen dyp læring. Ved å modellere resultatdomenet kan man oppnå både lett og effektiv bekreftelse. Dette fjerner behovet for å skape fasiter for ethvert tilfelle. En ekstra fordel med dette er at det fjerner feilkilder, for eksempel feiltrykk i notene eller at testene ikke dekker alle tilfeller.

En eksisterende grammatikk fra tidligere arbeider ble gjort om og en kompilator front-end ble implementert. I tillegg ble det også utført en studie av formelle språk for å finne nye og elegante måter å utvide grammatikken på. Dette ga gode resultater. Kompilatoren ble testet på datasettet PRiMuS og viste en feilrate på bare 0,14 %, som betyr at det er en god representasjon av det resultatdomenet. Noen av disse feilene viste seg å være tidligere urapporterte feil i datasettet som nå kan fjernes eller fikses.

Abstract

With the rise of deep learning techniques, Optical Music Recognition software that aims to read and understand musical notation has seen great progress. However, deep learning techniques are often difficult to validate, which is a crucial step in order to ensure the correctness of a solution. It is common to validate deep learning solutions by comparing results to hand crafted or computer generated ground truth representations. This requires the quality of the data sets to be very high, as well as every ground truth representation to be correct.

By utilizing well established technology like language theory and compiler construction, a modelling approach can prove to be more accurate and less labour intensive than the common practice in deep learning today. Creating a model of the result domain may lead to both easy and efficient verification. This eliminates the need for creating ground truth representations to cover every scenario. An added benefit of this is that it removes many sources of errors, such as misprints and lack of coverage in the tests.

An existing grammar was reworked and a compiler front-end was implemented. Additionally, a study of formal languages was also conducted in order to find new and elegant ways of extending the grammar. This yielded good results. The compiler front-end was tested on the PRiMuS dataset and exhibited an error rate of only 0.14%, which indicates a good representation of the intended language. Some of these errors turned out to be previously unreported errors in the data set, which can now be removed or fixed.

Preface

The idea behind this thesis came to me the fall of 2019 at a classical music performance, the semester before I started this thesis. This is where the basic idea of parsing music using a formal language came from, and it has stuck with me ever since. The initial idea consisted of creating my own system to read musical notation and then parse it, but I later discovered that this is part of the field of research called OMR. Understanding the complexity and scope of my initial idea, it was gradually narrowed down into this thesis.

I'd like to extend my gratitude to the people who have helped me. First of all, thanks to my advisor, Øystein Nytrø. Secondly, thanks to Ida Maria Henriksen Borgen for helping me with academic writing. Finally, thanks to Anne Cathrine Elster for letting me keep my place in the lab till the Covid19 situation forced the campus to close.

Dedicated to Norvald Omberg

Table of Contents

Problem Description i

Summary i

Abstract ii

Preface iii

Table of Contents vii

List of Tables ix

List of Figures xii

List of Listings xiv

Abbreviations xv

1 Introduction 1
- 1.1 Goals 3
- 1.2 Report Outline 4

2 Background 5
- 2.1 Formal Languages 5
  - 2.1.1 The Chomsky Hierarchy of Formal Languages 7
  - 2.1.2 Other Languages 8
  - 2.1.3 Parsing 8
  - 2.1.4 Notation 9
- 2.2 Compilers 10
  - 2.2.1 Compiler pipelines 10
  - 2.2.2 Compiler development tools 12
- 2.3 Musical Notation 15

2.3.1 The Symbols of Musical Notation 15
2.3.2 Taxonomy of Musical Scores 18
2.4 Optical Musical Recognition 18
2.4.1 Defining OMR and OMR Solutions 19
2.4.2 The OMR Pipeline 20
2.5 The PRiMuS Dataset 22

3 Result Validation in OMR **23**
3.1 Literature Review on Validation Techniques in OMR 23
3.2 Automatic Composition of Musical Scores 25
3.3 Conclusions 27

4 Experiment and Software Design **31**
4.1 Experiment Design 31
4.2 Software Design 34
4.3 Summary 36

5 Reworking the Original Grammar **39**
5.1 Representing Musical Notation 39
5.1.1 Motivation 40
5.1.2 Restrictions on the language 42
5.1.3 Adding New Symbols to the Reworked Grammar 46
5.2 Semantic Analysis 53
5.2.1 A Syntax Directed Translation 53
5.2.2 Constraints 55
5.3 Applicability of Other Formal Languages 59
5.3.1 Criterion and Evaluation 59
5.3.2 Context Free Grammars 60
5.3.3 Context-Sensitive Grammars 62
5.3.4 Van Wijngaard Grammars 64
5.3.5 Definite Clause Grammars 66
5.4 Insights from Language Study 68

6 Results **71**
6.1 Experimental results 71
6.1.1 Syntax Errors 72
6.1.2 Constraint Check Errors. 72

7 Conclusion **75**
7.1 Discussion 75
7.1.1 Examining the Syntax Errors 76
7.1.2 Errors in the PRiMuS Data Set 77
7.1.3 Examining the Constraint check errors 77
7.2 Conclusions 78
7.3 Future Work 79

Bibliography **81**

Appendices **87**

A Complete Grammars **89**
A.1 Original Context Free Grammar for Semantic Encoding in EBNF 89
A.2 The Complete Reworked Grammar 91

B Complete List of Misprinted Musical Incipits Found in the PRiMuS Data Set **93**

List of Tables

3.1 Condensed literature review. Each paper is presented with its employed techniques as well as validation schemes. 26
3.2 Overview of significant literature regarding algorithmic composition with a focus on grammar based approaches 28

6.1 Overview of error rates for the different grammars. The total number of incipits in the dataset is 87678 71
6.2 Distribution of syntax errors from the experiments using the Excerpt grammar. 72

List of Figures

1.1 Musical notation incipit 000051652-1_2_1 from the PRiMuS data set . . 2

2.1 A tree representation, or abstract syntax tree, for the simple sequence of production rules presented in Section 2.1 7
2.2 The Chomsky hierarchy showing the relationship between the four types of chomskian languages 8
2.3 An overview of the components of a compiler 10
2.4 A more detailed look at a compiler front-end 11
2.5 An example of musical notation 15
2.6 Two notes with different graphical properties determining both pitch and duration. The note head is indicated in red and the stem is indicated in blue. 16
2.7 Notes and rests of different common durations. Notes are on the top line and rests are on the bottom line. Each vertical pair has the same duration. 17
2.8 An illustration of the first OMR pipeline by Bainbridge et al [6] 20
2.9 The updated and improved OMR pileline as defined by Rebelo et al [50] . 21

3.1 A possible interpretation of a turtle curve generated by an L-system. Notes are numbered to clearly show the interpretation 27

4.1 Development of the musical notation compiler, MNC 33
4.2 Intermediate goals of the development process for MNC. 35
4.3 Diagram showing the data formats after each step as well as each type of possible error. 37

5.1 The musical notation example from Section 2.3 with a transcription. Figure 5.1a displays the original musical notation. Figure 5.1b displays the transcription of Figure 5.1a. 41
5.2 An illustration of the domains of the language defined by the Original Grammar (blue) in relation to the domain of the language of valid musical scores using the same alphabet as the Original Grammar (red). 41

5.3 An illustration of the domains of the language defined by the reworked grammar (green) in relation to the Original Grammar (blue) and the language of all valid musical scores (red) 47
5.4 A chord which is serialized in three different ways 48
5.5 Example of a simple, single staff polyphonic score 48
5.6 Nested Tuplets 50
5.7 An example score illustrating the tie problem 51
5.8 Illustration of the three classes of ties, with labels to help identify them. . 52
5.9 A comparison between an unprocessed AST for Listing 5.14 and the same AST after pruning. The untouched nodes keep the same line numbers, causing the empty lines in between to clearly show the removed verbosity. 56
5.10 An illustration of the tie matching algorithm. Each box is a note with a tie. The pitches are color coded. In Figure 5.10a each note with a tie start examines the following notes to detect errors. In Figure 5.10b the rest of the notes see if they have previously been checked. Notes that have not been examined, like the green box, are reported as errors 58

6.1 Incipit 201003440-1_1_1 is an example of a malformed score from the PRiMuS dataset. 73

B.1 Incipit 000102052-1_2_2 from package_aa. The 2 bar multirest is misplaced and there should be some bar lines. 93
B.2 Incipit 000108292-1_1_2 from package_aa. The first 1 bar multirest is misplaced and misses barlines. Does not make sense in the big picture, as there is a lone quarter note after it. 94
B.3 Incipit 000130315-1_1_1 from package_aa. Last bar and multirest is clashed together. Transcription states it is a 1111 bar multirest. 94
B.4 Incipit 201003440-1_1_1 from package_ab. The last bar and multirest is clashed together. 94
B.5 Incipit 201003441-1_1_1 from package_ab 94
B.6 Incipit 201003475-1_1_1 from package_ab 94
B.7 Incipit 210097416-1_5_2 from package_ab 94
B.8 Incipit 212001010-1_4_1 from package_ab 94

Listings

1.1 example output from an OMR solution 2
1.2 Possible output from a parser 3
2.1 excerpt from the MNC scanner definition 12
2.2 Excerpt from the parser definition in Bison showing a production rule . . 13
5.1 An excerpt of the Original Grammar containing the most relevant parts in EBNF. 40
5.2 Unfinished grammar from the grammar development phase in BNF. Irrelevant details are left out to clearly show the structure. The grammar features a `symbol` non-terminal with fewer productions and several new non-terminals 42
5.3 Excerpt from the grammar development phase in BNF. Irrelevant details are left out to clearly show the structure. It features a new axiom which has a single production that enforces a specific structure 43
5.4 Excerpt from reworked grammar in BNF showing the optional change of clefs between symbols and key and time signatures between bars. C-style comments denote empty production rules 44
5.5 Excerpt from reworked grammar in BNF showing the `multirest` symbol and how it is used 45
5.6 Excerpt from the reworked grammar showing grammatical changes to the note and rest symbols 46
5.7 Possible transcript of the polyphonic piece in Figure 5.5 using parenthesized encoding of chords as the only addition to the reworked grammar . . 47
5.8 Excerpt from the Reworked Grammar displaying the relevant parts for the newly introduced chord symbol 49
5.9 Excerpt from the reworked grammar displaying the relevant parts for the newly introduced tuplet symbol 50
5.10 An ambiguous transcription of Figure 5.7 using a tie 51
5.11 An example of the new way of defining ties 51
5.12 Excerpt from the reworked grammar in BNF showcasing the tie productions 52
5.13 Definition of a node in C 53

5.14 A syntactically correct string which is still invalid. The bar is underfilled and the ties do not match . . . 55
5.15 DCG for the language of $a^n b^n c^n$ written in SWI-Prolog . . . 67
5.16 DCG for a chord structure written in SWI-Prolog . . . 68

Abbreviations

ALGOL	=	Algorithmic Language
AI	=	Artificial Intelligence
AST	=	Abstact Syntax Tree
C-OMR	=	Classical Optical Music Recognition
CFG	=	Context Free Grammar
CSG	=	Context Sensitive Grammar
DL-OMR	=	Deep Learning based Optical Music Recognition
DCG	=	Definite Clause Grammar
HTR	=	Handwritten Text Recognition
IAL	=	International Algebraic Language
IR	=	Intermediate Representation
ISA	=	Instruction Set Architecture
MNC	=	Musical Notation Compiler
OCR	=	Optical Character Recognition
OMR	=	Optical Music Recognition
PRiMuS	=	Printed Images of Music Staves
VWG	=	Van Wijngaarden Grammar

Chapter 1

Introduction

Optical Music Recognition (OMR) concerns making computers understand the music embedded in documents like handwritten or printed musical scores. OMR has been researched since the late 1960s and is still considered a largely unsolved problem today. OMR can be divided into two distinct categories based on technique: *classical OMR* and *Deep Learning based OMR*.

Whenever classical OMR (C-OMR) is mentioned in this thesis, it refers to OMR solutions that do not make use of neural network models. Instead, technologies like digital image processing techniques such as morphology and frequency analysis are used. By removing noise and taking out non-crucial pieces of the image, like staff lines, researchers were able to extract the other musical symbols for classification [21].

The application of neural network models in the context of OMR is a more recent development. "Deep Learning-based OMR" (DL-OMR) refers to OMR solutions using the highly data driven approach of training artificial neural network models. Thus, researchers have been able to model challenging problems. This development has lead to a surge of successful OMR solutions that can accurately classify musical objects [37, 38]. Some researchers have even produced end-to-end neural network solutions [2, 10, 11]. However, the deep learning approach requires using large data sets of musical scores and *ground truth representations* for training and testing. Ground truth representations are correct representations of the musical scores that are used for comparison. The only way to evaluate whether a DL-OMR solution is correct is by comparing it with the ground truth representations. Therefore, the results of a DL-OMR solution cannot be validated when solving real life problems, as these lack a ground truth representation.

On the other hand, *formal grammars* are able to determine whether or not a particular string is correct without the use of ground truth representations. If a piece of musical notation is represented as a string, then formal grammars could be applied in this context. A formal grammar is a rewriting system that defines how a starting point can be expanded and rewritten into any string of the *formal language* it defines. Formal grammars also work the other way around, by determining if a given string is part of the formal language. This is called *parsing*. Parsing can solve the validation problem of a DL-OMR solution without

using predefined ground truth representations. Previous research within the field of OMR has already used formal grammars. In C-OMR it has been used to assemble complex musical symbols from a set of lines and dots. The field of algorithmic composition, which considers the problem of composing music automatically, grammars have been extensively used to compose music by repeatedly expanding a starting point.

This thesis proposes the use of formal languages to represent musical notation and automatically validate the results of a chosen OMR solution. A parser for a formal language representing music will be applied to the output of the OMR solution. Further, techniques commonly found in compilers will be applied to enforce additional constraints, thus enabling the detection of more errors. In order to fully define the language using only a grammar, a study of formal languages will be conducted. The goal of this study is to find a type of formal language that would allow the grammar to include the required constraints in its syntax. This separates the definition of the language from its implementation, as no constraint checking code will be necessary.

Figure 1.1: Musical notation incipit 000051652-1_2_1 from the PRiMuS data set

To illustrate the usefulness of the proposed approaches an example will now be provided. Figure 1.1 displays a musical notation incipit from a data set provided by Calvo-Zaragoza et al [10], see Section 2.5. The authors also propose an OMR solution that produces an output like the one displayed in Listing 1.1 if the incipit in Figure 1.1 was used as input.

Listing 1.1: example output from an OMR solution

```
1  clef-C1 keySignature-EbM timeSignature-2/4 multirest-23 barline
       rest-quarter rest-eighth note-Bb4_eighth note-Bb4_quarter.
       note-G4_eighth barline note-Eb5_quarter. note-D5_eighth
       barline note-C5_eighth note-C5_eighth rest-quarter barline
```

Currently, the only available method to validate the output is by manual verification or by comparing it to a previously made ground truth representation of the input image. Manual verification of outputs is error prone due to human errors, tedious and time consuming. A detection error has been emulated by deliberately removing a word from Listing 1.1, making it an invalid string, to illustrate how difficult the errors are to find manually. Alternatively, comparing the output against a ground truth is efficient and potentially not as error prone as it removes human interaction. Depending on how each ground truth representation was generated, there could be errors present. An error in a ground truth representation could either result in false positives, where an invalid string is accepted, or false negatives, where a valid string is rejected. Ground truth representations are only available when the musical score has been scanned or examined before, which severely limits practical use.

If the musical incipit depicted in Figure 1.1 is a brand new piece of music, and thus having no previously made ground truth representation, the only way to find out whether or not the output is correct is to do it manually.

This example problem could be solved using a formal grammar. Using formal grammars to define expected patterns and structures is a way to embed domain knowledge into a system. Because of this domain knowledge, the system will be able to reason about the input to detect errors without a ground truth representation. By constructing a parser for a grammar that defines the language of all possible outputs from an OMR solution, it would be possible to detect errors automatically. For example, running the parser on the contents of Listing 1.1 would immediately output an error message stating what it expected and what it got instead, like in Listing 1.2.

Listing 1.2: Possible output from a parser

```
1 Syntax error! Bar 2 is over filled. Did you miss a barline after
    note-Bb4_eighth and before note-Bb4_quarter.?
```

This will allow users to quickly identify errors in their results, similarly to how developers quickly find syntax errors in their code because of useful diagnostic messages.

1.1 Goals

This thesis aims to evaluate the feasibility of a language based validation technique in OMR. In order to do so, the grammar for musical notation defined in Calvo-Zaragoza et al [10] is reworked and extended. This grammar defines a format for encoding musical notation in plain text and is called the "Original Grammar", see section 4.1. Next, a compiler front-end that uses this grammar is implemented. This compiler will do lexical, syntax and semantic analysis of the musical notation, represented in the format defined in Calvo-Zaragoza et al [10]. Additionally, a study of other types of formal languages is conducted. The goal of the study is to find formal languages with features that may further develop the compiler.

The reworked grammar should include new features and improve the existing features, resulting in a language that looks identical to the original. It should only differ in terms of the underlying structure, as well as a few added symbols. This means that it will stay compatible with the data set provided by Calvo-Zaragoza et al [10] which will be used for testing.

A summary of the goals of this project is the following:

- The Original Grammar defined in Calvo-Zaragoza et al [10] is to be extended in the following ways:
  - **Restrict the defined language:**
    The Original Grammar accepts many strings that should not be part of the language. The reworked grammar should not accept such invalid strings.

  - **Increased vocabulary:**
    The reworked grammar should include support for new musical symbols, in addition to all the previously included symbols.

- The lexical, syntax and semantic analysis stages of a compiler is to be implemented. This compiler front-end should be able to validate strings that represent musical notation.

- A study of formal languages is to be conducted in order to see if one can embed context sensitive aspects of musical notation in the grammar itself. This will eliminate the need for a semantic analysis stage in the compiler.

1.2 Report Outline

The report is structured as follows:

- **Chapter 2** presents necessary background.
- **Chapter 3** presents a literature review on the field of OMR as well as algorithmic composition.
- **Chapter 4** presents the experiment- and software designs.
- **Chapter 5** presents the process of achieving the aforementioned goals.
- **Chapter 6** presents the results found after conducting the experiments, as well as a discussion on these.
- **Chapter 7** discusses and concludes the report and also proposes future work that would be interesting.
- **Appendix A** includes all grammars
- **Appendix B** presents the errors found in the PRiMuS data set.

Chapter 2

Background

This chapter describes the necessary background knowledge to fully understand the rest of this thesis.

First, formal grammars and languages are presented with history, definitions and examples in Section 2.1. This is an important part of this thesis as formal grammars are extensively used throughout this thesis. The presentation of formal grammars will then allow a description of compilers in Section 2.2, as compilers implement formal grammars.

Afterwards, musical notation is presented and explained in Section 2.3. Since musical notation is what will be modelled using formal languages, a basic understanding is necessary. Only parts of musical notation that is relevant to this thesis are presented.

Finally, the field of OMR is presented in Section 2.4. Since this thesis is done in the context of improving OMR, it is useful to know about the field of research. OMR is defined, its goals will be detailed and it is distinguished from similar fields of research. The Printed Images of Music Staves (PRiMuS) data set, a data set intended for OMR research, is also presented in this section as it will be used in this thesis.

2.1 Formal Languages

Formal languages are not natural languages, such as English and Norwegian. Both share a notion of alphabets, grammars, syntax and semantics, but are still two different phenomena. While natural languages are spoken by humans and are a product of natural evolution, formal languages are *designed* by humans for specific applications. Over thousands of years, our natural languages have evolved, and are still evolving, from simple noises to the complex languages we speak today [35]. This evolution was partly due to higher intellectual development, as well as a change in societal needs, culture and social interactions. Formal languages, however, do not naturally appear and evolve this way. These languages are tools deliberately created by humans to solve problems and model phenomena. Formal languages are part of the field of mathematics and originate from several subtopics of mathematics like discrete mathematics and computation theory. The formal languages take heritage from the field of linguistics, in particular the study of grammars by Noam

Chomsky [14] as well as the rewriting systems of Axel Thue [45, 54]. Another significant field of science that have contributed to the theory of formal languages is, perhaps surprisingly, biology [30, 31]. Later, other sciences have adopted the use of formal languages, with the most notable one being computer science.

A formal language, L, is a subset of all possible strings created by combining the symbols of an alphabet, Σ. They define what are valid strings and what are not by inclusion. Thus, a valid definition of a formal language can be a list of strings. Operations like concatenation and the Kleene closure are defined on languages [3]. Given a formal language L, then L^* is the Kleene closure over L. The Kleene closure over a language is any zero or more strings from L or in mathematical notation that $L^* = \cup_{i=0}^{\infty} L^i$. A different way to define a formal language, L, is an alphabet, Σ, accompanied by some rules, R, yielding $L = (R, \Sigma)$. The rules, R, come in different forms depending on the application.

One way of supplying these rules, R, is by defining a *formal grammar*. In this thesis, the chosen definition of a formal language is based on the formal grammar. Hence, a formal language is expressed as a function of the corresponding formal grammar, $L(G)$, given a grammar G. A formal grammar is a set of rewriting rules of some start symbol over an alphabet. By rewriting the start symbol into new strings that may be rewritten further, it is possible to represent even infinitely large languages with just a few rules. Using different types of grammars may enable new constructs in the language or restrict it, depending on the expressive capabilities of the type of grammar. Some grammars lack the expressiveness to match opening and closing parenthesis, while others are equivalent to a Turing machine.

All types of formal grammars have four elements in common: *production rules*, *terminals*, *non-terminals* and an *Axiom*. In mathematical notation a grammar G can be defined as $G = (P, N, \Sigma, S)$, where P is a set of production rules, N is a set of non-terminals, Σ is a set of terminals and is disjoint from N and S is the axiom.

A *production rule* defines how a given string can be transformed into a new string. They are often written as a left hand side, which is the original string, a transformation operator and a right hand side which states what the resulting string should look like.

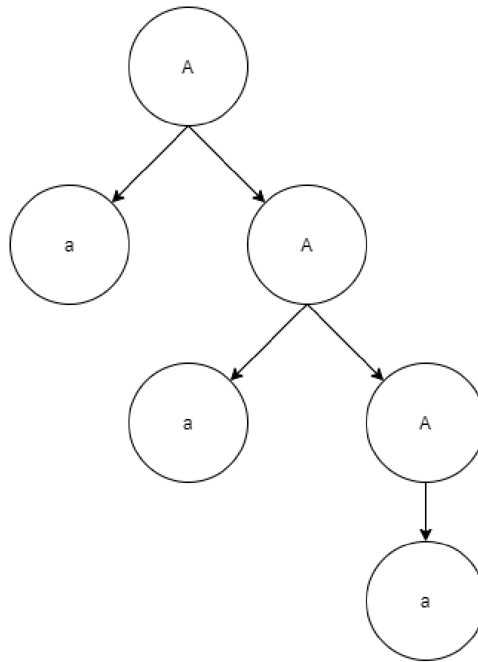
A *terminal* is a string which cannot be rewritten further. A *non-terminal* is a string that must be rewritten. The *axiom* is the starting point of the rewriting process and is a non-terminal. In the example below, a is a terminal and A is a non-terminal as well as the axiom. In order to ensure easy to read grammars, non-terminals are capitalized, while terminals are not.

Applying a production rule to a string is called a *derivation*. Hence, a string can be *derived* from another string. When there are no more possible derivations, i.e. a string with only terminals, then the rewriting is finished. A string that can be rewritten further, i.e. a string that contains non-terminals, is not finished and the rewriting process must continue. In most formal grammars the non-terminals are derived sequentially from left to right, but some exceptions exist.

An example of a simple grammar is shown below, with the production rules enumerated in parenthesis on the left hand side.

$$\begin{array}{lccc} (0) & A & \rightarrow & aA \\ (1) & A & \rightarrow & a \end{array}$$

This means that given an A, there is a production rule that allows the A to be rewritten to



.

Figure 2.1: A tree representation, or abstract syntax tree, for the simple sequence of production rules presented in Section 2.1

aA. This is an infinite language, and can be produced by starting with an A and repeatedly applying the production rules. An example of this is shown below. Note that the production rule that is used is shown in parenthesis on the right hand side. This numbering convention is used throughout this thesis.

$$\begin{array}{ccccc} A & \rightarrow & aA & (0) \\ aA & \rightarrow & aaA & (0) \\ aaA & \rightarrow & aaa & (1) \end{array}$$

These production rules may be represented as an abstract syntax tree (AST). The above example can be illustrated as in Figure 2.1.

2.1.1 The Chomsky Hierarchy of Formal Languages

In 1956, Noam Chomsky [15] defined a hierarchy, later named The Chomsky Hierarchy, which categorizes several types of languages and defines the relations between them. A graphical illustration of the set inclusions that make up the hierarchy is shown in Figure 2.2. Each category is defined by which restrictions are present in the production rules. The hierarchy states that regular languages are the most restricted, thus displayed in the innermost part of the figure. Since the class of context free languages also includes the regular languages, the regular languages are considered to be a special case of the context

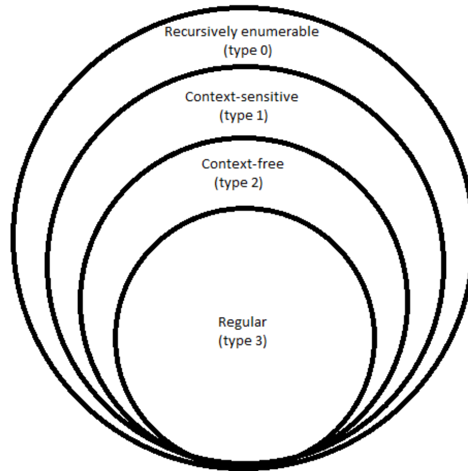


Figure 2.2: The Chomsky hierarchy showing the relationship between the four types of chomskian languages

free languages. Continuing in this manner, context free languages are in turn a special case of context sensitive languages, which again are a special case of recursively enumerable languages. All languages that are in this hierarchy are labelled “Chomskian Languages”.

2.1.2 Other Languages

Non-Chomskian languages also exist, and a few are presented later in this thesis. These languages often tend to be similar to the Chomskian languages in terms of production rules, terminals and non-terminals. However, one or more significant differences make them fall outside of the classification of the hierarchy. For example, the family of languages called *L-systems* might look similar to a Chomskian language, but their non-terminals are rewritten in parallel instead of sequentially from left to right, making the L-systems a different family of languages [30, 31].

Chomskian languages also differ in other aspects from non-Chomskian languages. Chomskian languages all operate on strings, but this is not a limitation seen in other types of languages. Graph grammars, for example, operate on graphs in this manner: given a graph on the left hand side, produce the output graph on the right hand side with the same amount of connecting nodes. A Van Wijngaarden grammar, on the other hand, consists of two grammars [56]. These two grammars fit in the Chomsky hierarchy, but the complete Van Wijngaard Grammar falls outside of it.

2.1.3 Parsing

As stated above, a grammar can expand a start symbol into any string of the language. However, a grammar can also do the opposite. Given a string, it can answer the question “Is this string part of the language?”. This is called *parsing*. Two common parsing techniques

are "top down parsing" and "bottom up parsing". Top down parsing starts with the start symbol, here A, and tries to replicate the input string by repeatedly applying the production rules. If the grammar successfully replicates the input string by rewriting the start symbol, the input string is proven valid. Bottom up parsing, on the other hand, starts with the input string and reduces it by using the production rules backwards. Using the example above, the input string "aaa" will be parsed by reducing the string in the steps shown below. If the input string is successfully reduced to the start symbol, the input string is proven valid.

$$
\begin{array}{ccccc}
aaa & \rightarrow & aaA & (1) \\
aaA & \rightarrow & aA & (0) \\
aA & \rightarrow & A & (0)
\end{array}
$$

The categories of bottom up and top down parsing are broad families of parsing algorithms with different properties such as varying amounts of look-ahead. Each type of grammar usually has one or more parsing algorithms associated with it. This is why the less expressive languages are often used, as it is a trade-off between expressiveness and complexity of parsing. Complex grammars like the Van Wijngaarden grammar and recursively enumerable grammars (type 0 in the Chomsky hierarchy) are seldom used in practice because of the high complexity of constructing a parser.

2.1.4 Notation

The notation of grammars is an important clarification in this thesis. As several different grammars will be examined, a clear notation is needed.

The Backus Naur Form (BNF) is the standard notation for context free grammars. The BNF consists of a left hand side with a non-terminal enclosed in angled brackets, the symbol "::=" which reads "is defined as" and finally a right hand side which defines the production rule of the left hand side. BNF also features an *or* operator, which is represented by the symbol "|". This operator allows a left hand side to be rewritten into either one thing or another. The production rules may become overly verbose in BNF. For example, representing a simple concept like an optional symbol requires the developer to create many production rules.

Because of this, the Extended Backus Naur Form (EBNF) has become popular. EBNF features many useful short hand forms that make it easier to read and write, such as the aforementioned optional symbol and sequences of symbols. A sequence of zero-or-more symbols can be expressed by enclosing the symbol in curly brackets. An optional symbol can be expressed by enclosing it in square brackets. Additionally, EBNF cleans up unnecessary use of angled brackets and other special symbols found in BNF. All EBNF grammars may be converted to an equivalent BNF grammar.

In this thesis, the right arrow symbol, $\rightarrow$, is used as the transformation operator when talking about abstract grammars. An abstract grammar is a grammar that is only theoretical, not a grammar that is implemented by some software. Abstract grammars are allowed to use features found in EBNF, such as repetition. In Section 5.3 only theoretical grammar examples that are not implemented are presented, so the right arrow is used.

If the grammar is implemented in code, then a BNF-like notation is used. Instead of using "::=" as the operator, a colon, :, is used. Angled brackets around non-terminals are

not used. The *or* operator stays the same. A production rule can feature an empty right hand side. This is the case in Section 5.1 as the presented grammar is implemented using Bison which uses these conventions. Bison will be presented in Section 2.2.2.

Exceptions to this rule will be stated when necessary. Notation conventions that are specific for a particular type of language will be explained when necessary. However, when possible the grammar notation in this thesis will comply to these conventions.

2.2 Compilers

Compilers and interpreters are an essential part of computer science today. A compiler is a program that takes source code as input and translates this into another language, most commonly translating a piece of code written in a high-level language into an equivalent or optimized low-level- or even machine language code. This is crucial in order to be able to run code written in a high level language. In the context of this thesis, the front-end of the compiler will be the most interesting part of a compiler. An interpreter is a program essentially performs the same task as the compiler, but with a few differences. Only the compiler is considered in this thesis.

High level languages, like Python and Haskell, would not exist without compilers and interpreters. The languages are defined by them. Because computers, or rather processors, are only able to execute instructions defined by their instruction set architecture (ISA), they do not understand high level languages. Because of this, a translation from high level language to machine-understandable language is needed.

2.2.1 Compiler pipelines

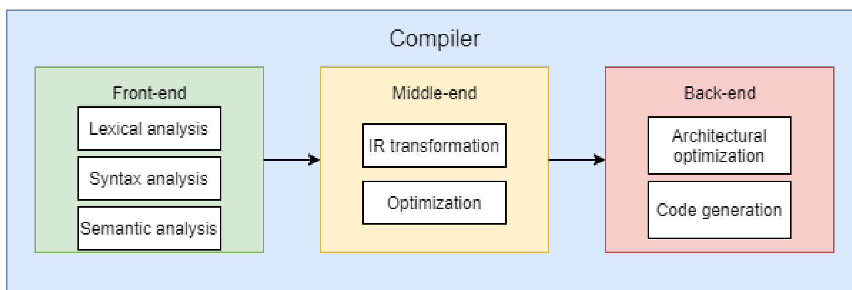


Figure 2.3: An overview of the components of a compiler

A typical compiler pipeline for processing a high level language will now be described, as displayed in Figure 2.3.

In the front-end, illustrated as the green box, the high level language source code is taken as input in order to extract and understand its contents. A more detailed view of a compiler front-end is displayed in Figure 2.4. The first part of the compiler front-end, the scanner, reads the code as a sequence of characters and turns this into a stream of *tokens*. A token is a string with an identified meaning. For example, in a programming language

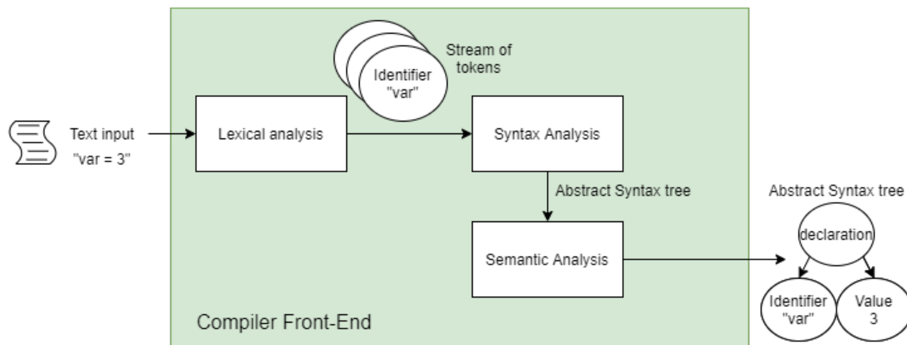


Figure 2.4: A more detailed look at a compiler front-end

such as C, the conditional statement `if` always starts with the string "*if*". After reading the sequence of characters "*if*" the scanner associates the string with a token representing the beginning of a conditional statement. In this example, the string itself is not used after this point, and may or may not be stored. However, in more complicated scenarios, such as identifier names, the token will represent an identifier as well as storing the input string which contains the identifier name.

Next, the tokens are passed to the parser. By analogy, the tokens are only the words of a sentence in a natural language like English. It is not yet known if the sentence follows the grammatical rules of the language or if the sentence makes sense. The parser will use the token stream to verify that the sentence is grammatically correct, but will not check if it makes sense. In order to check for grammatical correctness, the tokens are parsed using a grammar. The grammar defines what different types of sentences should look like. If the parser is unable to match the stream of tokens to the rules of the grammar, a syntax error is reported, i.e. the input token stream is invalid.

During parsing it is common to iteratively build a data structure called an *Intermediate Representation* (IR). This is called a *syntax directed translation* [1]. The IR is a language-independent representation of the code. The IR built during parsing might be just one of many intermediate representations the compiler uses during its process. An AST, as previously depicted in Figure 2.1, is a commonly used IR and is the only type of IR used in this thesis. Many parsing libraries will allow the developer to add functionality to construct an IR or perform other arbitrary actions during parsing. This representation allows for optimization and semantic analysis which is the last step in the front-end.

Following the previous analogy, the semantic analysis phase evaluates whether the sentence makes sense. Semantic analysis often regards syntax rules that could not be represented in the grammar. Since programming languages have some context sensitive aspects it is necessary to verify that these constraints are valid. For example, using an identifier in a function requires the identifier to be defined. However, This does not mean that the compiler checks whether or not the program does what the developer intended.

The compiler middle-end is responsible for transforming the IR into an equivalent and more efficient representation of the same code. Examples of typical optimizations include dead code elimination, constant propagation and loop-invariant code motion [1]. Details

on optimizations will not be presented here, as it is outside of this thesis' scope. The optimized IR is then sent forward to the back-end.

In the back-end, architecture specific optimizations, transformations and code generation is done. The middle-end is completely architecture independent, while the back-end is architecture dependent. The IR created during parsing contains enough data to generate machine- or assembly code. A single compiler may have several front- and back-ends in order to support different languages and architectures.

2.2.2 Compiler development tools

There are tools to help develop compilers and parsers. This subsection presents a selection of these tools. In this thesis, two tools have been used: Bison and Flex. Both these tools help develop the compiler front-end. Other tools were considered but not used in the implementation presented in this thesis.

Flex

Flex is a library for generating lexical analysers, also known as scanners, in the programming language C [20]. The scanner's job is to read a string input and categorize each word. In this context, a word is a syntactic token that has is relevant to the language. For example the word `if` is reserved in many languages and needs to be recognized. Identifiers also often follow strict naming rules, which needs to be defined. Words that are not recognized may be treated as errors that cause the program to either fail or take other actions.

When writing a Flex scanner one defines rules, as regular expressions, for each type of token, or word, defined in your language. Listing 2.1 shows an excerpt of a scanner definition used in this project. Notice the left hand side of each line features both a single string, such as "barline", options separated by a column symbol, regular expressions and finally start conditions enclosed in angled brackets.

Listing 2.1: excerpt from the MNC scanner definition

```
1  keySignature        { return KEY_TOKEN;                     }
2  timeSignature       { BEGIN(METER); return TIME_TOKEN;      }
3  barline             { return BARLINE_TOKEN;                 }
4  chord               { return CHORD_TOKEN;                   }
5  tuplet              { return TUPLET_TOKEN;                  }
6  n|x|bb|b|\#         { return ACCIDENTAL_TOKEN;              }
7  m                   { return MINOR_TOKEN;                   }
8  M                   { return MAJOR_TOKEN;                   }
9  <METER>C|C\/        { BEGIN(INITIAL); return METER_TOKEN;   }
10 [A-G]               { return DIATONIC_TOKEN;                }
11 [0-9]+              { BEGIN(INITIAL); return INTEGER_TOKEN; }
```

This results in generated source code for a scanner that can follows this specification. Whenever it recognizes one of the left hand sides it performs the action specified in the curly brackets on the right hand side. In this excerpt, all rules simply return a token that is used by the parser later. Arbitrary code can be put in these braces.

Some of the rules in the example also activate a start condition, denoted by the BEGIN macro. Start conditions are a powerful tool in order to provide context and remove ambiguity for the scanner. If there is no start condition on the left hand side, it is implicitly interpreted as the `INITIAL` start condition. In this example, lines 9 and 10 in listing 2.1, would be in conflict if there weren't any start conditions. The `DIATONIC_TOKEN` rule on line 10 captures all letters from A to G, while the `METER_TOKEN` rule on line 9 captures the letter C. It is clear that both rules capture the letter C. How would the scanner know which token is which? By recognizing where a meter token can arise, namely after a time signature as seen on line 2, this ambiguity is eliminated. The scanner will only return a meter token if the letter C was seen after a time signature. Otherwise, the letter C would be recognized as a diatonic token.

After the scanner has run, the string input is transformed to a token stream that is more suitable for parsing than a raw string input.

Bison

Bison is, similarly to flex, a code generating library that can be used with the programming language C [22]. Unlike Flex, Bison generates parsers. Specifically, Bison generates a LALR(1) parser which uses a bottom up algorithm with 1 symbol of look-ahead. Recall that parsers are programs that verify the syntax of a string according to a grammar, as described in Section 2.1.3. Bison and Flex fit well together, and may be configured to work together without any orchestrating code. The input token stream is the very same that is output from the scanner generated by Flex. The parsers job is to match the input token stream to patterns, revealing syntax errors in the process.

The patterns that tokens are matched by are defined as the production rules of a context free grammar in Bisons own format. This format is written as source code which is used to generate the parser and a human-friendly grammar notation. This format features the same constructs as BNF, so defining sequences of symbols and optional symbols is equally verbose. The human-readable grammar notation that Bison can output was presented in Section 2.1.4. See Listing 2.2 for an excerpt of the actual bison definition used in this project.

Listing 2.2: Excerpt from the parser definition in Bison showing a production rule

```
1   score:
2      clefcom keysig timesig bars {
3         root = (node_t*) malloc (sizeof(node_t));
4         node_init(root,SCORE,NULL,4,$1,$2,$3,$4);
5      };
6
7   major:
8      MAJOR_TOKEN {
9         node_t *node = (node_t *) malloc (sizeof(node_t));
10        node_init(node,MAJOR,NULL,0);
11        $$ = node;
12     };
```

The topmost production rule presented in Listing 2.2 is equivalent to

$$\text{score} = \text{clefcom keysig timesig bars}$$

where `clefcom`, `keysig`, `timesig` and `bars` are non-terminals defined elsewhere. Note that both production rules also includes a code block in curly brackets. Bison allows actions to be associated with every production rule. In this case, the action is to create a node. These nodes will be tied together to form an abstract syntax tree. This is an example of syntax directed translation.

Note that all of the terminals are tokens, and are represented with capitalized names. Recall that each lexical definition in Flex had an associated `return` statement that returned a token. These tokens are, in fact, not defined in the Flex source code but rather in the Bison source code. The bottom production rule in Listing 2.2 is equivalent to

$$\text{major} \rightarrow \text{MAJOR_TOKEN}$$

which shows the use of a token in a production rule, in turn creating a node. This node is then used in other production rules and is then associated with a parent node, thus creating the AST. Most tokens have a node type associated with it, to create a node for a token. The node types do, however, also include types for higher level nodes that does not directly use any tokens such as the SCORE type shown in the topmost production rule.

Parsec

Writing Flex and Bison specifications in an early development stage may prove time-consuming and labour-intensive. When the grammar is under development, and thus changing, it is hard to keep the specifications up-to-date. To mend this, Parsec could be used in order to rapidly construct parsers for grammars during development. For example, in order to change a symbol or a production in a software pipeline consisting of Flex and Bison one would have to change both the Flex and Bison source code. Depending on the grammar, this change will have consequences that propagate to various parts of the software, such as the actions of a syntax directed translation.

Parsec is a library for writing parsers using *parser combinators* [41]. A parser combinator is a higher order function that combines several input parsers into a single output parser. This approach allows the definition of small parsers, where each parser only accepts a single or a few tokens. These parsers may then be combined to create more complex parsers. This approach includes both lexical analysis and syntax analysis in one definition, unlike Flex and Bison which only does lexical analysis and syntax analysis respectively. Due to the combination of simple parsers, it is easy to change a definition and have the change propagate successfully throughout the parser.

Unlike Flex and Bison, which is commonly used with C/C++, Parsec is a library for the programming language *Haskell* [26]. Because of the difference in programming languages, a tool using Parsec is not developed for this thesis. It is practical to only consider one programming language, to avoid potential compatibility issues. However, an existing tool that is based on Parsec is found and used during grammar development. This tool is presented in Section 4.1.

2.3 Musical Notation

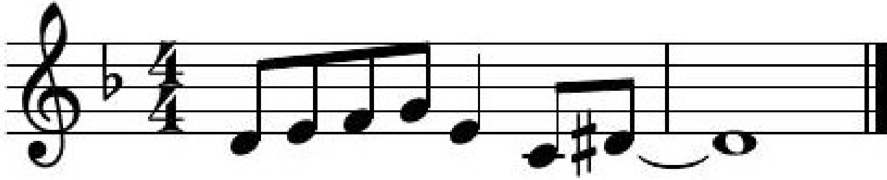


Figure 2.5: An example of musical notation

Musical notation is a system of representing a piece of music in writing. While many historical and current systems exist, this thesis will only consider the western classical notation.

2.3.1 The Symbols of Musical Notation

There are many components to the western classical notation. Figure 2.5 shows a small musical notation example which includes a variety of symbols which will be presented. This section will focus on the symbols that will appear later in this thesis, instead of covering the entire set of musical symbols.

First of all, there is the *staff*. The staff are the five horizontal lines that are used as the canvas on which music is written. It serves the purpose of creating a reference point, as other symbols can mean different things based on vertical position. With the staff, one does not have to measure the position manually, but instead one will quickly see on which line or space in between lines the symbol in question lies.

The staff is divided into several *bars*, or measures, by *bar lines*. Bar lines are the vertical lines that go across the staff, and they separate the bars from each other. The musical notation shown in Figure 2.5 has two bars. The bar is has a certain duration, denoted as a number of *beats*, which is indicated by a *time signature*. There must be at least one bar in a staff and all bars must be properly filled by other symbols that will be presented later.

The time signature is often a fraction, where the nominator is the number of beats and the denominator is the value of a beat. The denominator is also always a power of two, as is the case with most rhythmical definitions in music. The example in Figure 2.5 has a time signature of $\frac{4}{4}$. There are two noteworthy exceptions to the time signature, which are artefacts from older musical notation that has become a part of the modern musical notation. First there is the *Common Time* signature, which is represented as a C instead of a fraction. The Common Time signature is equivalent to a time signature of $\frac{4}{4}$. Finally, there is the *Cut Common Time* signature, which is denoted by a slashed C and is equivalent to a time signature of $\frac{2}{2}$. These two time signatures are mathematically equivalent, but the musical meaning differs. The fractions of a time signature, $\frac{x}{y}$, means the bar allows x beats of the value $\frac{1}{y}$. For example, a time signature of $\frac{3}{4}$ allows 3 quarter beats $(3 * \frac{1}{4})$.

Figure 2.6: Two notes with different graphical properties determining both pitch and duration. The note head is indicated in red and the stem is indicated in blue.

The *anacrusis* is a special case of a bar. The anacrusis is a bar that is not completely filled. It is, if present, always at the very start of the score and represents a pick-up or phrase that leads into the first bar. This means that a valid score can include one bar, the anacrusis, which is not properly filled, as long as it is at the very beginning.

The *note* is the basic building block for expressing the aural information in the musical score. informally, this means that the note represents what you hear. As seen in Figure 2.6 notes are the circles that are placed on or between staff lines. A note is comprised of two parts: The note head, which is indicated by the red squares, and the note stem, which is indicated by the blue square. The note stem may, in turn, be extended with a tail which is the line on top of the note stem from the note head. An example of several notes with different tails can be seen on the top line of Figure 2.7. The length of a note is determined by a combination of the head, stem and tail. The whole note, which lasts for 4 beats and is the longest commonly used duration, has no stem and does not have a filled in head. The vertical placement of the note determines its *pitch*, which is how "high" or "low" the note should sound. In western music, the pitches are enumerated by the letters A to G, called the *diatonic pitches*.

The pitch of a note is not possible to determine by only observing the note, but requires contextual information from another symbol called the *clef*. The Clef is one of three symbols, the C, F or G clef, that gives a reference point to which pitch is located where on the staff. The example in Figure 2.5 displays a G clef, which is the leftmost symbol.

It is also possible to modify the pitch of with other symbols called *sharps*, *flats* and *natural*. The sharps, flats and naturals are placed on the vertical position they modify. Sharps raise the pitch of all notes on the same vertical position by a semitone, while flats lower them by a semitone. Double sharps and flats also exist, which raise or lower the pitch by a whole tone. The naturals override the changes made by sharps and flats, causing the pitch to be reset. When these symbols appear sporadically, right before the note they modify, they are labelled *accidentals*.

They can also be placed at the start of a line, creating a *Key signature*. Key signatures define an implicit accidental on a set of pitches. For example, a key signature may specify that all pitches of F has an implicit sharp accidental, effectively raising the pitch by a semitone. The natural accidental may also temporarily override the key signature. In Figure 2.5, the key signature consists of a single flat, so any notes with that particular diatonic pitch will have an implicit flat accidental.

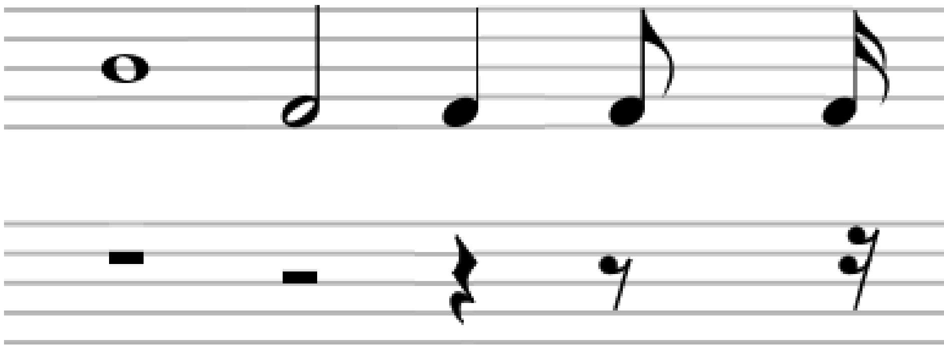


Figure 2.7: Notes and rests of different common durations. Notes are on the top line and rests are on the bottom line. Each vertical pair has the same duration.

The beginning of a score must specify a clef, time signature and key signature. This is called the *heading*. Without the clef or time signature, it is impossible to determine pitches and place bar lines. The key signature may consist of either sharps or flats in various numbers. The case of no sharps or flats is still a valid key signature, despite the lack of information. It is possible to change the clef, time signature and key signature, but only at certain places. The clef may change at any time, but the time and key signatures may only change at the start of a bar.

Several notes stacked on top of each other is called a *chord*, and simply represents that these notes should be played at the same time. Each note in the chord may have its pitch modified like a normal note. The chord has a uniform length, which means that all notes are held for the same base duration. The only exception to this is that individual notes may be *tied*, which will be explained later.

In direct contrast to the note, there is the rest. While notes express what is being actively played, the rests simply represent silence. Since silence does not have a pitch, its vertical placement with respect to the staff lines is less important. If the music is transcribed as two or more voices inside the same staff, then the vertical placement may indicate which voice is supposed to be at rest. The rests do, however, have duration and follow a similar logic to that of the note. The bottom line of Figure 2.7 display rests of different commonly used durations.

The duration of both the note and the rest may be modified in even more advanced manners. A collection of symbols, which in this thesis are called *modifiers*, can alter the duration of them in various ways.

First, there is the *tie*. The tie is a curved line that connects two or more rests or notes of the same pitch together. When two or more notes or rests are connected, then only the first is played but for the duration of all the tied notes or rests. This means that two quarter notes can be tied together to effectively form a half note. Note that ties can cross bar lines, making it the only way to represent a note or rest that should be held across bars.

Then there is the *dot*. A dot is simply a dot that is placed after the symbol it modifies. It multiplies the duration of a note or rest by half its value. This means that a dotted quarter note is equivalent to a quarter note and an eighth note tied together. Several dots may be

placed on the same symbol, which multiplies $\sum_{i=1}^{n} \frac{1}{2^i}$ to the base duration for n dots.

The last modifier is the *fermata*. The fermata is a symbol placed above the symbol it modifies and means that it should be held longer than its value may suggest. The exact time it should be held is not defined, and is up for interpretation by the musician, conductor etc.

Finally, the *tuplet* is a rhythmic construct that allows arbitrary rhythms to be represented. By grouping N notes together and stating that their total duration is M beats, an arbitrary note length is possible to achieve. The most common one is the triplet, a tuplet with $N = 3$, which creates three notes that last for a total of one beat. A tuplet is denoted by a series of notes that are bracketed together. The bracket states what N is, and sometimes what M is. When M is omitted, it is up to the context to determine the M. This is usually done by examining the length of the notes in the tuplet.

2.3.2 Taxonomy of Musical Scores

Often, one voice is written in each staff. According to the taxonomy of Calvo-Zaragoza, Hajič and Pacha [12] this is called a monophonic score. A monophonic score is often used to transcribe the part played by an instrument that is only able to play one note at a time, like a flute. Other uses for monophonic scores arise in jazz lead sheets and other transcriptions of melodies without transcribed accompaniment.

Expanding the monophonic score with the ability to play chords yields the homophonic score category. This still represents a single musician, but the musician is able to play several notes at the same time. The homophonic scores are still limited to one voice per score. Rhythmically, they are as simple as the monophonic scores.

Adding more voices to each staff yields the polyphonic category. This means that the score can represent what several musicians play in one score. While they may seem similar to the homophonic scores, the polyphonic scores are rhythmically more advanced. The polyphonic scores will represent two voices playing simultaneously similarly as chords, but allowing different durations.

Finally, the piano form category is for instruments that may play several voices at once, such as a piano. These are similar to polyphonic scores, but the piano form scores include several staffs that exhibit some interaction. Examples of such an interaction is phrasing that moves across staffs.

2.4 Optical Musical Recognition

Optical Musical Recognition, OMR, has been researched for over 50 years but still lacks an acknowledged definition. Research papers tend to not follow the same definition of OMR and seek out different goals. The technical aspect of a solution has been the most important one, and therefore no definition of OMR as a whole has been made and acknowledged. In this thesis, the definition by Calvo-Zaragoza et al [12] is to be used as their paper clearly states these problems and proposes solutions to them.

2.4.1 Defining OMR and OMR Solutions

Using Definition 1 from Calvo-Zaragoza et al [12] and its explanation, OMR obtains the following definition:

Definition. *Optical Music Recognition is a field of research that investigates how to computationally read music notation in documents.*

This clearly states that OMR is a field of research. Many authors have regarded OMR as a process, which simply turns OMR into a mapping from input to output. It is, however, possible to formulate several tasks with the same inputs and outputs, which makes this mapping definition insufficient. There are many sub-problems to consider for the same pair of input and output.

Next, the definition clearly distances OMR from the fields of musicology and other humanities by stating that OMR investigates how to *computationally* read music. OMR is not a study of how musical notation systems work, how they are decoded and how humans read music. Instead, OMR uses this knowledge in order to enable a computer to read musical notation.

Finally, the last part attempts to concisely and inclusively define OMR. It does not state the format of the output, nor does it state what the final goal of an OMR solution or piece of research should be. By only capturing the essence, which is reading musical notations from documents, the definition includes the many different approaches and goals within OMR. For example, both musical notation reconstruction solutions and musical playback solutions are included.

OMR is often compared to optical character recognition (OCR) and handwritten text recognition (HTR). While they do have much in common, there are some aspects of OMR that makes it more than "OCR for music". Both OMR, OCR and HTR all share a set of symbols that make up the alphabet of supported symbols. Then, symbols from the alphabet is identified in the document that is being read. OMR solutions could feature an alphabet of musical symbols, as well as letters, while OCR and HTR only feature letters. This is an important difference. The letter "A" is always the letter "A", no matter the configuration of positioning, relationship to other symbols etc. A note, on the other hand, bears a new meaning with different positions so one must keep track of both its position as well as the context it is used.

Another key difference is the extent of the analysis. OCR and HTR tends to read the symbols and thus produce a text. OMR also has to read the symbols but a great deal of effort has to be put in recovering the semantics, what the music *means*. Because the symbols may have new meanings due to features such as position and context, there is more to OMR than simply reading the graphical symbols.

As stated in the introduction, this thesis divides the field of OMR into deep learning based OMR (DL-OMR) and what is labelled classical OMR (C-OMR). In this thesis, C-OMR is simply OMR using any technique that is not deep learning. This means that the OMR system has domain knowledge embedded in it, so it does not need to learn. This domain knowledge can then be used on a result obtained from general techniques, such as image processing, to identify different cases of a problem. Depending on which sub-problem a C-OMR solution attempts to solve, there are a plethora of possible techniques to be used. Examples include image processing techniques such as morphology, binarization,

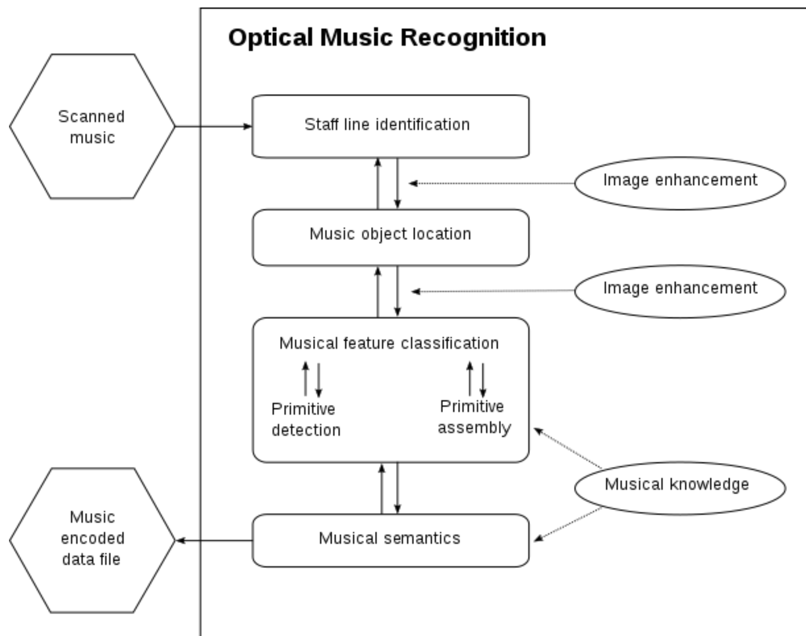


Figure 2.8: An illustration of the first OMR pipeline by Bainbridge et al [6]

histogram equalisation [23] and projections [21] as well as language based segmentation and assembly of graphical primitives [16].

Deep learning based OMR, however, denotes the approaches that utilize artificial neural networks in some or all parts of the solution. The highly data driven approach of training neural networks has yielded very promising results [10, 37, 38]. Its main drawback is that the neural networks are not understandable by humans and is therefore hard to reason with. Deep learning approaches has been successfully used in parts of the OMR pipeline, which will be described next, specifically staff processing [13], music object detection [37, 38] and musical notation reconstruction [39]. There are also solutions that entirely skip the pipeline and present complete end-to-end solutions [10, 11].

2.4.2 The OMR Pipeline

An OMR solution is typically divided into several stages, forming a pipeline. Bainbridge et al [6] defined the pipeline to be a 4 stage process, where scanned music goes through staff line identification, musical object location, musical feature classification and musical semantics extraction to output an encoded data file. This pipeline is seen in Figure 2.8.

Rebelo et al [50] extended upon this pipeline, which is now considered the standard for OMR and is still extensively used. The new pipeline still consists of 4 stages, namely preprocessing, music symbol recognition, musical notation recognition and final representation construction. Each of these stages has several defined sub-stages that may be part of it. The new pipeline can be seen in Figure 2.9. Do note that the pipelines are

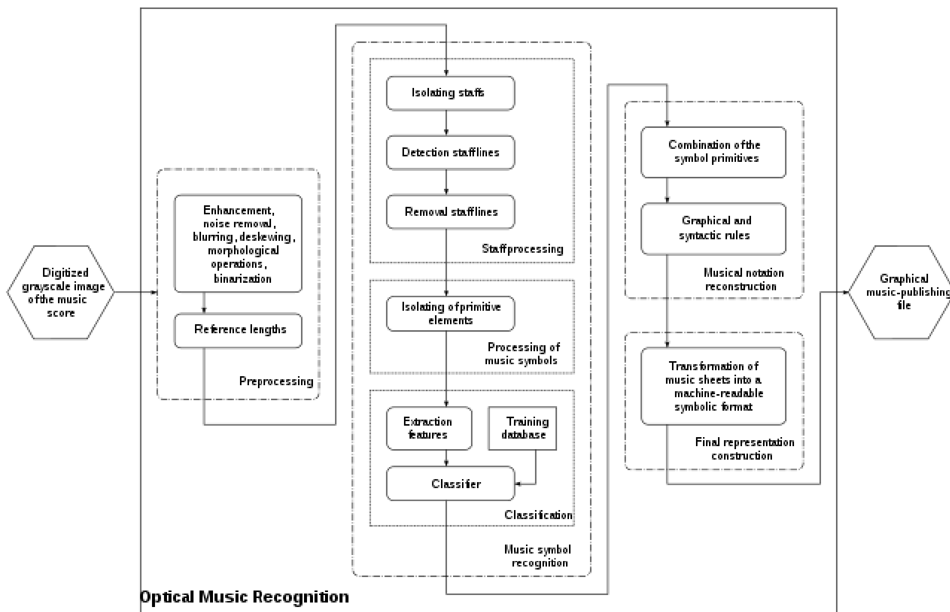


Figure 2.9: The updated and improved OMR pileline as defined by Rebelo et al [50]

mainly focused on reconstructing the graphical musical notation, not only understanding the contents.

The preprocessing stage consists of preparing the document for analysis. Many image processing techniques such as the morphological operations *open* and *close* requires, or prefers, inputs with certain properties. Therefore it is common to convert the image to a binary image to enable the use of more techniques.

Next, there is the music symbol recognition phase. By using the preprocessed image, this phase attempts to identify and classify all musical symbols. Typically, this has included removing the staff lines to isolate musical primitives for classification. Musical primitives are the basic graphical building blocks that are used to create musical symbols. For example, a note may consists of the three primitives note head, note stem and note tail.

With all the musical primitives identified, one may combine these into the complete musical symbols such as notes and key signatures. This is the musical notation reconstruction stage. During this stage, rules concerning the musical semantics are also applied.

Finally, the final representation construction phase simply transforms the data into a useful format. This could be a PDF-document, or a file format that is compatible with a musical notation editing software.

In Calvo-Zaragoza et al [12], it is also argued that the final representation can be something that contains a lower level of understanding than musical notation reconstruction. To precisely reconstruct the musical notation, complete understanding is required. This includes things like whether or not a particular note stem is pointing up or down. However,

an equally valid goal of an OMR solution is to play the music back. This does not necessarily require the musical notation reconstruction stage in the pipeline as whether or not a note stem points up or down is irrelevant. However, a similar phase to musical notation reconstruction which applies semantic rules to the data is still required. Due to this, it is worth noting that the pipeline of Rebelo et al [50] is the standard in the field of OMR, although it is often used with varying terminology.

2.5 The PRiMuS Dataset

The Printed Images of Music Staves (PRiMuS) Dataset is a dataset created by Calvo-Zaragoza et al [10]. It contains 87678 real musical incipits, taken from the RISM project[1]. A musical incipit means that it is not a complete score, but rather just an excerpt. Usually, it is an excerpt from the beginning of the score. The incipits always feature a single staff on a single line, as well as only homophonic scores using the limited alphabet defined in Calvo-Zaragoza et al [10].

Each incipit is represented in five ways. There is a generated PNG image of the musical notation, with a midi file for playback. Next, there are four textual representations. There are two existing formats, namely the Plaine and Easie (PAE) encoding and the Music Encoding Initiative (MEI) encoding [51]. Additionally, there are two new formats defined by Calvo-Zaragoza et al [10] which are called the *Agnostic* and *Semantic* encoding. These are encodings that allow the representation of a single staff as one dimensional sequences, which is practical for neural network models. Every symbol is self-contained, so they do not depend on contextual information to interpret them.

The Agnostic Encoding is a low level representation while the Semantic Encoding is on a higher level. The Agnostic encoding contains the graphical symbols, such as notes, and their location. This means that the musical semantics are not clear in this representation, such as pitch. The Semantic Encoding has included this information but omitted low level information. A note is represented as its pitch and duration, without any mention of where it is placed graphically.

Every incipit is generated by importing the PAE encoding from the RISM project into the musical engraving software Verovio. Verovio can generate images of the musical score as well as convert it to MEI. Furthermore, the MEI encoding was converted to semantic and agnostic encoding. Finally, the semantic encoding is converted to MIDI. This method of generation follows a chain of dependence from the PAE encoding.

[1] `https://opac.rism.info/`

Chapter 3

Result Validation in OMR

A literature review is conducted in order to identify previously applied validation techniques. This chapter presents a condensed overview of the results of this review, including only papers and techniques that were deemed relevant to this thesis.

The review is divided into two parts. The first part, found in Section 3.1, only considers papers related to the field of OMR and surveys which validation techniques, if any, are used.

The second part of the review, found in Section 3.2, considers papers from the field of algorithmic composition. Algorithmic composition is studied because of its many similarities to OMR, as both fields attempt to model music and musical notation.

Finally the chapter is concluded in Section 3.3. The conclusion summarizes the findings in order to identify room for improvement.

3.1 Literature Review on Validation Techniques in OMR

It is hypothesized that the new DL-OMR solutions tend to only use ground truth representations for validation. This is common practice within the deep learning community. A literature search is conducted in order to survey validation techniques in the state of the art DL-OMR solutions, as well as older publications using classical computer vision and OMR techniques. The goal of the literature review is to discover what has been done in OMR validation and what techniques that have been used.

As deep learning is a data-driven technique, many aspects of its development is naturally based on data. This includes validation. During the development of a deep learning solution, an artificial neural network is typically trained by mass amounts of data from a data set containing a data point and a ground truth representation. This data set is divided into several partitions, the training set, the validation set and the test set. Exactly what these partitions are meant for and how training a neural network is done is not relevant for this thesis, however it is worth noting that the only validation of correctness is done by comparison with the test set. This often makes very much sense to do, as the inner workings of the systems the neural nets attempts to model are highly complex and

unknown. In the context of OMR, however, the system of musical notation is very well defined and known – potentially allowing results to be validated by a different approach than the ground truth representation approach.

Table 3.1 shows the results of the literature review. The table uses the established abbreviations DL-OMR and C-OMR to categorize the respective OMR solutions. For larger OMR solutions that have several articles published about it, only one of the articles will be presented in the table. An example of this is Bainbridges OMR system *Cantor*.

It is clear from the literature that the deep learning based approaches all favor the common deep learning approach by comparing obtained results to a ground truth representation. In fact, no other validation methods were mentioned at all. However, Formal languages and grammars were used in a few deep learning based papers. In End-to-End Neural Optical Music Recognition of Monophonic Scores [10], two new formats of musical encoding is devised for use with the proposed OMR solution. These formats, namely the agnostic and semantic encoding format, are defined by a context free grammar. This grammar is otherwise not mentioned, so one may not assume it is used for validation purposes but only for generative purposes. The same paper constructed the PRiMuS dataset, which contains ground truth representations for every musical incipit in both semantic and agnostic encoding. Most likely, the grammar is made to give a formal definition of the formats, but unfortunately the language defined by the grammar is not the intended language for the encoding formats. The language produced by the grammar includes strings that represent invalid pieces of music, so it is not suitable for generation nor validation as is.

In Alfaro-Contreras et al [2], the agnostic encoding is extended to handle homophonic scores. It is not extended using a formal grammar, but rather by explanation of each proposed extension with illustrations. It does make sense to not use grammars in this paper, as they are extending a low-level graphical representation with serialization of multidimensional data. Chords and other constructs that include multiple vertically stacked symbols needs a way to be serialized in order to be represented as a string which are one dimensional.

C-OMR employs several strategies, depending on several factors including the scope of the article. As the OMR pipeline consists of many stages, many papers focus on improving a single stage or even a single sub problem of a single stage. Thus, not every paper mentions any form of validation process that is relevant for this thesis. Because of this, only a relevant subset of the papers reviewed are presented in Table 3.1.

The ground truth representation approach of validation is also rather established in C-OMR. When evaluating the final endpoint or even the semantic stage of an OMR solution, the ground truth representation approach is dominant. During the testing and evaluating phase of developing a solution this makes perfect sense, due to the controlled environment of a test. It is feasible to construct testing materials with ground truth representations for many, if not all, plausible cases. The ground truth representations will also test correctness on many levels, depending on the experiments. For musical notation reconstruction solutions the ground truth representation will include both the semantics of the music, what the music means, as well as the graphical aspects linked to the task, such as whether or not the note stem points up or down.

On the other hand, the ground truth representation approach is not possible outside the testing environments. If the test has not previously been constructed with a ground

truth representation, a test item and verification of these, the approach simply fails. If a system's results is validated only by testing with ground truth representations, it has no way of solving real life problems.

A common technique used by the C-OMR solutions in the low-level graphical segmentation and classification stages are formal languages and grammars. Most of the papers deconstruct the graphical image of a musical score into so-called musical primitives. This has proven to be the most reliant method of symbol segmentation and consequently symbol recognition. These musical primitives include a single note stem, an accidental or just the note head, the circle, of the note. These primitives are then classified. After classification, the primitives need to be combined in order to make meaningful and complete musical symbols. This combination of primitives is in many works done by parsing. Several different formal languages have seen use in this stage, for example context-free grammars [21], extended definite clause grammars [7] and graph grammars [5].

3.2 Automatic Composition of Musical Scores

The field of algorithmic musical composition is a closely related topic in the context of both syntax and semantic analysis of musical notation. While OMR solutions read musical notation and attempt to understand it, algorithmic composition attempts to automatically create valid pieces of music. Semantic analysis in OMR can be used to detect errors by asking questions like "Assuming a sane input, does it make sense that this measure is over filled?". Algorithmic composition, on the other hand, may evaluate a potential next step in the process by asking similar questions like "If I add this note, will it overfill the measure?". These example questions are of course trivial, but serves to illustrate the point that both OMR and algorithmic composition in essence asks many of the same questions. Due to this, an additional literature study of algorithmic composition is conducted. The literature study will have a focus on formal language based approaches because of the findings in Section 3.1, and can potentially be an inspiration on how to further develop the language based approaches for validation in OMR. An overview of the findings is found in Table 3.2, which includes grammars used and additional notes.

The search yielded a clear trend on first glance, *L-systems*. An L-system is a type of formal grammar, where the non-terminals are expanded in parallel. Often, a mechanism of translating the derived strings to graphics is also supplied. L-systems have seen widespread and diverse use in the field of algorithmic composition. Since the broad category of L-systems include a vast amount of different languages with different properties, there are many opportunities to be explored. Two of which are included in Table 3.2.

The first paper, Score Generation With L-Systems by P. Prusinkiewicz [47], uses the basic form of L-systems, the 0L-system. The grammar produces a string that contains instructions for a turtle graphics processor. After processing the string to create an image, the image itself is interpreted at music. The interpretation proposed by Prusinkiewicz is as follows: A cursor is tracing the line produced by the turtle graphics processor. As the cursor moves, sound is played. The vertical position of the line controls pitch, and its horizontal length controls duration. Movement along the horizontal direction does nothing. The pitches may be mapped onto a certain key signature to restrict the possible pitches down to a sensible diatonic selection. This is illustrated in Figure 3.1, for the key

Title	Authors	Year	Category	Validation
A Baseline for General Music Object Detection with Deep Learning [38]	Alexander Pacha, Jan Hajic, Jr., Jorge Calvo-Zaragoza	2018	DL-OMR	Ground truth representation
Handwritten Music Object Detection: Open Issues and Baseline Results [37]	Alexander Pacha, Kwon-Young Choi, Bertrand Couasnon,Yann Ricquebourg, Richard Zannibi, Horst	2018	DL-OMR	Ground truth representation
Towards Self-Learning Optical Music Recognition [36]	Alexander Pacha, Horst, Eidenberger	2017	DL-OMR	Ground truth representation
End-to-End Neural Optical Music Recognition of Monophonic Scores [10]	Jorge Calvo-Zaragoza, David Rizo	2018	DL-OMR	Ground truth representation
Approaching end-to-end optical music recognition for homophonic scores [2]	María Alfaro-Contreras, Jorge Calvo-Zaragoza, José M. Iñesta	2019	DL-OMR	Ground truth representation
A music notation construction engine for optical music recognitio [7]	David Bainbridge, Tim Bell	2003	Score Reconstruction	Low level representation grammar (BCG, extended Definite clause grammar)
Extensible optical music recognition [5]	David Bainbridge	1997	C-OMR	Various steps
Using a grammar for a reliable full score recognition system [16]	Bertrand Coüasnon	1995	C-OMR	Low level representation grammar
Optical Music recognition using projections [21]	Ichiro Fujinaga	1988	C-OMR	Low level representation grammar

Table 3.1: Condensed literature review. Each paper is presented with its employed techniques as well as validation schemes.

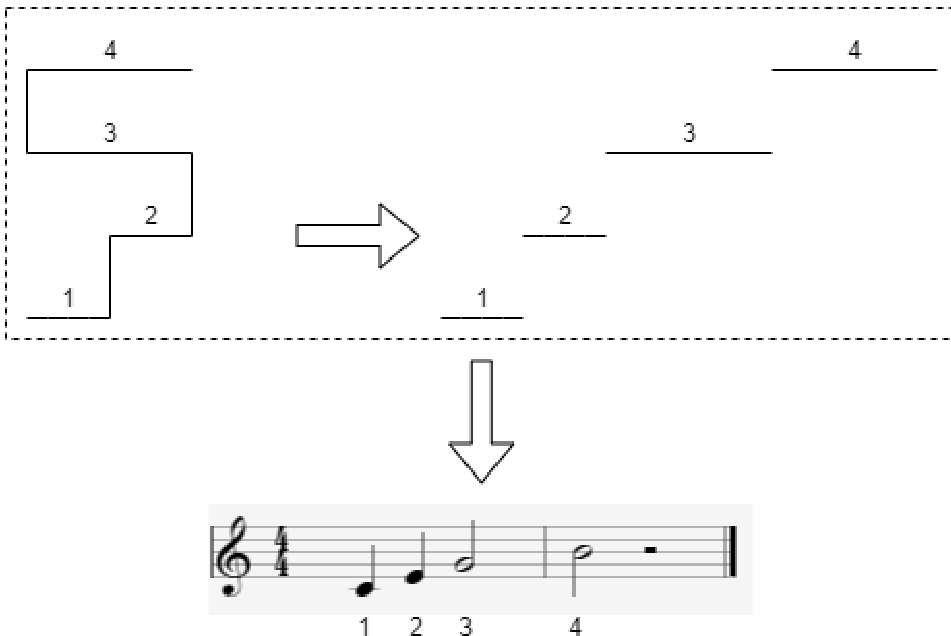


Figure 3.1: A possible interpretation of a turtle curve generated by an L-system. Notes are numbered to clearly show the interpretation

of C major. Albeit not implemented in the paper, a suggestion on how to do polyphony is also made as it is easy to create a branching line with turtle graphics and L-systems. This is the most creative and visually pleasing solution, illustrating how diverse the representation of music may be in computer science.

Next, in Grammar Based Music Composition by Jon McCormack, a different type of L-system is used in order to achieve a great number of things. As the grammar is hierarchical, the level of the compositional stage influences what productions are available. It is also stochastic, enabling several productions for the same left hand side. This made it possible to compactly represent a Markov chain inside the language. Other extensions included parametric productions, making the production rules similar to mathematical functions.

3.3 Conclusions

After the execution of this literature review, an opportunity for improvement was located. Modern DL-OMR solutions only tend to use ground truth representations for validation, while the C-OMR solutions tend to supplement this with parsing. The C-OMR solutions use of parsing is mostly at a low abstraction level, dealing with graphical primitives, and is because of this prone to errors due to the error propagating nature of the approach. If a single primitive is misinterpreted, it can very well cause a domino effect which propagates the initial error onto all successive symbols. This is the main source of criticism of C-OMR

Title	Authors	Year	Grammar	Notes
Grammar-Based Automated Music Composition in Haskell [49]	Donya Quick, Paul Hudak	2013	Stochastic temporal graph grammar (STGG)	STGGs are an extended version of the graph grammar introduced in the paper. They feature both stochastic properties as well as a notion of time. Also parallell rewriting, like L-systems
Score Generation With L-Systems [47]	Przemyslaw Prusinkiewicz	1986	0L-system	Space-filling turtle graphics that are interpreted musically
Grammar Based Music Composition [33]	Jon McCormack	1996	SE0L-system	Stochastic, Extended 0L-system that produce encoded musical strings
Musical L-systems [32]	Stelios Manousakis	2006	various L-systems	3d Space-filling turtle graphics that are interpreted musically
Procedural Music Generation with Grammars	Lukas Eibensteiner	2018	context free grammar, context sensitive aspects	Some rules may wait for adjacent symbols to be derived. Deadlock prevention algorithms were necessary.
Bol Processor Grammars [8]	Bernard Bel, James Kippen	2008	recursively enumerable grammars (type 0)	Developed a notion of pattern language which is contained within the type 0 languages, but cannot be compared to the other categories of the chomsky hierarchy
AI Methods for Algorithmic Composition: A Survey, a Critical View and Future Prospects [40]	George Papadopoulos, Geraint Wiggins	2000	Various	Survey of both grammar based systems as well as other approaches

Table 3.2: Overview of significant literature regarding algorithmic composition with a focus on grammar based approaches

and its use of formal language.

Furthermore, the use of grammars for the lower level symbol construction is an intermediate stage that might not be an explicit stage in the DL-OMR solutions. Depending on the system, a modern DL-OMR solution might not give the developer direct control over symbol construction and classification. Rather, the developer trains a neural network with fitting data to make the network learn classification by itself. This has eliminated the need for this intermediate stage in modern OMR solutions, and might be the reason that fewer solutions are implementing language based validation.

The extensive use of generative grammars in algorithmic composition follows the trend of using language to represent music, with its own flavor. Stochastic extensions of languages are a common sight as it enables some non-determinism that will lead to potentially more interesting compositions. This is unfortunately not an interesting extension in the context of validation in OMR, as determinism is important. The key takeaway from the papers are the feasibility of creating extensions of formal languages to fit the needs of the application. This is unlike the commonly used grammars in OMR solutions, that tend to favor unextended languages.

The validation of the final result produced by the OMR solution, however, still sees a lack of new techniques. Using the ground truth approach is not possible to do outside of a testing environment and should therefore not be the only source of validation. It is practical for a system to be able to tell the user whether or not the process was a success. Of course, it might not be possible for the system to detect all errors by itself, but it should be able to catch many. This validation stage of the high level musical notation may be done efficiently and modularly by the use of a, possibly extended, formal language as inspired from the field of algorithmic composition and C-OMR.

Chapter 4

Experiment and Software Design

This chapter details the methodology of this thesis. First, the experiment design is presented in Section 4.1, i.e. how the different stages of development and testing will be conducted. Terminology will also be clarified. Next, the software design is presented in Section 4.2. The software being developed in this thesis is presented on a high level and considers what data is being transferred where as well as what errors might occur.

4.1 Experiment Design

In order to evaluate the feasibility of a validation solution utilizing formal grammars, a prototype called the Musical Notation Compiler (MNC) will be implemented as a compiler front-end and tested. It is not necessary to create a complete compiler yet. If the analogy of code generation in the compiler back-end is score generation or music playback, then the back-end is application specific. As this thesis only considers validation, only the front-end is relevant.

The MNC will implement a grammar for a language representing musical notation on a high level in order to detect errors in the input. The MNC will not be part of the OMR solution itself, but rather a stand-alone error detection stage. A flowchart illustrating the development process is displayed in Figure 4.1, and the intermediate goals of the process is illustrated in Figure 4.2. The main focus of this thesis will be to implement and test the MNC. In addition, research on formal language theory and how it could be used to extend the MNC is a secondary focus.

Whenever *The Reworked Grammar* is mentioned, it is referring to the grammar that is produced and implemented during this thesis. The grammar presented in the paper by Calvo-Zaragoza et al [10] defined a format originally named the Semantic Encoding. However, as the word "Semantic" is already liberally used in the field of language theory and compilers this might be confusing. This grammar will instead be referred to as *The Original Grammar*. An *invalid string* will refer to a string that should not be accepted, as it does not belong in the intended language.

For the MNC prototype, The Original Grammar presented in Calzo-Zaragoza et al [10] and in Section 2.5 is used as a starting point and reworked. It was deemed practical to take advantage of a previously created grammar. Further, since the Original Grammar defines a superset of the output domain of an already functioning OMR solution, it shows the goal of this thesis is realistic. However, the Original Grammar is not suitable for error detection as it accepts invalid strings. Note that the original definition of the grammar is in EBNF but BNF will be used in this thesis. The difference between the two is presented in Section 2.1.4.

The PrIMuS dataset presented in Section 2.5 features a ground truth representation of each incipit in the language defined by the Original Grammar. This means that each image comes with a correct transcription of itself. Because of this the prototype can be tested using this data set. Simply observing if the Reworked Grammar accepts the ground truth representation will reveal potential flaws in the grammar.

During the grammar development the goal is to restrict it, as well as expand it with new symbols. This is illustrated in the red box of Figure 4.1. The language defined by the grammar should be the intended language, containing only strings that represent valid pieces of musical notation. Originally, the alphabet is rather limited. Hence, the alphabet will also be expanded to show that the new reworked grammar is both robust and easily extended. Since the PrIMuS dataset only contains incipits containing symbols from the original alphabet, new examples need to be constructed for testing and illustration. After this phase is completed, the reworked grammar will be completed, as shown as the output of the red box in fig. 4.2. This phase will be completed using a tool fit for rapid testing and development of grammars. It is not productive to do the grammar development on a full fledged compiler. Instead, this tool parses a file containing an input grammar, and quickly constructs a parser for the grammar. The constructed parser can then be used to test the input grammar, without the need of writing any code. The tool is not written by the author of this thesis, but found online without a named author[1].

When the grammar is satisfying and has performed well on tests, the compiler construction begins. This process is illustrated in the blue box of fig. 4.1. Using the programming language C with the Flex and Bison code generating libraries, as described in Section 2.2.2, ease the reimplementation the parser for the reworked grammar. Additionally, these libraries allows for the storing and transforming of the data parsed in a practical data structure for later use. This data structure will later be used for constraint checks that are not possible to do by parsing. After this phase is completed a functioning syntax analyser, or parser, for the reworked grammar is produced. This is illustrated as the output of the blue box in Figure 4.2.

During the compiler development phase, small tests will be conducted until it is deemed fit for a full test on the PRiMuS dataset. These small tests will be meaningless pieces of "music" constructed by hand to test the incomplete state of the grammar at a given point. When the compiler supports the full alphabet then full tests will be run on the PRiMuS dataset, revealing how well the reworked grammar defines the intended language. Errors should be examined and fixed in order to ensure that the language model is as good as possible before moving on.

Errors will be discussed throughout the next chapters. There are multiple types of

[1] `https://rosettacode.org/wiki/Parse_EBNF#Haskell`

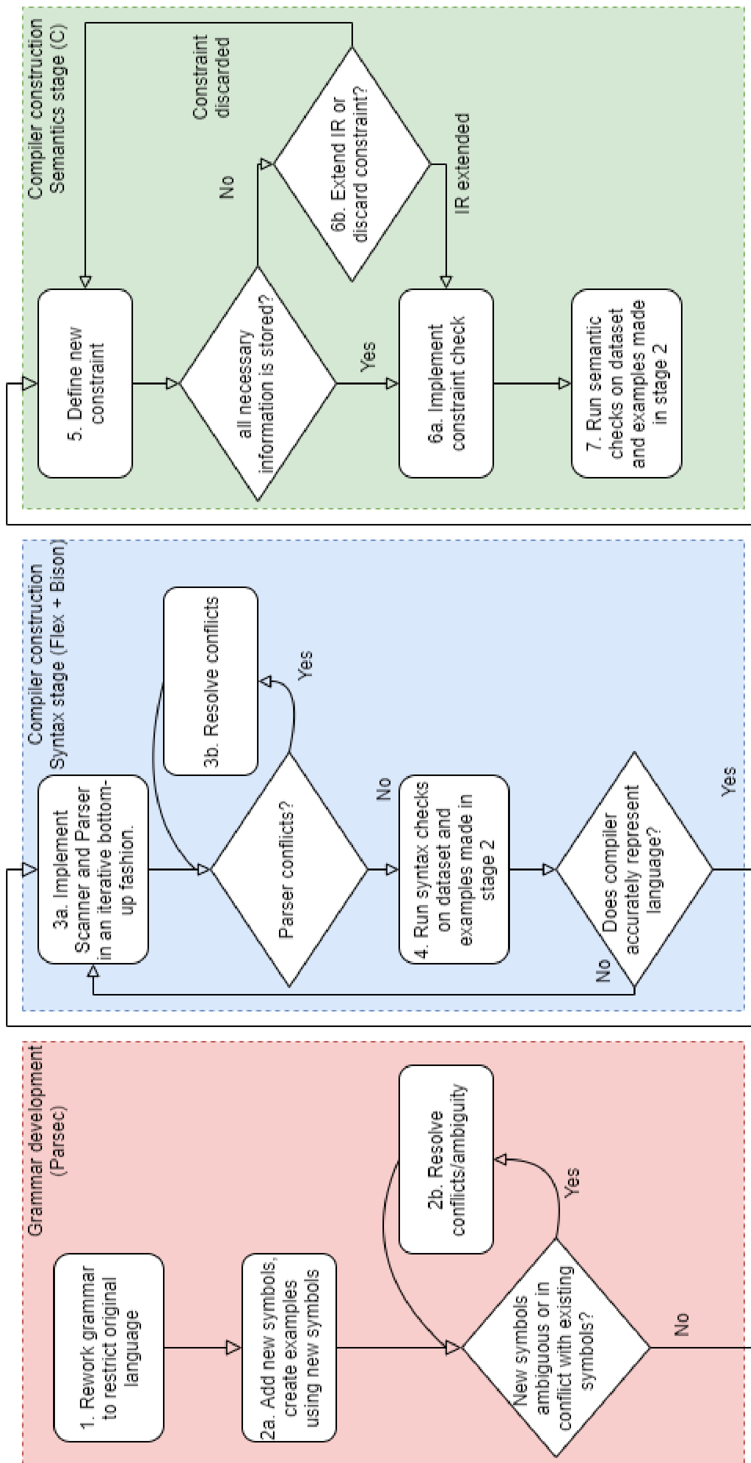


Figure 4.1: Development of the musical notation compiler, MNC

errors that are discussed. First there is the *Lexical Error*. This error can occur during lexical analysis, and denotes a string that is not part of the alphabet. Since the data set is generated with a well defined alphabet, these errors will not occur during tests on the data set as only words from the alphabet is used. Then there is the *Syntax Error*. A syntax error can occur during parsing and denotes a string that does not follow the rules of the grammar. This is analogous to a sentence that contains known words but breaks grammatical rules. Finally, there is the *Constraint Error*. These errors are similar to the syntax errors and occurs during the semantic analysis stage, also known as constraint checking. They denote a string that breaks rules of the language which could not be represented in the grammar. fig. 4.3 illustrates which errors may occur where.

Throughout, it is assumed that the ground truth representations of the PRiMuS data set are correct. Any error that occurs will thus be caused by a weakness or inaccuracy in the formal language that is developed in this thesis. There is one exception to this, which will be presented in Section 6.1 and discussed in Section 7.1. The errors that occur will be analysed by manual inspection if the number is low enough. If a high number of errors is seen then a frequency analysis is made on the error logs in order to identify the most prominent errors.

Up to this point, creating a parser has been the primary focus. When a good representation of the intended languages syntax is achieved, additional constraint checks can be implemented. This process is illustrated in the green box of Figure 4.1. The constraint checks will enforce the rules that could not be represented in the context free grammar. An example of such a constraint is checking whether or not a bar is properly filled. This is the semantic analysis stage which is the last stage of the compiler front-end. When this is done, a complete compiler front-end has been created, which is illustrated as the final white box in Figure 4.2. The PRiMuS dataset will be used to test the constraints as much as possible. In the case of a constraint that requires symbols or data that is not present in the PRiMuS data set, hand crafted examples are created and used.

Additionally, patterns and rules that require different expressive capabilities than those of the context free grammar are noted along the way. A study of formal grammar is then to be conducted in order to find ways of expressing these patterns with a formal grammar. The findings of this study will not be implemented in the MNC, but is illustrated using smaller hand-made examples. The constraint checking phase is the only thing that is not possible to embed in the context free grammar, as it is context sensitive. By studying other types of languages, it might be possible to embed the constraint checking in a different type of grammar. If this is successful, then the entire language definition can be contained within a grammar. If not, code or some other definition of the constraints will have to be kept as an important part of the language definition.

4.2 Software Design

During the course of this thesis, a compiler front-end is developed from scratch in C99 using Flex and Bison. The compiler front-end is comprised of several parts detailed below. The choice of both language and libraries were made for several reasons. Firstly, both Flex and Bison are acknowledged tools with plenty of documentation and users, which made them good candidates. Secondly, the authors familiarity with both C and the libraries saved

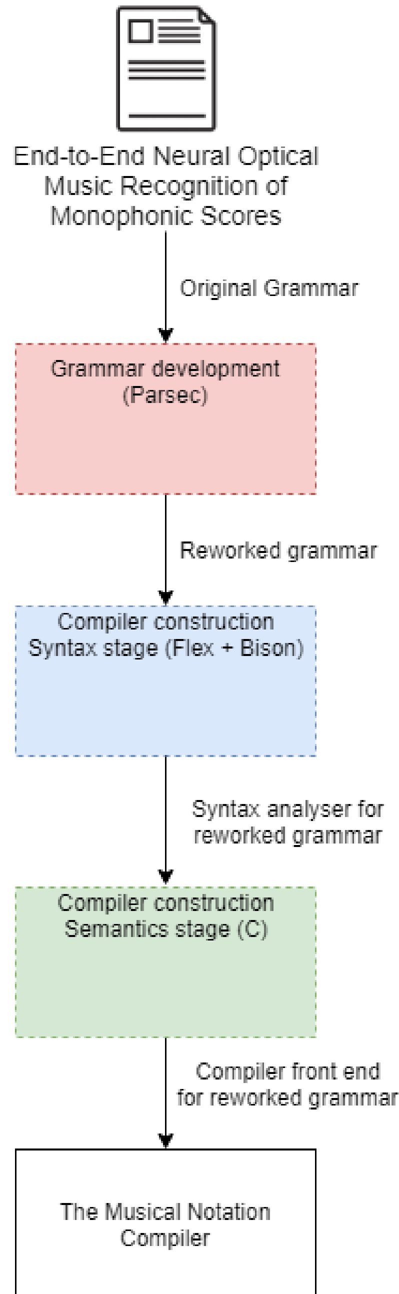


Figure 4.2: Intermediate goals of the development process for MNC.

time as it was not necessary to learn a whole new language or library. The use of C99 was chosen as it is a broadly used standard of C. While not the most portable one, such as ANSI C which is also known as C89, it offers a good middle ground between functionality and portability.

Firstly, there is the lexical analysis phase. This phase is implemented using the code generation library Flex, as described in Section 2.2.2. A lexical analyser scans the input string for matches of a predetermined set of patterns. in case of a match, it returns a token of the appropriate type. If the string is not matched, a lexical error is reported which indicates that an unknown string was encountered. A token is merely a symbol that describes the abstract classification of the matched string. For example, the string “note” is matched and a `NOTE` token is returned, stating that the matched string is a note.

Now the input string has been turned into a stream of tokens by the lexical analyser. This stream of tokens is passed to the syntax analyser, or parser, which is implemented by using the code generation library Bison as described in Section 2.2.2. The parser examines the token stream for patterns, as defined in the grammar it implements. Every token should be part of a pattern, and every out-of-place token will throw a syntax error. After parsing, the data is stored for later use as an abstract syntax tree.

Finally, the parsed data is subject to constraint checks implemented in C. This is the semantic analysis phase, which in this thesis is formalized as a set of constraints. During this phase, all data is assumed to be syntactically valid according to the grammar. The goal is to detect errors that could not be enforced by implementing a context free grammar. For clarity, these errors are called constraint errors, despite the fact that they represent a part of the syntax. It is known that programming languages often have context-sensitive aspects, despite their definition being a context free grammar, and this is also the case in musical notation.

A summary of what type of data are passed between which part of the program is shown in Figure 4.3.

4.3 Summary

This chapter presented the experiment and software design as well as some useful definitions and terminology. By defining a place to start the development, that is the Original Grammar, as well as an ultimate goal, the MNC compiler front-end, there is now a clear path from the start to end of development. Intermediate goals were defined, as well as what errors are to be encountered in each stage. These goals and errors will be discussed further in the coming chapters. Both the software- and experiment designs will be implemented in the following chapter.

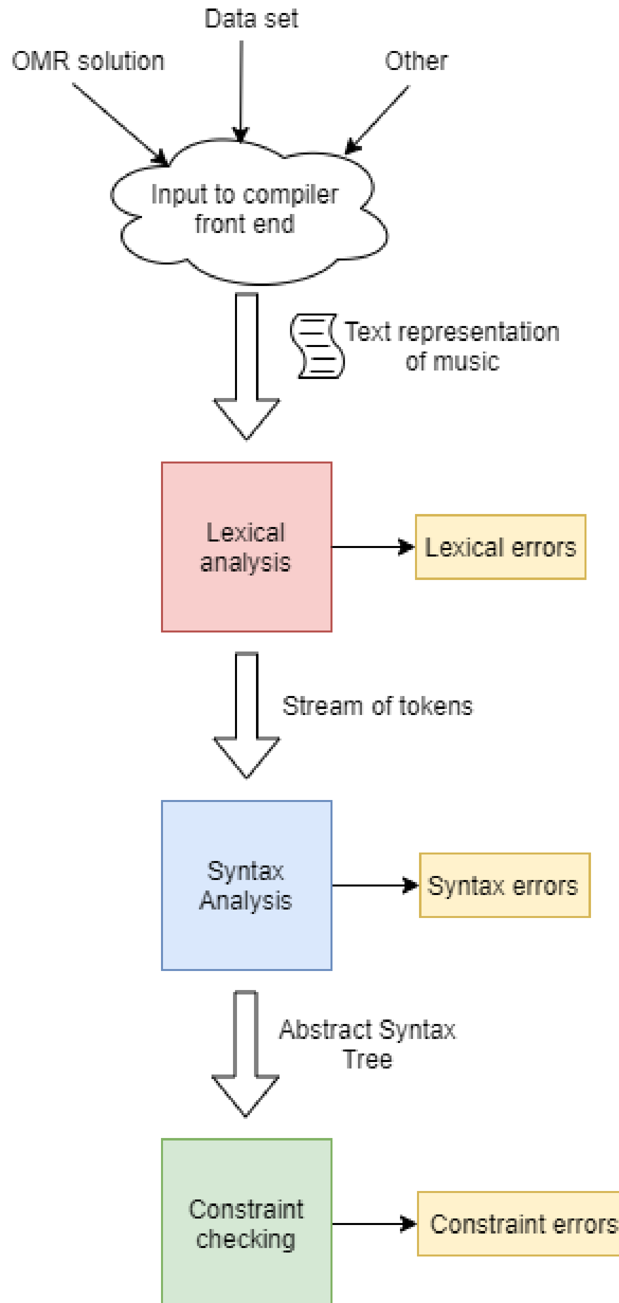


Figure 4.3: Diagram showing the data formats after each step as well as each type of possible error.

Chapter 5

Reworking the Original Grammar

The work done in this thesis is comprised of three parts. The first part, found in section 5.1, focuses on the Original Grammar which Calvo-Zaragoza and Rizo [10] defined to use in their OMR solution in their paper. This grammar will be reworked and extended so it may be used to validate the output of the OMR solution. The resulting grammar is the Reworked Grammar. Throughout this section, there will be a notion of a non-existent grammar G_{music}, which defines the intended language for music, $L(G_{music})$. The grammar itself and the language it produces is used for illustration purposes.

By using the new Reworked Grammar, a compiler front-end, MNC, will be implemented in Section 5.2. MNC will do both syntax and semantic analysis on the musical score. Semantic analysis is formulated as a set of constraints that must be upheld. These constraints could not be represented in the Reworked Grammar. The MNC will attempt to parse the musical notation to detect syntax errors and store it as an IR. This IR can then be used for semantic analysis which will detect constraint errors.

The last part, found in section 5.3, will study different types of formal languages that might be of interest when expanding the compiler. The motivation is to be able to fully define the language with just a formal grammar. The constraints that are checked during semantic analysis are in essence part of the languages syntax, but a context free grammar could not represent these constraints.

All put together, these tools and studies will allow the OMR-system made by Calvo-Zaragoza et al to use the Reworked Grammar in order to validate its results.

5.1 Representing Musical Notation

Calvo-Zaragoza et al defined two formats for representing musical scores [10], the Semantic- and Agnostic encoding. This project will only consider grammar for the Semantic Encoding, named the Original Grammar as specified in Section 4.1. The language defined by the Original Grammar represents the high-level representation of music, which is the focus of this thesis. The goal is to rework the context-free grammar presented in their paper so that it is possible to parse strings of musical notation with it, revealing any syntax errors.

Additionally, new symbols should be added to expand the musical repertoire the grammar can represent, showing its usefulness in a more complex musical setting.

The starting point for this thesis, the Original Grammar, is defined in EBNF and an excerpt from its original definition containing the relevant parts can be seen in Listing 5.1. Note the use of square and curly brackets to represent sequences and optional symbols, as described in Section 2.1.4.

Listing 5.1: An excerpt of the Original Grammar containing the most relevant parts in EBNF.

```
1   sequence = symbol sep symbol {sep symbol} .
2   symbol   = clef sepsymbol clefnote linenumber
3            | keysignature sepsymbol diatonic [accidentals]
               (major|minor)
4            | timesignature sepsymbol (metersigns | (integer
               slash integer))
5            | (note | gracenote) sepsymbol pitch sepvalues figure
               [dots]
6            | tie
7            | barline
8            | rest sepsymbol figure [dots] .
9   dots     = dot {dot} .
10  pitch    = diatonic [accidentals] octave .
11  figure   = "whole" | "half" | "quarter" | "eighth" [...] .
12  sep      = ' ' .
13  sepsymbol = '-' .
14  sepvalues = '_' .
```

This grammar can produce strings that represent musical notation. Figure 5.1 displays the musical notation example from Section 2.3 with a corresponding textual representation.

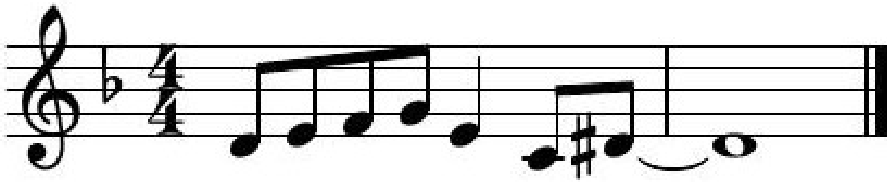
5.1.1 Motivation

The Original Grammar, G_{orig}, has two main issues. First of all, it was too shallow to enforce any structure representing a musical score. It is trivial to see in Listing 5.1 that it will accept any sequence of symbols from the alphabet. See Section A.1 for the complete grammar. This means that the grammar will accept valid strings representing musical notation consisting of symbols from its alphabet, but it will also accept all strings that use the same alphabet. Formally, the language of the Original Grammar, $L(G_{orig})$, is a strict superset of the intended language, $L(G_{music})$. This is illustrated in Figure 5.2.

A worked example

An example will illustrate the weakness of the Original Grammar: The axiom is `sequence`. This can be expanded into a list of symbols using the following production rule:

$$\texttt{sequence} = \texttt{symbol sep symbol \{sep symbol\}}$$



(a) A musical notation example.

```
1   clef-G2 keysignature-Dm timeSignature-4/4 note-D3_eighth
2   note-E3_eighth note-F3_eighth note-G3_eighth note-E3_quarter
3   note-C3_eighth note-D#3_eighth tie barline note-D#3_whole
4   barline
```

(b) A transcription of the musical notation example in (a).

Figure 5.1: The musical notation example from Section 2.3 with a transcription. Figure 5.1a displays the original musical notation. Figure 5.1b displays the transcription of Figure 5.1a.

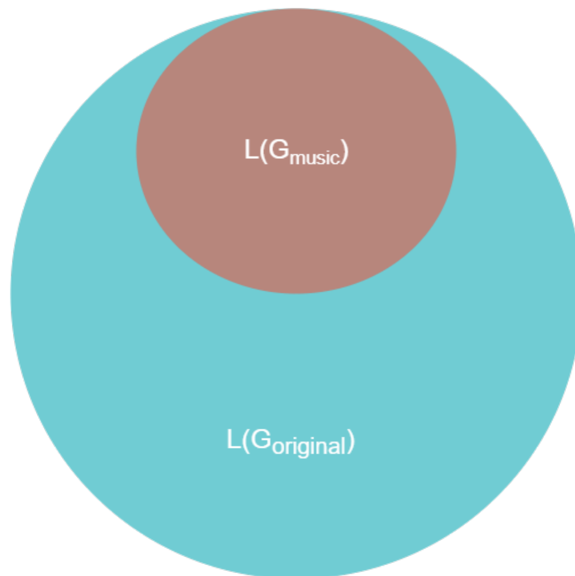


Figure 5.2: An illustration of the domains of the language defined by the Original Grammar (blue) in relation to the domain of the language of valid musical scores using the same alphabet as the Original Grammar (red).

This list of symbols may be consist of two symbols or expanded infinitely long, due to the use of curly brackets. Next, each of these symbols may become a bar line using the following production rule:

```
symbol = barline
```

Which produces the final string

```
barline sep barline sep barline [...]
```

Now, the string produced by the Original Grammar is an infinite number of bar lines, which is an obvious violation of the rules of musical notation.

Since an invalid string could be produced by the Original Grammar, then a parser implementing it would accept this string containing only bar lines as valid. This proves that it is not possible to use this grammar in order to validate results produced by an OMR-solution like the one presented in Calvo-Zaragoza et al [10].

5.1.2 Restrictions on the language

In order to only accept strings that describe valid pieces of music, it is necessary to know what defines it. This section will describe how the Original Grammar is transformed using domain knowledge of musical notation. Throughout this section, all grammars will be in the human readable Bison grammar format, presented in Section 2.1.4.

High-Level Structure of a Musical Score

As described in Section 2.3, a valid score is required to have a heading that features a clef, a time signature and a key signature as well as at least one bar with a sequence of either or both notes and rests. Without this, it is impossible to know what pitch a note has and how many beats there are in a bar. In order to create a grammar that is able to uphold this rule unambiguously, there are a few steps that must be taken.

First of all, the `symbol` non-terminal has too many productions, causing it to be used too often. Both notes, rests, bar lines and time- and key signatures are all included in the productions of `symbol`. Because of this, it is impossible to see the difference between a well-formed score and a string containing bar lines. The parser will see both valid and invalid strings as the same stream of identical tokens. In order to fix this, the `symbol` production is split into several non-terminals. From now on, the role of a `symbol` is to contain something with a duration, meaning only a note or rest. The definitions of the clef, time signature and key signature are taken out of `symbol` and put in their own respective non-terminals. A bar line or tie is not considered a symbol any more and will be brought back later. The result of this operation can be observed in the unfinished grammar in Listing 5.2. Note that all unfinished grammars shows an intermediate development stage and are for illustration purposes only.

Listing 5.2: Unfinished grammar from the grammar development phase in BNF. Irrelevant details are left out to clearly show the structure. The grammar features a `symbol` non-terminal with fewer productions and several new non-terminals

```
1   sequence : symbol
2            | symbol sequence
3
4   symbol   : note
5            | gracenote
6            | rest
7
8   clefcom  : clef clefnote linenumber
9
10  keysig   : keysignature
11
12  timesig  : timesignature
```

Next, a notion of a bar is necessary to correctly place the bar lines. This introduces a new non-terminal called the `bar`. A bar is simply a list of `symbol` ending with a `barline`. The former axiom `sequence` is put inside the `bar` and also renamed to `symbol_list` to more clearly explain what it is.

Now all the pieces needed to create a valid heading for the score are present. Creating a new axiom called `score` which starts with a heading and ends with one or more bars. This effectively implements the rules of musical notation requiring a heading and at least one bar containing symbols. Listing 5.3 illustrates what the `score` and `bar` productions looks like. In the Original Grammar, a `symbol` could be expanded into everything, at any point. This has now been fixed.

Listing 5.3: Excerpt from the grammar development phase in BNF. Irrelevant details are left out to clearly show the structure. It features a new axiom which has a single production that enforces a specific structure

```
1   score       : clefcom keysig timesig bars
2
3   bars        : bar
4               | bar bars
5
6   bar         : symbol_list barline
7
8   symbol_list : symbol
9               | symbol_list
10
11  symbol      : note
12              | gracenote
13              | rest
14
15  clefcom     : clef
16
17  keysig      : keysignature
18
19  timesig     : timesignature
```

Changing Clefs and Signatures

The grammars presented in Listings 5.2 and 5.3 have focused on the high-level structure for an entire musical score, but there are other rules that need to be enforced by the grammar.

The grammar presented in Listing 5.3 has no way of changing the clef, time- or key signature after the initial heading. This should be possible to do, but only in certain places. It is allowed to change the time- and key signature at the start of every bar, but not in the middle of a bar. Determining how many beats there are in a particular bar is impossible to do if the time signature could change in the middle, as this behaviour is not defined. Changing the key signature in the middle of a bar is a non-standard operation and is commonly solved using accidentals, not by an explicit change of key signature. To reflect this in the grammar, the `bar` production is expanded to include an optional time- and key signature at the beginning. This allows the optional change of a key or time signature at the beginning of each bar. Similarly, the clef may change in between every note. This is done by adding an optional clef before each element in the `symbol_list`. Listing 5.4 shows an excerpt of the Reworked Grammar, displaying the relevant productions.

Listing 5.4: Excerpt from reworked grammar in BNF showing the optional change of clefs between symbols and key and time signatures between bars. C-style comments denote empty production rules

```
1   bar         : opt_keysig opt_timesig symbol_list barline
2
3   symbol_list : opt_clef symbol
4               | opt_clef symbol symbol_list
5
6   opt_clef    : clef
7               | /* nothing */
8
9   opt_timesig : timesig
10              | /* nothing */
11
12  opt_keysig  : keysig
13              | /* nothing */
```

Multirests

The multirest symbol needs to be addressed on its own, as it is the only symbol in the alphabet that always spans an entire bar or more. Thus, it does not make sense to categorize it among the notes and rests which are all sub-bar symbols. Since a multirest always lasts at least one bar and is always the only symbol in the bars it spans, it is instead categorized as a type of bar. While technically a multirest may last for several bars, the classification as a type of bar makes sense from a graphical point of view. The multirest symbol is written as one symbol, in its own bar.

Listing 5.5: Excerpt from reworked grammar in BNF showing the `multirest` symbol and how it is used

```
1    bar        : opt_keysig opt_timesig symbol_list barline
2               | opt_keysig opt_timesig multirest barline
3
4    multirest  : MULTIREST_TOKEN SEP_SYM_TOKEN integer
```

Other Changes

A number of smaller changes to the Original Grammar were also made, for both the purpose of clarity and prevent ambiguity. Due to the liberal use of the hyphen and underscore, or `sepsymbol` and `sepvalues` respectively, there were many possibilities for ambiguities. A parser implementing a grammar can only look so far ahead, and cannot always know whether or not to reduce the token stream at a certain point or keep going. The rather extensive restructuring done in this section was an iterative approach, sometimes by trial and error, in order to disambiguate the grammar. Because of this, some of these changes are made simply to create a more robust grammar for future expansion and will not affect the language defined by the grammar.

The specification of a note or rests length was factored out of the note and rest productions, as it was essentially duplication. This lead to the introduction of the `length` symbol, which contains all information regarding the computable length of a note or rest. Fermatas are not part of this, as the extra duration they impose is subjective to the player and is therefore not computable. The production rule can be seen on line 13 in Listing 5.6.

Expanding on this, it is clear that the note and rest share some modifiers. Symbols like the fermata and tie are termed modifiers. They all have two things in common, they do not affect the pitch or base duration of a note or rest in an analysis and they are optional. A quarter note with a fermata is still considered a quarter note when checking whether or not the bar is filled, and a tied eighth rest is still an eighth rest. Therefore, these modifiers are grouped together in a symbol called `extensions` which can be applied to any note or rest. This also features the added benefit of grouping together many symbols separated by underscores into a single symbol, preventing ambiguity.

In order to further clarity of structure, some more refactoring is done to the note and rest symbols. The base of each symbol is put in its own production rule and the top-level note and rest symbols are transformed into the combination of the base and extension. An excerpt of the reworked grammar that shows the new productions can be seen in Listing 5.6.

Listing 5.6: Excerpt from the reworked grammar showing grammatical changes to the note and rest symbols

```
1  symbol  : note
2          | rest
3
4  note    : note_base extensions
5          | note_base
6
7  rest    : rest_base extensions
8          | rest_base
9
10 rest_base : REST_TOKEN SEP_SYM_TOKEN length
11
12 note_base : NOTE_TOKEN SEP_SYM_TOKEN pitch SEP_VALUES_TOKEN length
13           | GRACE_NOTE_TOKEN SEP_SYM_TOKEN pitch SEP_VALUES_TOKEN
               length
14
15 extensions  : SEP_VALUES_TOKEN ties SEP_VALUES_TOKEN fermata
16             | SEP_VALUES_TOKEN fermata
17             | SEP_VALUES_TOKEN ties
18
19 length : figure dots
```

5.1.3 Adding New Symbols to the Reworked Grammar

As stated in Calvo-Zaragoza et al [10], the Original Grammar is not musically exhaustive but serves as a convenient starting point from which to build more complex systems. While it would have been trivial to extend the Original Grammar with new symbols, the added complexity and structure of the reworked grammar could have complicated this process. To prove that extending the grammar is still an easy task due to the new structure, some more symbols are added. To name a few examples of missing symbols, there is no support for chords, tuplets, indication of dynamics, multiple staves, lyrics etc. This drastically limits the amount of music the grammar can represent. The Original Grammar may only represent monophonic, single stave scores with a limited rhythmic and non-existing dynamic vocabulary. A small selection of symbols, based on a subjective evaluation of the impact on expressiveness versus complexity of implementation, are added to the grammar. The new symbols are the chord and the tuplet. Additionally, the tie is being redone which, in essence, makes it a brand new symbol in the context of the development of the grammar.

The additions also lead to the development of some interesting properties of the grammar and the classes of music it can represent. Note that all symbols added in this section are not present in the PRiMuS dataset, so testing is less thorough as only handcrafted experiments are conducted. Adding the new symbols should not affect the existing symbols, with the exception of the tie, so that the grammar stays compatible with the PRiMuS dataset. By extending the alphabet of the grammar, thus extending the amount of musical

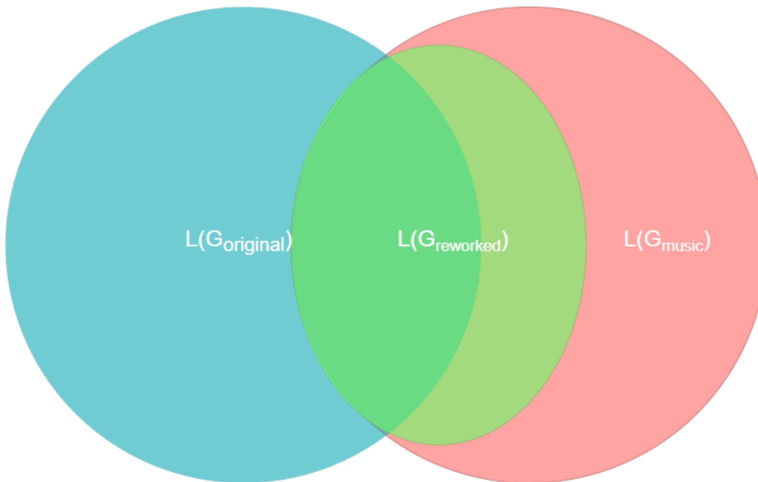


Figure 5.3: An illustration of the domains of the language defined by the reworked grammar (green) in relation to the Original Grammar (blue) and the language of all valid musical scores (red)

constructs it can represent, the goal is to create a grammar that may represent every valid string the Original Grammar could represent in addition to strings containing the newly added symbols. This brings the Reworked Grammar closer to the grammar of the intended language, G_{music}, as shown in Figure 5.3.

Chords

The first symbol to add is the chord. The chord is an interesting case for two reasons. Firstly, it poses the problem of serialization: the notes of a chord could be read in any order. Figure 5.4 illustrates this problem. Some proposals for this were made in Alfaro-Contreras et al [2]. The proposals were the remain-at-position character encoding which is inspired by typewriters, causing the OMR solution to stop reading to the right but rather upwards and the parenthesized encoding which encapsulated vertically aligned symbols. These proposals were made to the agnostic representation, which makes them consider the graphical point of view like most of the previous grammars in the related work, and not to the Original Grammar which considers the high-level musical semantics.

Secondly, the introduction of a structure such as the chord will move the class of representable music from monophonic scores to homophonic scores, or even polyphonic scores, according to the taxonomy described in Section 2.3.2. Depending on how it is implemented, this could lead to a vast increase in the language domain. Polyphony may be implemented as a monophonic score with the added chord structure, to allow the representation of notes that occur at the same time. Assuming the use of a parenthesized approach, as proposed earlier, a score like shown in Figure 5.5 could very well be represented as shown in Listing 5.7 without any other additions.

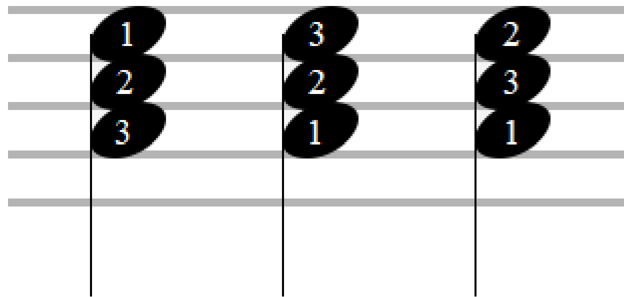


Figure 5.4: A chord which is serialized in three different ways

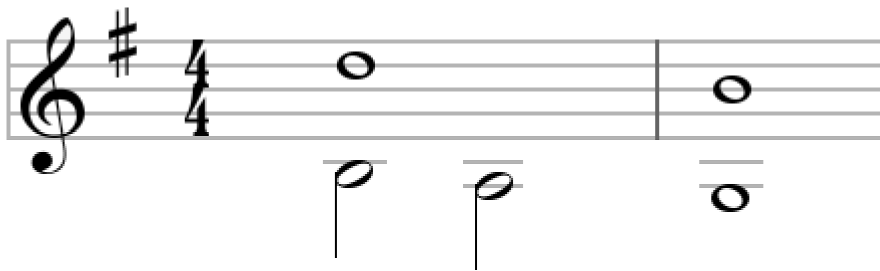


Figure 5.5: Example of a simple, single staff polyphonic score

Listing 5.7: Possible transcript of the polyphonic piece in Figure 5.5 using parenthesized encoding of chords as the only addition to the reworked grammar

```
1 clef-G2
2 keySignature-GM
3 timeSignature-4/4
4 (note-D4_whole note-B2_half)
5 note-A2_half
6 barline
7 (note-B3_whole note-G2_half)
8 barline
```

Unfortunately, chords and polyphony are still different things. Chords played by one single instrument are denoted by the graphical chord symbol, while a polyphonic score may *form* chords by having several voices play or sing at the same time without using special symbols. Specifically, the chord symbol has several note heads connected by a single stem, while a polyphonic score forming a chord may choose not to. The score shown in Figure 5.5 does not group the chords by a single stem and therefore does not consist of any chord symbols. This distinction of what the musical symbol of a chord is, and how polyphony may form chords *without* using the chord symbol, is the background of the choices made in this grammar.

A chord is treated as a distinct symbol in the grammar. It features several pitches that

may or may not be different, as some instruments are able to play the same pitch multiple times simultaneously. The duration of a chord is uniform, meaning that the base duration of each note in the chord is the same. The apt reader may have already spotted the fact that the notes in Figure 5.5 do not all have stems, so it would be impossible to unite them with a single stem without altering the semantics. Other advantages of implementing chords as a separate symbol include not-yet-implemented symbols like the arpeggio. The arpeggio is a wavy line drawn next to a chord, indicating that the notes of the chord are to be played one after another while being held. It would not make sense to place an arpeggio symbol next to a chord made up of two voices in a polyphonic context. This causes the class of musical scores the grammar can represent to be homophonic, not polyphonic, since it is limited to one voice that can play several notes at once to build up chords.

The grammar is extended with the definition of a chord, as seen in Listing 5.8. An implementational choice was made to create a separate `pitch_list` for the chord for clarity, but this was not necessary. Note that the serialization problem is not addressed, as this solution accepts the notes of the chord in any order. Since the chord has a uniform length, it fulfils the requirements posed by the `symbol` non-terminal in order to be part of its productions.

Listing 5.8: Excerpt from the Reworked Grammar displaying the relevant parts for the newly introduced chord symbol

```
1 symbol  : note
2         | rest
3         | chord
4
5 chord   : CHORD_TOKEN SEP_SYM_TOKEN length pitch_list
6
7 pitch_list : SEP_SYM_TOKEN pitch pitch_list
8            | SEP_SYM_TOKEN pitch
```

Tuplets

The tuplet is a rhythmic symbol that adds depth in terms of representing rhythms. A tuplet divides m beats into n equal subdivisions. This means that it is possible to easily transcribe the rhythm of three notes of equal length being played during the time span of one beat. Transcribing a tuplet as an equivalent rhythm without the use of tuplets may be either impossible, unreadable or both. While most tuplets are named after the n, a seven-tuplet for example, there are some exceptions due to their common use. The triplet and quintuplet, indicating an n of 3 and 5 respectively, are by far the most common.

A tuplet has several equivalent notations. What they all have in common, is that they group a number of notes together and indicate the n subdivisions of the m beats. The m is not always specified and may have to be deduced from other pieces of information, such as the durations of the notes.

There are an infinite amount of ways to use the tuplet. Basic use of a triplet (tuplet with $n = 3$) may consist of three equal length notes. Any of these notes may be replaced with a rest, combined into a single note of a longer duration or split into more notes with

a total duration equal to the replaced notes. Tuplets are even possible to place inside other tuplets, forming nested tuplets as depicted in Figure 5.6.

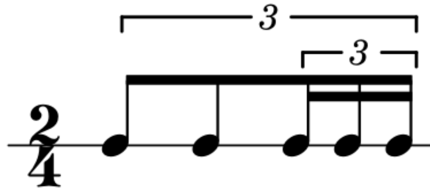


Figure 5.6: Nested Tuplets

Due to this freedom, the tuplet is implemented as an annotated encapsulation. This means that the entire contents of the tuplet is put inside a start and an end symbol. The start symbol also specifies the n and m parameters. Inside the tuplet, the contents are simply defined as a `symbol_list` which is the same non-terminal that is used inside a `bar`. Since the tuplet has a well-defined length, it is part of the `symbols` production rules.

Listing 5.9: Excerpt from the reworked grammar displaying the relevant parts for the newly introduced tuplet symbol

```
1  symbol : note
2         | rest
3         | chord
4         | tuplet
5
6  tuplet : TUPLET_TOKEN SEP_VALUES_TOKEN integer symbol_list
       END_TOKEN
```

Ties

Finally, the tie symbol binds two notes of equal pitch together, extending the duration of the first note by the duration of the second note. This thesis does not consider the slur, which is a similar looking symbol but with a different set of rules and meaning. The `tie` symbol has effectively been removed during the rework in order to reimplement the same functionality with new symbols. As the `tie` symbol was originally one of the many productions of `symbol`, it could appear in nonsensical places. For example, it was possible to find a tie between a clef and a rest which is in violation of musical notation rules. By having the `tie` symbol appear independently in between the symbols it ties together, a lot of flexibility is lost. It should be possible to tie notes that are not temporally adjacent, but this is impossible by the use of a stand-alone `tie` symbol without any connection to any other symbols. As stated in Calvo-Zaragoza et al [10] each symbol, including the tie, should be self-contained. Therefore, in order to allow two notes that are not adjacent to be

tied together, a new solution must be devised.

Additionally, the introduction of the `chord`, which in turn caused the category of representable scores to become homophonic, further complicated this. Now that multiple notes may be played at the same time, it is no longer unambiguous which notes are tied together. Consider the following example: A score as depicted in Figure 5.7 is transcribed as in Listing 5.10 using the new `chord` symbol.

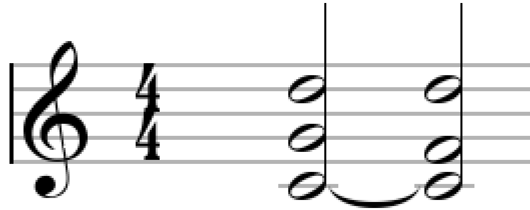


Figure 5.7: An example score illustrating the tie problem

Listing 5.10: An ambiguous transcription of Figure 5.7 using a tie

```
1 clef-G2
2 keysignature-CM
3 timeSignature-4/4
4 chord-half-C3-G3-D4
5 tie
6 chord-half-C3-F3-D4
7 barline
```

How would one, by only looking at the transcription, determine which note in the chords are tied together? It is also equally feasible that two, or even more, notes from each chord are tied together. Clearly, some valuable information is lost using this approach.

Instead of keeping the tie as a stand-alone symbol, it is proposed to include the tie in the category of modifiers for a note and rest. Specifically, this means that every note or rest should indicate whether or not it is tied. By placing this piece of information in each note and rest, it is unambiguous which notes are tied together. Listing 5.11 showcases the new proposed notation for this and it is indeed clear where the tie starts and ends, even though the notes are not adjacent.

Listing 5.11: An example of the new way of defining ties

```
1 clef-G2
2 keysignature-CM
3 timeSignature-4/4
4 note-C4_quarter_tiestart
5 note-G4_half
6 note-C4_quarter_tieboth
7 barline
8 note-C4_whole_tieend
9 barline
```

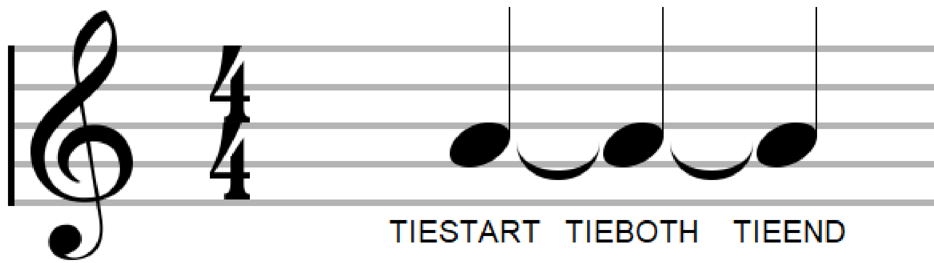


Figure 5.8: Illustration of the three classes of ties, with labels to help identify them.

During the rework described in Section 5.1.2, the `tie` was removed from the list of possible productions for a `symbol`. The newly introduced `extensions` symbol featured a, currently undefined, symbol called `ties`. The `ties` symbol has three productions, namely the `TIESTART`, `TIEBOTH` and `TIEEND` productions. `TIESTART` and `TIEEND` denotes the start and end of a tie, respectively. `TIEBOTH` represents the combination of both, as ties may extend across several notes, like illustrated in Figure 5.8. The relevant production rules are shown in Listing 5.12

The complete Reworked Grammar is listed in Appendix A.2.

Listing 5.12: Excerpt from the reworked grammar in BNF showcasing the tie productions

```
1 extensions: SEP_VALUES_TOKEN ties SEP_VALUES_TOKEN fermata
2           | SEP_VALUES_TOKEN ties
3
4 ties: TIESTART_TOKEN
5      | TIEEND_TOKEN
6      | TIEBOTH_TOKEN
```

5.2 Semantic Analysis

The previous section described how the MNC will reject syntactically invalid scores, but it is still possible to create invalid scores with valid syntax. Examples of this are bars that are not properly filled and ties without a start or end. The following section will detail the set of constraints that make up the semantic checking phase. These constraints will be implemented to detect more errors after a successful syntax analysis phase.

5.2.1 A Syntax Directed Translation

During syntax analysis, each production rule has an *action* attached to it, making it a syntax-directed definition [1]. These actions are arbitrary pieces of code, which in this project has been used to extract data from the input and create an intermediate representation of the musical score. This is a strategy commonly seen in compilers. The intermediate representation used in this thesis is an AST, as described in Section 2.2.

Listing 5.13: Definition of a node in C

```
1  typedef struct node {
2     node_index_t type;
3     void *data;
4     size_t n_children;
5     struct node **children;
6  } node_t;
```

During the syntax analysis, or parsing, the action attached to a production rule creates a node. A node is defined as a C-structure containing a type, a data pointer and a list of children and its length. See Listing 5.13 for the definition in C. During creation, the `type` field is set to reflect what production rule was used. The `data` and `children` pointers are optional, as some nodes are superfluous while others represent something by simply existing. Using the `children` pointers of each node, it is possible to connect them together and form a traversable tree. Leaf nodes have no children, which causes `n_-children` to be set to 0 and the `children` pointer not to be initialized. The data pointer is used to capture raw data associated with the node. For example, a `PITCH` node does not have any data because it can use its children to deduce its data. Its children, however, have raw data. Typically, only leaf nodes have raw data, but there are a few exceptions. An `INTEGER` node has the integer value saved in its data field and has no children. The `BAR` node has its bar index saved in its data but is not a leaf node. Controversely, the `MAJOR` and `MINOR` nodes are leaf nodes without any data.

Due to several reasons, there will be superfluous nodes in the AST. Because of the several optional structures in the reworked grammar from Section 5.1, there has to be constructed a node for it regardless of whether or not the optional rule was used. This leads to empty nodes without any useful information. Since the grammar is implemented in Bison which can represent a BNF-like grammar, all lists will be recursively constructed forming linked lists. This is a representation that adds verbosity without giving any benefits to this solution. Similarly, dots are not counted during parsing but rather compiled into a

list. Other nodes may provide no benefits other than having a single child. These may be removed entirely, after moving the child of the node to the parents' list of children. The `SYMBOL` node is an example of this.

These superfluous nodes are candidates of *pruning*. Pruning is the act of simply removing unnecessary verbosity and is implemented in four steps. First, the tree is traversed and superfluous optional nodes are removed. An optional node without a child means that the optional production was not used, and the node is indeed superfluous. This may only happen to a key signature, time signature and clef. The other class of superfluous nodes are the `SYMBOL` node and optional key- and time signature nodes which were used. Since the only interesting part of these nodes are their single child, as discussed in the paragraph above, it is always safe to remove them and put their child in their position. At the end of this step, there should not be any `SYMBOL` nodes or any optional nodes.

The second step removes the verbosity of linked lists. By flattening the linked list structures, it is possible to represent a list as a single top-level list node with a pointer to all of its children without having to traverse every other child to reach the last child. This step is done by traversing the tree in a depth-first fashion and identifying all list nodes. If a list with more than one child is found it means that the list has another list node among its children. Then, the current node must add the contents of its child's list to itself. This algorithm is showed in Algorithm 1.

Algorithm 1: Flatten Linked Lists

input : Abstract Syntax Sub Tree node
output: Flattened Abstract Syntax Sub Tree node
$root \leftarrow$ input node
for $node \leftarrow$ *root.children* **do**
    flatten list of node.children
    if *root is list and root.n_children* > 1 **then**
        if *node is list* **then**
            root.children $\leftarrow$ (*root.children* $\cup$ *node.children*) $\setminus$ node
        end
    end
end

The third step is similar to the second step but is specialized for dots. Dots are initially represented as linked lists with no data in them, as this was practical during parsing. The final representation should be a node containing an indication of whether or not there are dots. If there are dots, the data field should contain the number of dots. To do this, a routine similar to the flattening of lists described in Algorithm 1 is utilized. It differs by the fact that it does not merge the elements while discarding the superfluous nodes, but rather only discards the nodes while counting how many are discarded. When it is finished, the counter is set to the amount of dots that were present. The details are presented in Algorithm 2.

Finally, the last step is not mandatory but was considered helpful enough to be included. The grammar supports special symbols for common and cut common time signatures. This means that the time signatures of $\frac{4}{4}$ and $\frac{2}{2}$ are represented as the strings `C` and `C/` respectively. As the fraction of the time signature will be used for semantic analysis

Algorithm 2: Count Dots

```
input  : Abstract Syntax Sub Tree node
output: Amount of dots as an integer
root ← input node
if node.type is DOTS then
    if node.n_children == 1 then
        return 1 + count dots of node.children[0]
    end
else
    return 0
end
```

later, it is useful to convert the special symbols to numerical fractions. The AST is scanned for occurrences of these special symbols. If they are found, they are simply replaced by the equivalent fraction.

After the AST has been processed, it is easier to use it for further analysis. To show the effect of the processing an illustrative example is made in Listing 5.14. This example includes most of the steps described in this section. After parsing this example, the resulting AST before and after pruning is displayed in Figure 5.9.

Listing 5.14: A syntactically correct string which is still invalid. The bar is underfilled and the ties do not match

```
1 clef-G2
2 keySignature-AbM
3 timeSignature-3/4
4 note-Eb5_eighth
5 note-C5_sixteenth_tiestart
6 note-D#5_sixteenth.._tieend
7 barline
```

5.2.2 Constraints

Using the processed AST, it is easier to perform constraint checks on it. The constraint checks enforce syntax rules that were not possible to represent in the grammar. After an iterative process of proposing constraints that are relevant and feasible to implement, two candidates were chosen to be implemented in this project. These candidates are checking whether or not bars are properly filled and checking that all ties are matched properly. Both of these candidates were chosen because the AST contained all relevant information, so the parsing process did not need to be altered. They both also represent their own, potentially large, class of errors. The example in Listing 5.14 showcases both of these errors.

```
1   SCORE
2     CLEFCOM
3       DIATONIC(G)
4       INTEGER(2)
5     OPT_KEYSIG
6       KEYSIG
7         DIATONIC(A)
8         ACCIDENTALS(b)
9         MAJOR
10    OPT_TIMESIG
11      TIMESIG
12        INTEGER(3)
13        INTEGER(4)
14    BARLIST
15      BAR
16        OPT_KEYSIG
17        OPT_TIMESIG
18        SYMBOLLIST
19          OPT_CLEF
20          SYMBOL
21            NOTE
22              NOTEBASE
23                PITCH
24                  DIATONIC(E)
25                  ACCIDENTALS(b)
26                  INTEGER(5)
27                LENGTH
28                  FIGURE(0.125000)
29                  NODOT(0)
30          SYMBOLLIST
31            OPT_CLEF
32            SYMBOL
33              NOTE
34                NOTEBASE
35                  PITCH
36                    DIATONIC(C)
37                    ACCIDENTALS(n)
38                    INTEGER(5)
39                  LENGTH
40                    FIGURE(0.062500)
41                    NODOT(0)
42                EXTENSION
43                  TIESTART
44            SYMBOLLIST
45              OPT_CLEF
46              SYMBOL
47                NOTE
48                  NOTEBASE
49                    PITCH
50                      DIATONIC(D)
51                      ACCIDENTALS(#)
52                      INTEGER(5)
53                    LENGTH
54                      FIGURE(0.062500)
55                      DOTS
56                        DOTS
57                          NODOT
58                  EXTENSION
59                    TIEEND
```

```
1   SCORE
2     CLEFCOM
3       DIATONIC(G)
4       INTEGER(2)
5
6     KEYSIG
7       DIATONIC(A)
8       ACCIDENTALS(b)
9       MAJOR
10
11    TIMESIG
12      INTEGER(3)
13      INTEGER(4)
14    BARLIST
15      BAR
16
17
18        SYMBOLLIST
19
20
21          NOTE
22            NOTEBASE
23              PITCH
24                DIATONIC(E)
25                ACCIDENTALS(b)
26                INTEGER(5)
27              LENGTH
28                FIGURE(0.125000)
29                NODOT(0)
30
31
32
33          NOTE
34            NOTEBASE
35              PITCH
36                DIATONIC(C)
37                ACCIDENTALS(n)
38                INTEGER(5)
39              LENGTH
40                FIGURE(0.062500)
41                NODOT(0)
42            EXTENSION
43              TIESTART
44
45
46
47          NOTE
48            NOTEBASE
49              PITCH
50                DIATONIC(D)
51                ACCIDENTALS(#)
52                INTEGER(5)
53              LENGTH
54                FIGURE(0.062500)
55                DOTS(2)
56
57
58            EXTENSION
59              TIEEND
```

Figure 5.9: A comparison between an unprocessed AST for Listing 5.14 and the same AST after pruning. The untouched nodes keep the same line numbers, causing the empty lines in between to clearly show the removed verbosity.

Checking whether or not a bar is properly filled

The first constraint to be checked is whether or not the bars are properly filled. This is an important rule in music notation. By checking this, both overfilled and underfilled bars will be revealed. Since this is a basic rule of musical notation, it is safe to assume that the input score has properly filled bars. This means that by revealing an error like this gives a hint to what could have gone wrong in earlier stages. If a bar is overfilled, it could be because of a bar line that has not been detected by the OMR system. This is especially apparent when the length of the overfilled bar is two times what it should be. In the case of an underfilled bar, it is possible that a note or a rest has not been detected by the OMR system. Outputting which bar is not properly filled will help researchers determine the error by giving them these hints.

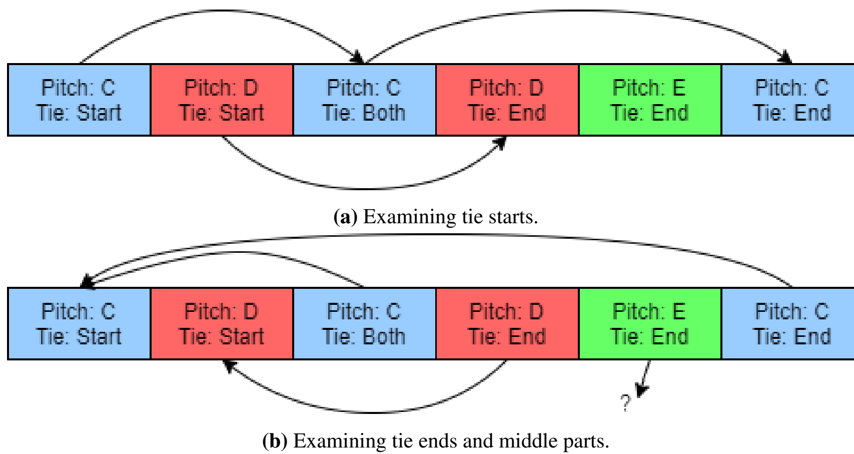
Implementing this constraint is straight forward. The produced AST is structured in a way that all relevant information is found within each respective bar. By traversing the AST and keeping track of the last seen time signature, it is only a matter of summing up the durations of every symbol in a bar for every bar. When a time signature is recorded, the constraint is updated because the time signature represents the target sum. The target sum is simply calculated as $\frac{n}{m}$, where n and m are the numerator and denominator of the time signature. For example, a time signature of $\frac{4}{4}$ is calculated to be 1 and a time signature of $\frac{6}{8}$ is calculated to be 0.75. Next, the different durations for notes and rests are mapped from their string representation to a numerical representation. A whole note is mapped to 1 and a quarter note is mapped to 0.25. This gives the general formula of an nth note which is mapped to $\frac{1}{n}$. By traversing a bar and finding all the notes and lengths, it is trivial to sum them up to compare against the target sum. Dots also need to be considered. Recall from Section 2.3 that a single dot adds half of the duration of the symbol. Mathematically, this means multiplying the duration by a factor of 1.5 in the case of one dot. Each symbol may have several dots, increasing the factor by $\frac{1}{2^i}$ for every $dot_i \in dots$. This leads to Equation (5.1) representing the calculation of the duration of a dotted symbol.

$$duration = base * \sum_{i=1}^{n} \frac{1}{2^i} \tag{5.1}$$

Grace notes are not considered in this calculation, as their duration should not count towards the target sum.

Checking that ties are matched

The second constraint is tie matching. Recall that a tie is now represented as an optional extension of a note that can be either a tie start, tie end or a middle part. For each pitch, a tie start has to come before an end or a middle part. Likewise, a middle part has to be in between a start and an end, and there may be multiple middle parts. By checking that the ties are in the correct order and are properly started and terminated, it is again possible to uncover potential detection errors in an OMR context. If a tie start is missing, but it has a middle part and an ending, it is clear that the OMR process has failed. A tie without a start would simply be impossible to draw on paper, thus it is safe to conclude that this is an error caused by the OMR system.



(a) Examining tie starts.

(b) Examining tie ends and middle parts.

Figure 5.10: An illustration of the tie matching algorithm. Each box is a note with a tie. The pitches are color coded. In Figure 5.10a each note with a tie start examines the following notes to detect errors. In Figure 5.10b the rest of the notes see if they have previously been checked. Notes that have not been examined, like the green box, are reported as errors

A novel algorithm to detect these kinds of errors was devised. The AST is scanned for nodes with a tie. These nodes are stored in an array for later use. When all nodes are found, the algorithm only considers the array of nodes with ties. By iterating through the array, each node is verified differently based on what kind of tie it has. The tie matching algorithm is illustrated in Figure 5.10.

A node with a tie start will look ahead in the array and will only consider nodes with the same pitch as itself. This is illustrated in Figure 5.10a. By keeping track of how many starts and ends it has seen, it can judge whether or not the ties are in order. If the node sees a new tie start when the number of starts is greater than the number of ends, it will immediately report an error as it does not make sense that two notes of the same pitch start a tie without ever ending them. If a middle part is seen when the number of starts is greater than the number of ends, no action is made. If the number of starts and ends are equal, an error is reported as this means the middle part is not in between a start and an end. At the end of the iteration, the number of starts and ends are compared to see if they are balanced.

The look-ahead procedure by the nodes with a tie start will catch errors within their scope. A node with a middle part or a tie end will look backwards in the array. To check for errors, it is simply needed to see if a node with a tie start has included the current node in their error check. This is illustrated in Figure 5.10b. If a previous node has included the current node, any errors would have already been reported and the algorithm may move onto the next node. In the case of a node that not been included in a previous search, an error will be reported as this means there is a middle tie or tie end with no start.

5.3 Applicability of Other Formal Languages

In order to extend the capabilities of the MNC compiler front-end, a study of formal languages was conducted. The goal is to extend the capabilities of the syntax analysis phase. This means that the goal is to try to include the constraints in the formal grammar instead of implementing the constraint checks in code like in the previous section. If the constraint checks are embedded in the grammar, then the constraint checking code is unnecessary. Without the constraint checking code, then the complete language definition is contained within the grammar itself. With all details embedded in the grammar, then it is more valuable by itself as it does not need to depend on some reference implementation.

The language study is not implemented as a parser and thus not going to be tested on the PRiMuS data set. It is meant to provide ideas for potential future work that will expand on the ideas presented in this thesis on a theoretical level.

5.3.1 Criterion and Evaluation

Each type of formal language is evaluated by the use of examples and fictional scenarios where they might provide a more elegant solution to a problem than other types of languages. Each language will be presented with a technical description, a worked example to illustrate their workings, strengths and weaknesses and a final evaluation. Using the worked example, the languages' restrictiveness and generative capabilities are evaluated. Each language will also have its practicality and technical complexity evaluated. A sought-after property is the ability to solve a counting problem. The reason why will be illustrated by example later. The counting problem refers to being able to represent a production of a string that is defined as S^n given an n. That is, given an n, repeat something n times. The fact that it needs to be given an n implies some context sensitivity. Each language will have their counting abilities evaluated in particular.

An Example of the Usefulness of a Counting Grammar

A grammar that is able to solve problems involving counting would be a useful extension to the MNC front-end. Because of this, the following illustrative example will be presented to show the usefulness of such a property. As of now, the MNC checks whether or not the bars are properly filled by running analyses on the AST. Instead of having to sum up the durations for all symbols inside a bar to check whether or not it is properly filled, the counting property of a grammar could do this by syntax analysis. There are, of course, other use cases for this counting property as well. Recall the production of a bar that is used in the reworked grammar from Section 5.1.2:

bar : opt_keysig opt_timesig symbol_list barline

If a hypothetical grammar was able to solve a counting problem, and always knew the most recent time signature, it would be possible to solve this by parsing alone. If the production created n beats and a bar line, a strategy similar to what is found in the buddy memory allocation system [28] could be used. By allowing each beat to split itself in two, one could mix and match the available duration to fit the input. If there are non-terminals left

when input ends, there is not enough notes. If there are no more non-terminals but still more input, there are too many notes.

Using a hypothetical type of grammar that supports this by the use of parameters, one could define such a rule like this:

$$\begin{array}{lccc} (0) & \text{bar(last timesig } n) & \rightarrow & (\text{beat}(1))^n \text{ barline} \\ (1) & \text{beat(duration } n) & \rightarrow & \text{beat}(\frac{n}{2}) \text{ beat}(\frac{n}{2}) \end{array}$$

If the time signature is $\frac{3}{4}$, the production above would yield

beat(1) beat(1) beat(1) barline (0)

Given an input consisting of 2 quarter notes and 2 eighth notes, the above would split itself up to become

beat(1) beat(1) beat(0.5) beat(0.5) barline (1)

Which is an accurate rhythmical representation that will catch these errors purely by syntax analysis. The next step would be to fit in the actual notes with pitches and modifiers in these illustrative productions rules.

5.3.2 Context Free Grammars

Context Free Grammars (CFGs) are a very common type of grammar in the field of computer science, due to its expressive capabilities and simplicity. Originally defined by Noam Chomsky in the field of linguistics [15], computer scientists later adopted the context free grammars to represent the syntax of programming languages [27, 48] and data formats [9] to name a few.

The earliest use of context free grammars in computer science is by John Backus in his work on the International Algebraic Language(IAL), later renamed ALGOL(Algorithmic Language), in his 1958 report on the syntax of the proposed language IAL [4]. In the report, John Backus first informally describes the language with examples and descriptions and then formally describes it using a notation that will be known as the Backus-Nauer Form (BNF), which is described in Section 2.1.4. The formal definition of the language in this paper only contained the syntax rules of the language, the semantics were added later due to time restrictions. After this report, using the BNF-notation to write context free grammars has become the standard. Also, context free grammars became more and more common to use to define programming languages, as many programming languages show few or no context-sensitive properties.

Context Free Grammars and languages are, as previously stated, a Chomskian language. In the Chomsky Hierarchy, it is the type-2 language, encompassing the regular type-3 languages. Using the BNF-notation, a rule in a context free grammar is always on the form [55]

$$A \rightarrow \alpha$$

where α can be any, possibly empty, string.

Informally, this means that the non-terminal symbol A is not affected by adjacent symbols (hence the term context **free**). The left side of the production rule, α, can be any string consisting of zero or more terminals and non-terminals.

In contrast, regular languages, which are type 3 in the Chomsky hierarchy, have stricter rules regarding the left side. Regular languages may only expand in one direction, so there exists both right- and left-regular languages. Formally, this means that all production rules are on the form

$$A \to aB$$

if the language is right-regular. Likewise, all productions of a left-regular language must be on the form

$$A \to Ba$$

The relationship between regular and context free languages is trivially obvious from these definitions. For example, the production

$$A \to aB$$

is regular, as stated above, and also context free because α can be any string. By saying that $\alpha = aB$, it is trivial to see that a regular language is a special case of a context free language. On the other hand, a production such as

$$A \to aBc$$

breaks the form of regular productions, meaning it is only context free.

The above example exhibits one of the properties that make context-free grammars important in the field of computer science. Context free grammars allow the language to match brackets or other symbols and strings denoting the start and end of a construct. Many modern languages are block-based where each block is started and ended with a reserved symbol or keyword, such as the numerous curly bracketed languages like C and Java, ALGOL with the keywords `BEGIN` and `END` and Lisp with parenthesis. A regular language would not be able to tell whether or not the input source code has the correct amount of starting- and closing symbols, while a context-free language will be able to do this based on the above observation. In order to accomplish this, one only has to substitute the terminals a and b for starting and closing block symbols. For example, in the case of parenthesis one might define the production as follows:

$$\begin{array}{ccc} A & \to & \text{'(' } A \text{ ')'} \\ A & \to & \epsilon \end{array}$$

where ϵ is the empty string. This language represents the language of all possible strings on the form $(^n)^n$ = (), (()), ((())) and so on. Notice how the amount of opening and closing parenthesis are always balanced. The closest possible representation one can achieve using a regular language with the kleene star is:

$$\begin{array}{ccc} A & \to & \text{'('* } B \\ B & \to & \text{')'*} \end{array}$$

This production represents the language of all strings on the form $(^n)^m$ which will accept the language above, where $n = m$ but also all the strings where $n \neq m$, which means the brackets are not balanced.

Context free grammars have their limitations as well. For instance, they cannot solve any problem that involves counting across productions. The common example to illustrate this is the language of all strings on the form $a^n b^n c^n$, meaning all strings with n a's followed by n b's and n c's such as *aabbcc* where $n = 2$ and ϵ where $n = 0$.

One could attempt to model the language with a starting condition as follows

$$S \rightarrow aAbBcC$$

but it won't take long until it becomes apparent that the context free languages simply do not support this kind of ordered counting problems. It is possible to create a context free language that contains n a's, b's and c's if the order is not considered. The following example shows this, using only the properties of a regular language.

$$S \rightarrow abcS$$

Solutions to this problem will be presented later in this chapter, as other languages have mechanisms to solve this problem. Despite this, the original definitions of the formats to represent music proposed in Calzo-Zaragoza et al [10] was defined using a context free grammar. This serves as a testament to its expressive capabilities in the context of representing musical notation in addition to defining the syntax of programming languages.

5.3.3 Context-Sensitive Grammars

Context Sensitive Grammars (CSG) are the next step out in the Chomsky hierarchy. The two innermost types of languages in the Chomsky hierarchy were described in Section 5.3.2, and can be seen as special cases of the context sensitive grammar. Sharing the same origin from Noam Chomsky, the context sensitive grammars represent a more expressive class of language than its previously mentioned siblings.

Despite it being more expressive, the expressiveness is considered to be mostly theoretical. The difficulty of writing a context sensitive grammar alone has prevented it from seeing much practical use. To make matters worse, the process of writing a parser for such a grammar is more difficult. While it can be proven that it is theoretically possible to construct a parser for any given grammar [25], the required book-keeping and high complexity would make the procedure infeasible for practical use. Little progress on context sensitive parsing has been made the last years, because it is not a field of active research due to its reputation to be human unfriendly.

Choosing to ignore its reputation and complexity, it is still interesting to see what the context sensitivity can theoretically bring to the table. By using the class of context sensitive, type-1, languages to illustrate the potential solutions it may offer, later sections may use this as a baseline to show how they may do it even better or easier.

A context sensitive grammar rule is on the form

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

where α and β are possibly empty strings of both terminals and non-terminals, A is a non-terminal and γ is a non-empty string of non-terminals and terminals. Note how the context of A, namely α and β, is not altered in any way. Only one of the non-terminals on the left hand side can be replaced in the right hand side.

By setting α and β to the empty string, ϵ, the relation to context free grammars become apparent. Informally, a context sensitive grammar that does not care about its context is context free.

Returning to the example of a language containing all strings $a^n b^n c^n$, it can be shown that it is representable using a context sensitive language. In fact, there are many possible context sensitive grammars that represent this language. The following is taken from Grune and Jacobs' book Parsing Techniques [25], and is chosen due to its brevity. Rules are enumerated in parenthesis.

$$
\begin{array}{cccc}
(0) & S & \rightarrow & abc \\
(1) & S & \rightarrow & aSQ \\
(2) & bQc & \rightarrow & bbcc \\
(3) & cQ & \rightarrow & Qc
\end{array}
$$

Which, from the following derivations, may construct the strings on the desired form, here with $n = 3$. The number of the rule that is used in each step is displayed in the parenthesis.

$$
\begin{array}{cccc}
S & \rightarrow & aSQ & (1) \\
aSQ & \rightarrow & aaSQQ & (1) \\
aaSQQ & \rightarrow & aaabcQQ & (0) \\
aaabcQQ & \rightarrow & aaabQcQ & (3) \\
aaabQcQ & \rightarrow & aaabbccQ & (2) \\
aaabbccQ & \rightarrow & aaabbcQc & (3) \\
aaabbcQc & \rightarrow & aaabbQcc & (3) \\
aaabbQcc & \rightarrow & aaabbbccc & (2)
\end{array}
$$

The awareness of context may allow the example from Section 5.3.1 to be implemented. If one could match α with something that uniquely identifies the time signature, it would be possible to create a context sensitive rule to create rhythmically sound bars.

$$\text{timesignature } \tfrac{3}{4} \text{ A } \beta \rightarrow \text{timesignature } \tfrac{3}{4} \text{ beat beat beat barline } \beta$$

However, this would mean manually creating production rules for each time signature as well as every subdivision of a beat. A context sensitive grammar cannot generate rules by itself, so every type of `timesignature` would have to have its own symbol for it to be used. This means that the grammar developer would have to both write rules for all supported time signatures as well as all supported figures, such as the quarter and eighth notes:

$$
\begin{aligned}
\text{timesignature } \tfrac{1}{4} \text{ A } \beta &\rightarrow \text{timesignature } \tfrac{1}{4} \text{ beat barline } \beta \\
\text{timesignature } \tfrac{2}{4} \text{ A } \beta &\rightarrow \text{timesignature } \tfrac{2}{4} \text{ beat beat barline } \beta \\
\text{timesignature } \tfrac{3}{4} \text{ A } \beta &\rightarrow \text{timesignature } \tfrac{3}{4} \text{ beat beat beat barline } \beta \\
&\ldots \\
\text{beat} &\rightarrow \text{halfbeat halfbeat} \\
\text{halfbeat} &\rightarrow \text{quarterbeat quarterbeat} \\
\text{quarterbeat} &\rightarrow \text{eighthbeat eighthbeat} \\
&\ldots
\end{aligned}
$$

This is a major limitation on top of the already complex parser for a context sensitive grammar. In our context of OMR, there should be automatic support for all time signatures and figures. One may not make assumptions about what time signatures and figures are used, unless the solution comes with a predefined set of supported variations.

In summary, the context sensitive grammar allows for more expressive power than the context free grammars but at a significant cost. Both the process of constructing the parser, as well as grammar development has significantly more complexity tied to it. In this section, a scenario was identified in which the developer had to manually add many symbols in order for the grammar to do what is desired. This will lead to a verbose grammar that is hard to read and potentially harder to maintain.

5.3.4 Van Wijngaard Grammars

The first type of formal grammar that is not classified within the Chomsky hierarchy is the Van Wijngaard Grammar (VWG). The VWGs are not a common sight, but are infamous for being able to represent just about anything [18]. They were invented by Aadrian van Wijngaarden during the development of ALGOL68 [56, 57] for a specific purpose. Context free grammars for a programming languages syntax were at the time the standard way of defining the syntax, but programming languages tend to have a few context sensitive aspects. This meant that the syntax of a language was completely defined by the use of a context free grammar and a description of the context sensitive restrictions written in a natural language. For example, in most languages, an identifier has to be defined before use. A context free grammar may produce the correct statements for the creation and use of identifiers, but cannot check whether or not the identifier in question has previously been defined and is accessible in the current scope. The VWG aims to solve this.

The Van Wijngaard Grammar features a two-level definition. This means that there are in fact two grammars working together to define the language. The first two-level grammar is attributed to L. Meertens and K. Koster in their paper “Basic English, a Generative Grammar for a Part of English” [34], which was thought to be lost but fortunately was found. Dick Grune provided an English translation [24] as the original paper is written in Dutch. The VWGs employ a similar approach, in which the first context free grammar derives rules based on the second context free grammar. This allows the derivation of infinitely many rules with a finite set of productions, which was identified as a useful feature in the previous Section 5.3.3 on Context sensitive grammars. The terminology and notation used by Guy de Chastellier in his paper about VWGs [17] will also be used here, despite that the definitions differ from the original papers by Aadrian van Wijngaarden. This is due to personal preference, as Chastelliers terminology and notation was found to

be more clear.

A VWG is defined by two vocabularies, two grammars and an axiom. The vocabularies are the *variables*, V_i, and the *values*, V_u. The set of *basic strings* are a subset of V_u that works similarly to terminals in other grammars. The two grammars are called the *meta-rules* and the *pseudo-rules*. A production from the pseudo-rules is denoted by $B \mapsto C$. The pseudo-rules are analogous to the production rules of a normal grammar, they rewrite *values* to new strings of *values* and basic strings. However, the pseudo-rules can also include a *variable*, that is an element of V_i. The pseudo-rules do no specify how to rewrite *variable*s, and this is where the meta-rules come in. A production from the meta-rules is indicated by $A \rightarrow B$, and rewrites the *variables* in the pseudo-rules. The result of a pseudo-rule that has been rewritten by the meta-rules are called simply a *rule*, and is denoted by $A \Rightarrow C$. The terminating condition of the meta-rules are when there are no more *variables*, and the terminating condition for the pseudo rules are when there are no more *values*, thus leaving only a string consisting of basic strings.

To both illustrate how this works, as well as the context sensitivity this brings, the example of $A^nB^nC^n$ will be represented as a VWG. The following grammar is taken from Guy de Chastelliers paper [17]. The axiom is N, the variables, V_i, are N and L, the values, V_u, are a, b, c, u and the basic strings are a, b, c. The meta rules of the system are:

$$\begin{array}{cccc} (0) & N & \rightarrow & u \\ (1) & N & \rightarrow & Nu \\ (2) & L & \rightarrow & a \\ (3) & L & \rightarrow & b \\ (4) & L & \rightarrow & c \end{array}$$

The pseudo rules for the system are:

$$\begin{array}{cccc} (5) & N & \mapsto & Na, Nb, Nc \\ (6) & NuL & \mapsto & NL, L \\ (7) & uL & \mapsto & L \end{array}$$

In order to derive *aaabbbccc*, or $a^3b^3c^3$, the axiom is rewritten using the meta-rules. By rewriting the axiom, using the meta rules, *axiomatic strings* are obtained. It is trivially shown that this can construct all axiomatic strings defining the language u^n:

$$\begin{array}{cccc} N & \rightarrow & Nu & (1) \\ Nu & \rightarrow & Nuu & (1) \\ Nuu & \rightarrow & uuu & (0) \end{array}$$

Observe how the result contains no *variables*, and thus the meta-rules are finished processing the axiom. Next, the pseudo-rules are rewritten using the meta-rules. By applying the meta-rules (rules 0-4) repeatedly on the pseudo-rules (rules 5-7), an infinite set of rules may be derived. The following derivations supply the required rules:

$$
\begin{array}{cccc}
(0) & u & \Rightarrow & uaubuc \\
(1) & uu & \Rightarrow & uuauubuuc \\
 & ... & & \\
(2) & ua & \Rightarrow & a \\
(3) & ub & \Rightarrow & b \\
(4) & uc & \Rightarrow & c \\
 & ... & & \\
(5) & uua & \Rightarrow & uaa \\
(6) & uub & \Rightarrow & ubb \\
(7) & uuc & \Rightarrow & ucc
\end{array}
$$

These rules, given an axiomatic string, u^n that is derived, as shown above, will yield the language of $a^n b^n c^n$. For example, given the axiomatic string uu, which is u^2, the following can be derived:

$$
\begin{array}{cccc}
uu & \Rightarrow & uuauubuuc & (1) \\
uuauubuuc & \Rightarrow & uaaubbucc & (5,6,7) \\
uaaubbucc & \Rightarrow & aabbcc & (2,3,4)
\end{array}
$$

The two level grammar structure of a VWG is extremely expressive. The two level grammar allows an infinite amount of production rules to be generated from a small set of rules. This removes the problem of manual labour that was present in the CSG. However, this expressiveness comes at a cost. It is an undecidable problem to decide whether or not a string is part of a language defined by a VWG [18]. In the general, unrestricted case a VWG can describe every recursively enumerable (type-0) language. Some restrictions and normal forms have been proposed, but the construction of a parser for a VWG is still too complex to be considered practical or even feasible.

5.3.5 Definite Clause Grammars

In a whole other paradigm is the Definite Clause Grammar, or DCG for short. They are considered a to be in a different paradigm because a DCG represents the parsing problem as a deduction problem in first order logic. It is closely tied to logic programming and Prolog in particular, which was partially designed for language processing. Other types of formal languages do not have a particular programming language or paradigm so closely associated with them.

First, some basic terminology introduced in the paper "Parsing as deduction" [44] is presented. A definite clause is a type of logic statement on the form

$$P \Leftarrow Q_1 \ \& \ \cdots \ \& \ Q_n$$

where P and $Q_1 \ldots Q_n$ are *literals*. P is labeled the *head* of the clause and $Q_1 \ldots Q_n$ is the *body* of the clause. Every literal may also take an argument. It is read as "P is true if all Qs are true". In the case of $n = 0$, the clause is reduced to only P is called a unit clause. A unit clause represents something true. This sums up the two kinds of statements in definite clause logic, namely the unit clause that is true and the clause that is true if the body of the clause is true.

Next, the relation between logic clauses and production rules in a grammar is key [43]. By translating each production rule into a definite clause, it is possible to represent a parsing problem as a first order predicate logic problem. When all the rules are definite clauses, the problem of parsing a string becomes the problem of proving a set of propositions. There are additional implementational details such as algorithms for automated proofs and syntactic sugar, but these are not considered relevant for the context of this thesis.

The arguments that each literal may take drastically increases the expressive powers of a DCG compared to a CFG. Additional statements may be added in curly brackets after the production, which allows more conditions that must be satisfied or to keep track of state. It has even been stated that the VWG and the affix grammars are special cases of a DCG [43]. Just like the VWG, the DCG is capable of representing an infinite amount of context free rules with a finite amount of productions. Likewise, it is similar to the affix grammar in the sense that it adds the opportunity to add non-terminal productions with arguments.

DCGs are written in Prolog with a few unique syntax elements. The production operator, which is usually the equality sign or right arrow, is --> in a Prolog program defining a DCG. Terminals are put in square brackets and arguments are in parenthesis. Additional conditions and procedure calls are in curly brackets. Each rule is terminated by a dot, and may consist of several literals separated by commas.

Listing 5.15: DCG for the language of $a^n b^n c^n$ written in SWI-Prolog

```
1  s --> a(N), b(N), c(N).
2  a(0) --> [].
3  a(M) --> [a], a(N), {M is N + 1}.
4  b(0) --> [].
5  b(M) --> [b], b(N), {M is N + 1}.
6  c(0) --> [].
7  c(M) --> [c], c(N), {M is N + 1}.
```

Returning to the example of the language of $a^n b^n c^n$, it is easily produced using a DCG. Listing 5.15 displays a valid DCG that may be loaded up in an interactive Prolog environment. It will successfully parse all strings that are part of the language $a^n b^n c^n$. The key is that the identifier N denotes the same value in the first production rule, s, this means that each of the a, b and c productions use the same value for N. Each of the a, b and c productions then produce a single of the respective letters as well as itself, with the argument decremented by 1. The decrement is done in the curly brackets. This proves that the DCG formalism allows for context sensitivity and counting. Providing rules in a DCG that does not use its arguments are equivalent to using a CFG.

Because of its ease of use, a small extra experiment was implemented. Simply writing the rules yields a functional parser by loading it in a Prolog environment and using the `phrase/2` statement. The experiment is shown in Listing 5.16, and displays a DCG for a hypothetical chord structure. This chord structure is not used in the Reworked Grammar, as described in Section 5.1.3, but displays some other properties. In particular, it is stated how many notes, N there are in the chord, so it distinguishes 3-note chords (triads) from 4-note chords. Additionally, it generates a structure that expects exactly N notes. More

precisely, it generates the language that consists of

$$\epsilon$$
$$\text{chord 1 note}$$
$$\text{chord 2 note note}$$
$$\ldots$$
$$\text{chord N (note}^N\text{)}$$

While this is less general than the chord structure that is currently implemented in the MNC, it features a fail safe mechanism. If a chord identified with N notes and followed by the wrong amount of notes, the error will be detected. As of now, the chord is blind to the number of notes that are contained within it.

Listing 5.16: DCG for a chord structure written in SWI-Prolog

```
1  s --> chord(N) .
2  chord(0) --> [].
3  chord(N) --> [chord, N], note(N).
4  note(0) --> [].
5  note(M) --> [note], note(N), {M is N+1}.
```

Due to its expressive capabilities and ease of use, the DCG should be a prime candidate to extend the MNC. Unfortunately, because of the choice made in terms of technology, it will not be this easy. The MNC is implemented in C, while DCGs are used in logic programming languages like Prolog. Using the DCG would lead to massive reworks in software architecture and may even lead to a complete rewrite, despite the apparent advantages of the DCG.

5.4 Insights from Language Study

The previous section presented a number of formal languages. The ones that were included had some interesting new features that could be useful, often building upon the previously mentioned languages. The goal of the study was embedding the constraint checks described in Section 5.2 into the syntax of the language itself.

It was found that some degree of context sensitivity was required to achieve the goal of the study. What was labelled as *counting* was identified as an important property for expressing musical notation and illustrated using the example of the language $a^n b^n c^n$. The first type of language that could do this was the CSG. However, it was clear that it was impractical to use the CSG, due to the amount of manual labour in grammar development.

In order to find a more practical solution, the ability to generate an infinite amount of rules from a finite set of production rules was found in the VWG. This feature removed the need for creating hand made productions for each possible case. However, the VWG suffered from complexity issues. It is undecidable whether or not a string is part of a language defined by a VWG, and thus it is infeasible to construct an efficient parser.

A language that had both sufficient expressive capabilities and manageable complexity is the DCG. The DCG can be viewed as a powerful extension of a CFG, including both context sensitivity and the ability to create an infinite set of rules. By constructing a new

grammar for musical notation, using a DCG, it will be possible to embed the constraint checking phase into the grammar. However, this will not be done in this thesis.

Other languages were also studied. In particular, L-Systems [30, 31, 32, 52] seemed promising because of its extensive use in the field of algorithmic composition, as described in Section 3.2. However, the features of an L-System did not prove useful in this context.

Other types of languages that were studied but was omitted from Section 5.3 included the families of Affix Grammars [29, 58, 59], and Graph Grammars [19, 42, 46, 53].

Chapter 6

Results

This chapter presents the results obtained by running tests on the MNC using the PRiMuS dataset. The tests only differ in two ways: their grammar and whether or not constraint checking is enabled. The baseline results are obtained by running the tests using the Original Grammar from Calvo-Zaragoza et al [10], which accepts any sequence of musical symbols. The test called “Reworked” uses the Reworked Grammar, which was presented in Section 5.1.2. It features a strict musical structure which requires a valid heading at the start of each score, followed by a sequence of bars that are terminated by a bar line. The test called “Excerpt” uses a grammar that is derived from the Reworked Grammar later in this chapter.

The results labelled “Reworked + constraints” and “Excerpt + constraints” are obtained by running the same test as above with the respective grammars, as well as having the constraint checks enabled. Note that the tie checking will uncover any errors, as the PRiMuS dataset does not include the updated ties described in Section 5.2. Therefore, it is not enabled during testing on the PRiMuS dataset.

6.1 Experimental results

Grammar	Number of errors	Error percentage
Baseline	0	0 %
Reworked	50670	57.79 %
Reworked + constraints	51837	59.12 %
Excerpt	13	0.14 %
Excerpt + constraints	4610	5.25 %

Table 6.1: Overview of error rates for the different grammars. The total number of incipits in the dataset is 87678

Table 6.1 displays the total error rates of the different grammars and constraint checks.

Due to fact that the PRiMuS dataset is comprised of many musical incipits, or excerpts, many of the elements did not feature a complete musical score, so phenomena like missing the last bar line is a common slight. Because of this, the reworked grammar performed poorly. The numerous missing bar lines littered the error logs, and the grammar seemed to model the language poorly. Out of the 87678 elements in the data set, 50670 of them reported errors, leading to an error rate of 57.79%. Analysing the errors manually was not feasible.

To explore the impact of the missing bar lines, the Reworked Grammar was modified. The resulting *Excerpt grammar* enforces the same structure as the Reworked Grammar, except it does not require a valid heading at the start and does not require the very last bar line. This effectively makes the MNC tolerate a musical score that has been cropped, only showing the middle part. Running the test again revealed that the Excerpt Grammar did a good job at modelling the language, with a total number of errors counting to 13 which is a 0.14% error rate. With such a low number of syntax errors, it was possible to analyse them all manually. An overview of the errors found in this test is shown in Table 6.2.

Type of Error	Number
Clef and Keysignature in wrong order	5
... of which has a wrong key signature.	3
Misprint/corrupted	8

Table 6.2: Distribution of syntax errors from the experiments using the Excerpt grammar.

6.1.1 Syntax Errors

As Table 6.2 states, there are two main syntax errors that were uncovered. The first error, *Clef and Keysignature in wrong order*, appeared 5 times. This is an artefact of the structure imposed by the Reworked Grammar. The structure requires the updated key signature to come before the updated clef, in the event of these symbols both updating at the same time. In these 5 cases, the key signature is listed after the clef and is thus recognized as a syntax error. In 3 of the 5 cases, the key signature that causes a syntax error should not be there. By inspecting the image and comparing to the ground truth representation it was observed that there is a mismatch between the ground truth representation and the generated image. The image does not have a change of key signature, but the ground truth representation does.

The second error, *Misprint/corrupted*, denotes scores that seem to be malformed. Images that have symbols covering each other, bar lines that seem to be missing and other abnormalities are observed in all the eight cases. An example is shown in Figure 6.1.

6.1.2 Constraint Check Errors.

From Table 6.1 it is obvious that a higher rate of syntax errors prevent running the constraint check, as a valid AST is required to do any checking. While the Reworked Grammar

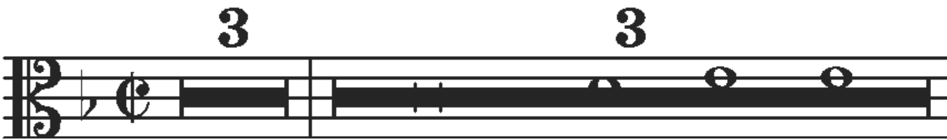


Figure 6.1: Incipit 201003440-1_1_1 is an example of a malformed score from the PRiMuS dataset.

uncovered 1167 errors during constraint checking, the Excerpt Grammar uncovered 4597. The difference between these two numbers are likely caused by incipits that would have failed the constraint checks, but did not pass the syntax analysis. Therefore, it is likely that the grammar with the fewest syntax errors reveals the most accurate number of constraint check errors.

Chapter 7

Conclusion

This chapter concludes the work done in this project. The results are discussed and some are highlighted. Finally, possible improvements and features are discussed.

7.1 Discussion

Assuming the PRiMuS dataset is an accurate representation of real life problems, the reworked grammar seems to be providing a fair syntactical model of the language, with only lesser modifications. The excerpt grammar only had two minor changes made to it and fixed the problem of musical incipits not including all the parts of a valid complete score. How significant the measured syntax error rate is, presented in Section 6.1, depends on how true the underlying assumption of PRiMuS being an accurate representation of real life problems. Calvo-Zaragoza et al [10] stated that the Original Grammar, and its alphabet, would only serve as a starting point for future research. Hence, the PRiMuS dataset only contains incipits that use this alphabet. With this in mind, it is hard to say whether or not a technique for validation, as presented in this thesis, would do well in a real life scenario. The only thing regarding the validity of such a technique is that it seemed to be effective in this scenario with a limited alphabet.

The constraint checks, however, did not lead to any conclusive insights. They might indeed prove useful in a real life scenario when there is no ground truth representation. Due to the limited tests, however, this is inconclusive. As the PRiMuS data represents ties in a different way than the reworked grammar presented in Section 5.1, the tie matching check is only tested by a few handcrafted sets of input. Checking whether or not each bar was properly filled resulted in many errors which will be looked into more closely in Section 7.1.3

Fortunately, the literature review presented in Chapter 3 backs up the observations found in this thesis: That it is possible to model the language of music as a grammar and use it to validate the correctness of a string representing music. This thesis only differs in the abstraction level, as most of the related work uses language theoretic approaches

during the low-level image segmentation and musical symbol construction phases. Here, a high-level grammar for music is used to evaluate a textual representation of a score.

The advantage of employing such a high level technique is modularity. The MNC, is a stand-alone program that can be used by new and existing OMR solutions with little to no coding effort, given they use the same language. Adding a validation step like the MNC as an intermediate step in the OMR pipeline described in Section 2.4.2 might save the OMR solution in question some work. Depending on what the OMR solution intends to do with the read data, time can be saved by detecting these errors early. By aborting early when an error is reported by the MNC, expensive operations that would later fail because of this error can be avoided. Examples of such operations are the synthesis of an audio file or the conversion to a musical notation file format for use with software like Sibelius[1]. These may take a significant amount of time to complete, in the order of seconds to minutes depending on input size. Additionally, while the conversion from an invalid string of music to a new format might be possible, it will surely end in end-user frustration when the exported file does not work with their music player or musical notation software.

7.1.1 Examining the Syntax Errors

As shown in Table 6.2, syntax errors were found. The first class of syntax error, the *Clef and Keysignature in wrong order* errors, are proof that the language model defined by the reworked grammar is not perfect. However, this syntax error only occured 5 times in the 87678 total incipits. One may argue that this error is insignificant due to its low rate of occurance. On the other hand, it might hint that the PRiMuS dataset does not represent a real life scenario. Looking at the grammar and at the strings that caused the errors revealed that the grammar can not handle the change of a clef at the very start of a bar. If the change of clef appears before the key signature symbol, then it is reported as an error by the parser. Changing the clef after the first symbol in a bar is permitted. This weakness is present in the current Reworked Grammar, and is an example of a mismatch between the intended language and the language defined by the Reworked Grammar. Changing the clef at the beginning of a bar is also not an uncommon sight, which makes it interesting to see the low number of errors like this.

Additionally, after manually reading through the syntax errors, an interesting phenomenon is observed. The ground truth representations of the incipits that reported *Clef and Keysignature in wrong order* errors some times had the wrong key signature. An incipit in the key of D minor would, according to the ground truth representation, change to C major after a clef was changed at the start of a bar. Comparing the ground truth representation to the image, there is no indication of changing key signature to C major. The interesting problem with C major (and A minor) is that the key signature should be blank. While generating the ground truth representations, the system might have mistaken a clef change followed by some blank space as both a clef change and a key change. If this is the case, some of these errors are false negatives. However, this is still an inaccuracy in the grammar because they are not all false negatives.

[1]https://www.avid.com/sibelius

7.1.2 Errors in the PRiMuS Data Set

The last category of syntax errors, the *Misprint/Corrupted* errors, were an interesting find. These errors were thoroughly inspected, and the respective incipits found to be invalid. In order words, it is not a weakness in the language model but rather errors in the data set. The image representation is corrupted and the ground truth representation does not reflect what is seen on the image.

However, all the errors involve the multirest symbol. It might be possible that the image generation did not handle multirests correctly, and that in some cases the only missing symbols are a few bar lines. Since it is now known where these musical notation incipits come from, it is impossible to find the original piece and find out. Whether or not it is a small bug in image generation or a bigger issue, it still invalidates these incipits. As the image is corrupted, it should not be used for either training or validation of systems. Using invalid images for training can lead to the system learning something that it should not. For example, a system may learn that multirests can overlap a bar with other symbols in it as seen in Figure 6.1, which is not allowed.

To the author's knowledge, this is not a known issue. No papers related to the PRiMuS data set mention it [10, 11]. Other sources, like this list of OMR-related data sets on github[2], has no mention of these errors either.

These findings prove, to some extent, the feasibility and effectiveness of a language based validation technique as proposed in this thesis. The fact that no one else has mentioned these errors might hint that the proposed technique of this thesis is either the only validation of the PRiMuS data set, or that previous attempts were flawed. For instance, using The Original Grammar to validate the data set revealed zero errors, as stated in Table 6.1. The complete list of erroneous incipits can be found in Appendix B

7.1.3 Examining the Constraint check errors

Like stated in Section 6.1.2 and Table 6.1, there are a lot of errors during constraint checking. Since the only constraint check that is compatible with the PRiMuS data set is whether or not all bars are properly filled, it is the only source of errors. The other constraint check is disabled during tests on the PRiMuS data set.

Unfortunately, the results do not create very clear patterns. Out of the 87678 total incipits, 4597 of them reported errors. This is an error rate of 5.24%. Of these 5.24%, there are, by manual inspection, not any clear trends. However, a few findings will be discussed.

Firstly, for the same reason the Excerpt grammar was created, the constraint checks ignore the first and last bar. This is simply due to the fact that the data set consists of excerpts or incipits, not complete scores. Because of this, the real number of constraint check errors might be even more inaccurate. The real number of constraint check errors would be possible to find given no syntax errors and no scores that are incomplete.

Secondly, the act of ignoring the first bar both solves the problem of an incomplete score as well as an anacrusis. This solution made the constraint check much easier to implement so that these results could be obtained in time. However, it is not a particularly

[2]https://github.com/apacha/OMR-Datasets

good solution in the context of the long term goal of improving the validation of OMR results. The anacrusis should be classified as something unique and treated separately. As the implementation is now, the MNC does not see the difference between a first bar and an anacrusis, no matter how corrupt they might be.

Finally, there is one trend in the constraint check errors that have been observed. The total summed up duration of a bar that contains errors tends to be consistent across all bars in the same incipit. This means that if the time signature requires 4 beats in every bar, it is common to see an erroneous incipit with bars that consistently add up to, for example, 3 beats. Of course, some incipits have inconsistent durations as well, and could contain just a single bar missing a note. However, this seems to be a consistent finding when looking though the error logs. The consistent duration across bars also tend to be either double the intended duration or half the intended duration. This raises the interesting question of whether this is due to an error during image generation or if the original music looks like this. Since the data set only contains real music, taken from a bigger archive of musical notation incipits, it seems unlikely that errors like this occur as frequently as 5.24%. The case of a time signature requiring 4 beats and every bar consistently summing up to 3 beats seems like an error in the data set. However, the case of double or half the intended duration may be intentional as especially pre-baroque music is known to make exceptions to these rules [21].

Because it is not stated which incipit from the PRiMuS dataset is from which piece of music, the above questions are hard to answer. If one could check the original musical notation for each incipit that reported errors, then it would be possible to state whether or not these phenomena are errors that may have occurred during generation. As of now, it is impossible to determine whether the constraint check errors found in this thesis are intentional or not.

7.2 Conclusions

In this thesis, a design experiment concerning the use of a model based approach to validate the results of an OMR solution was conducted. A DL-OMR solution was chosen and a language model for its result domain was developed in the form of a compiler front-end and tested on an OMR data set.

For DL-OMR purposes, the proposed solution cannot help during the training phase and its results can only partially be validated. When a DL-OMR solution is deemed “finished” and starts to see real life use, it has no longer access to ground truth representations as it will encounter unique real world problems. A language based model may catch many potential errors an OMR system may encounter, by syntax and semantic analysis. It will, however, not catch classification errors like a misread note pitch. For example, this method will not report an error if a pitch of F is read instead of a G.

Additionally, a study of formal languages was conducted in the context of this validation technique. By identifying needs and evaluating different kinds of formal languages in the context of these needs, some useful insights were obtained. Since musical notation is context sensitive, much like programming languages, some additional constraint checks were needed to do when using a context free grammar.

It was found that a DCG would allow these constraint checks to be embedded in its

grammar, thus removing the need for any constraint checking code. By doing this, the complete definition of the language model is contained within the grammar. This has the advantage of separating the implementation from the model. By allowing the complete language definition to be developed by only editing the grammar, it is no longer important how it is implemented in code. As of now, the constraint checking code is a part of the language model definition as the DCG grammar was not developed.

Testing the Reworked Grammar on an OMR dataset, PRiMuS, showed good results with a 0.14% error rate during syntax analysis, some of which were in fact errors in the dataset – not the grammar. The fact that these errors were not mentioned in any literature, and is thought to be undiscovered until now, could be regarded as proof that this approach can be rather effective. The remaining errors were due to inaccuracies in the language model.

The language model validation approach is not limited to DL-OMR solutions. All OMR solutions that produce output suitable for parsing, like strings instead of annotated images, will be able to employ a similar strategy to the one presented in this thesis. In fact, it may be a useful feature in many other fields of research than OMR. Some previous work has already used language based models with success, and this thesis hopes to bring these models back to combine with the deep learning approaches to achieve even better results than the current state of the art.

7.3 Future Work

Disregarding many of the shortcomings of the prototype compiler, MNC, there is much potential for future work. While there were discovered inaccuracies in the reworked grammar, these should not be a priority. This work is intended to answer whether or not this technique is feasible and the MNC is not intended to be a complete solution.

There are in particular two points of future work that could be of interest. Firstly, since unmentioned errors in the PRiMuS data set was found, it would be useful to do additional analyses to potentially detect even more errors. By detecting errors, it is possible to clean up the PRiMuS data set by either removing the erroneous incipits or repairing them. This would be valuable, as the data set is used to train OMR solutions. Training on bad data will lead to poor performance and bad models. The effects may be small since there were such a low number of errors. Either way, these are now known errors which should be fixed for correctness.

Finally, this thesis showed that formal languages can be used to create a model-based validation method for OMR solutions. This implies that there might be untapped potential in other fields of research. For example, computationally reading handwritten math notation would be an interesting case to explore. Although it may have limited use cases, it is an interesting thought as math expressions are structured in a way that fits formal languages. There exist many formal grammars for arbitrary math expressions, which could be implemented on top of a handwritten text recognition solution to check for correctness. With a bit of ambition, this can ultimately end up in automatic grading of handwritten math assignments.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers, principles, techniques. *Addison Wesley*, 7(8):9, 1986.

[2] M. Alfaro-Contreras, J. Calvo-Zaragoza, and J. M. Iñesta. Approaching end-to-end optical music recognition for homophonic scores. In A. Morales, J. Fierrez, J. S. Sánchez, and B. Ribeiro, editors, *Pattern Recognition and Image Analysis*, pages 147–158, Cham, 2019. Springer International Publishing. ISBN 978-3-030-31321-0.

[3] R. Antonsen. *Logiske metoder: kunsten å tenke abstrakt og matematisk*. Universitetsforlaget, 2014.

[4] J. W. Backus. The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. *Proceedings of the International Comference on Information Processing, 1959*, 1959.

[5] D. Bainbridge. *Extensible Optical Music Recognition*. PhD thesis, University of Canterbury, 1997.

[6] D. Bainbridge and T. Bell. The challenge of optical music recognition. *Computers and the Humanities*, 35:95–121, 05 2001. doi: 10.1023/A:1002485918032.

[7] D. Bainbridge and T. Bell. A music notation construction engine for optical music recognition. *Software: Practice and Experience*, 33(2):173–200, 2003.

[8] B. Bel and J. Kippen. Bol processor grammars, 1992.

[9] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, et al. Extensible markup language (xml) 1.0, 2000.

[10] J. Calvo-Zaragoza and D. Rizo. End-to-end neural optical music recognition of monophonic scores. *Applied Sciences*, 8(4), 2018. ISSN 2076-3417. doi: 10.3390/app8040606. URL `http://www.mdpi.com/2076-3417/8/4/606`.

[11] J. Calvo-Zaragoza and D. Rizo. Camera-primus: Neural end-to-end optical music recognition on realistic monophonic scores. In *ISMIR*, pages 248–255, 2018.

[12] J. Calvo-Zaragoza, J. H. Jr., and A. Pacha. Understanding optical music recognition, 2019.

[13] F. J. Castellanos, J. Calvo-Zaragoza, G. Vigliensoni, and I. Fujinaga. Document analysis of music score images with selectional auto-encoders. In *ISMIR*, pages 256–263, 2018.

[14] N. Chomsky. Systems of syntactic analysis. *The Journal of Symbolic Logic*, 18(3): 242–256, 1953.

[15] N. Chomsky. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124, 1956.

[16] B. Coüasnon and B. Rétif. Using a grammar for a reliable full score recognition system. In *ICMC*. Citeseer, 1995.

[17] G. de Chastellier and A. Colmerauer. W-grammar. In *Proceedings of the 1969 24th national conference*, pages 511–518, 1969.

[18] P. Deussen. A decidability criterion for van wijngaarden grammars. *Acta Inf.*, 5(4): 353–375, Dec. 1975. ISSN 0001-5903. doi: 10.1007/BF00264566. URL `https://doi.org/10.1007/BF00264566`.

[19] H. Ehrig, A. Habel, and H.-J. Kreowski. Introduction to graph grammars with applications to semantic networks. *Computers & Mathematics with Applications*, 23 (6):557 – 572, 1992. ISSN 0898-1221. doi: https://doi.org/10.1016/0898-1221(92)90124-Z. URL `http://www.sciencedirect.com/science/article/pii/089812219290124Z`.

[20] Flex. The fast lexical analyzer. `https://github.com/westes/flex`. Accessed: 2020-27-05.

[21] I. Fujinaga. *Optical music recognition using projections*. PhD thesis, McGill University Montreal, Canada, 1988.

[22] GNU Bison. Gnu project, free software foundation. `https://www.gnu.org/software/bison/`. Accessed: 2020-27-05.

[23] R. Göcke. Building a system for writer identification on handwritten music scores. In *Proceedings of the IASTED International Conference on Signal Processing, Pattern Recognition, and Applications (SPPRA)*, pages 250–255, 2003.

[24] D. Grune. translation of "basic english, a generative grammar for a part of english". `https://dickgrune.com/Books/PTAPG_2nd_Edition/LowAvailability/PSG_Translation.pdf`. Accessed: 2020-27-03.

[25] D. Grune and C. J. Jacobs. Parsing techniques. *Monographs in Computer Science. Springer,*, page 13, 2007.

[26] Haskell. Haskell: An advanced, purely functional programming language. `https://www.haskell.org/`. Accessed: 2020-27-05.

[27] Haskell grammar specification. Syntax reference. `https://www.haskell.org/onlinereport/haskell2010/haskellch10.html`. Accessed: 2020-21-04.

[28] K. C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, Oct. 1965. ISSN 0001-0782. doi: 10.1145/365628.365655. URL `https://doi.org/10.1145/365628.365655`.

[29] C. Koster. *Affix Grammars for Natural Languages*, pages 469–484. Springer Berlin Heidelberg, 1991. doi: 10.1007/3-540-54572-7_19.

[30] A. Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280 – 299, 1968. ISSN 0022-5193. doi: https://doi.org/10.1016/0022-5193(68)90079-9. URL `http://www.sciencedirect.com/science/article/pii/0022519368900799`.

[31] A. Lindenmayer. Mathematical models for cellular interactions in development ii. simple and branching filaments with two-sided inputs. *Journal of Theoretical Biology*, 18(3):300 – 315, 1968. ISSN 0022-5193. doi: https://doi.org/10.1016/0022-5193(68)90080-5. URL `http://www.sciencedirect.com/science/article/pii/0022519368900805`.

[32] S. Manousakis. Musical l-systems. *Koninklijk Conservatorium, The Hague (master thesis)*, 2006.

[33] J. McCormack. Grammar based music composition. *Complex systems*, 96:321–336, 1996.

[34] L. Meertens and C. Koster. Basic english, a generative grammar for a part of english. In *Euratom Seminar\Machine en Talen", Amsterdam*, 1962.

[35] F. M. Müller. *Lectures on the Science of Language*, volume 1. Longman, Green, 1873.

[36] A. Pacha and H. Eidenberger. Towards self-learning optical music recognition. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 795–800, 2017.

[37] A. Pacha, K. Choi, B. Coüasnon, Y. Ricquebourg, R. Zanibbi, and H. Eidenberger. Handwritten music object detection: Open issues and baseline results. In *2018 13th IAPR International Workshop on Document Analysis Systems (DAS)*, pages 163–168, 2018.

[38] A. Pacha, J. Hajič, and J. Calvo-Zaragoza. A baseline for general music object detection with deep learning. *Applied Sciences*, 8(9):1488, 2018.

[39] A. Pacha, J. Calvo-Zaragoza, and J. Hajic jr. Learning notation graph construction for full-pipeline optical music recognition. In *20th International Society for Music Information Retrieval Conference*, 2019.

[40] G. Papadopoulos and G. Wiggins. Ai methods for algorithmic composition: A survey, a critical view and future prospects. In *AISB Symposium on Musical Creativity*, volume 124, pages 110–117. Edinburgh, UK, 1999.

[41] Parsec. Parsec: Monadic parser combinators. `https://hackage.haskell.org/package/parsec`. Accessed: 2020-27-05.

[42] T. Pavlidis. Linear and context-free graph grammars. *J. ACM*, 19(1):11–22, Jan. 1972. ISSN 0004-5411. doi: 10.1145/321679.321682. URL `https://doi.org/10.1145/321679.321682`.

[43] F. C. Pereira and D. H. Warren. Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13(3):231 – 278, 1980. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(80)90003-X. URL `http://www.sciencedirect.com/science/article/pii/000437028090003X`.

[44] F. C. N. Pereira and D. H. D. Warren. Parsing as deduction. In *Proceedings of the 21st Annual Meeting on Association for Computational Linguistics*, ACL '83, page 137–144, USA, 1983. Association for Computational Linguistics. doi: 10.3115/981311.981338. URL `https://doi.org/10.3115/981311.981338`.

[45] J. F. Power. Thue's 1914 paper: a translation. *arXiv preprint arXiv:1308.5858*, 2013.

[46] T. W. Pratt. Pair grammars, graph languages and string-to-graph translations. *Journal of Computer and System Sciences*, 5(6):560–595, 1971.

[47] P. Prusinkiewicz. Score generation with l-systems. In *ICMC*, 1986.

[48] Python grammar specification. Full grammar specification. `https://docs.python.org/3/reference/grammar.html`. Accessed: 2020-21-04.

[49] D. Quick and P. Hudak. Grammar-based automated music composition in haskell. In *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design*, FARM '13, page 59–70, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323864. doi: 10.1145/2505341.2505345. URL `https://doi.org/10.1145/2505341.2505345`.

[50] A. Rebelo, I. Fujinaga, F. Paszkiewicz, A. R. Marcal, C. Guedes, and J. S. Cardoso. Optical music recognition: state-of-the-art and open issues. *International Journal of Multimedia Information Retrieval*, 1(3):173–190, 2012.

[51] P. Roland. The music encoding initiative (mei). In *Proceedings of the First International Conference on Musical Applications Using XML*, volume 1060, pages 55–59, 2002.

[52] G. Rozenberg and A. Salomaa. *Handbook of Formal Languages Volume 1 Word, Language, Grammar*. Springer-Verlag Berlin Heidelberg, 1997.

[53] G. Rozenberg and A. Salomaa. *Handbook of Formal Languages: Volume 3 Beyond Words*. Springer Science & Business Media, 2012.

[54] A. Thue. *Selected mathematical papers of Axel Thue*. Universitetsforlaget, 1977.

[55] P. Van-Roy, S. Haridi, et al. *Concepts, techniques, and models of computer programming*. MIT press, 2004.

[56] A. van Wijngaarden. Orthogonal design and description of a formal language. *Stichting Mathematisch Centrum. Rekenafdeling*, 1965.

[57] A. Van Wijngaarden, B. Mailloux, J. Peck, C. Koster, M. Sintzoff, C. Lindsey, L. Meertens, and R. Fisker. Report on the algorithmic language algol 68. *Numerische Mathematik*, 14(1):79–218, 1969.

[58] D. A. Watt. *Analysis-oriented two-level grammars*. PhD thesis, University of Glasgow, 1974.

[59] D. A. Watt. The parsing problem for affix grammars. *Acta Informatica*, 8(1):1–20, 1977.

Appendices

Appendix A

Complete Grammars

A.1 Original Context Free Grammar for Semantic Encoding in EBNF

Calvo-Zaragoza et al [10] used slightly non-standard EBNF notation. When an excerpt of this grammar was presented in Section 5.1, it was rewritten in EBNF. Here, it is presented in its original form. An optional symbol is not enclosed in square brackets, but has a question mark appended. A sequence of zero-or-more symbols are not enclosed in curly brackets, but has an asterisk appended. Several symbols may be enclosed in parenthesis before appending the question mark or asterisk. A sequence of one-or-more symbols has a plus sign appended.

The Original Context Free Grammar

```
sequence  =  (symbol sep symbol)*
  symbol  =  clef sepsymbol clefnote linenumber
          |  timesignature sepsymbol (metersigns | (integer slash integer))
          |  keysignature sepsymbol diatonic accidentals? (major | minor)?
          |  (note | gracenote) sepsymbol pitch sepvalues
                 figure dots? (sepvalues fermata)? (sepvalues trill)?
          |  tie
          |  barline
          |  rest sepsymbol figure dots? (sepvalues fermata)?
          |  multirest sepsymbol integer
   pitch  =  diatonic accidentals? octave
  octave  =  digit
    dots  =  dot+
```

The Original Lexical Rules

This is exactly as written in Calvo-Zaragoza et al [10], with the same inconsistent use of quotation marks. `thickbarline` is not used in the grammar, and was not included in the Reworked Grammar made in this thesis. The figure `two_hundred_fifty_six`, representing a two-hundred-and-fifty-sixth note, and the `trill` symbols are not present in the PRiMuS dataset.

digit = (‘0’ .. ‘9’)

integer = (‘0’ .. ‘9’)+

slash = ‘.’

clefnote = {‘C’ | ‘G’ | ‘F’}

linenumber = { ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’}

accidentals = { ‘bb’ | ‘b’ | ‘n’ | ‘#’ | ‘x’}

metersigns = “C” | “C/”

trill = “trill”

fermata = “fermata”

clef = “clef”

note = “note”

gracenote = “gracenote”

rest = “rest”

multirest = “multirest”

barline = “barline”

thickbarline = “thickbarline”

figure = { “quadruple_whole”

| “double_whole”

| “whole”

| “half”

| “quarter”

| “eighth”

| “sixteenth”

| “thirty_second”

| “sixty_fourth”

| “hundred_twenty_eight”

| “two_hundred_fifty_six” }

dot = “.”

tie = “tie”

diatonicpitch = { “A” | “B” | “C” | “D” | “E” | “F” | “G”}

keysignature = “keySignature”

timesignature = “timeSignature”

minor = “m”

major = “M”

sep = TAB

sepsymbol = “-”

sepvalues = “_”

A.2 The Complete Reworked Grammar

This is generated by Bison by compiling with the verbose setting on. This generates a file, `y.output`, which contains any errors, conflicts or warnings as well as the complete grammar, list of non-terminals and terminals as well as transition tables in a human readable format.

$accept is a symbol that bison creates which represents an accepting state. If the production rule of $accept is derived, it is valid. $end denotes end of input. %empty denotes an empty production body, which is used to represent optional production rules.

```
     $accept  :  score $end
       score  :  clefcom keysig timesig bars
        bars  :  bar
              |  bar bars
         bar  :  opt_keysig opt_timesig symbol_list BARLINE_TOKEN
              |  opt_keysig opt_timesig multirest BARLINE_TOKEN
    last_bar  :  opt_keysig opt_timesig symbol_list
              |  opt_keysig opt_timesig symbol_list BARLINE_TOKEN
              |  opt_keysig opt_timesig multirest BARLINE_TOKEN
              |  opt_keysig opt_timesig multirest
   multirest  :  MULTIREST_TOKEN SEP_SYM_TOKEN integer
 symbol_list  :  opt_clef symbol
              |  opt_clef symbol symbol_list
    opt_clef  :  %empty
              |  CLEF_TOKEN SEP_SYM_TOKEN diatonic integer
      symbol  :  note
              |  rest
              |  chord
              |  tuplet
      tuplet  :  TUPLET_TOKEN SEP_VALUES_TOKEN
                     integer symbol_list END_TOKEN
       chord  :  CHORD_TOKEN SEP_SYM_TOKEN length pitch_list ties
        rest  :  REST_TOKEN SEP_SYM_TOKEN length
              |  REST_TOKEN SEP_SYM_TOKEN length extensions
        note  :  note_base extensions
              |  note_base
   note_base  :  NOTE_TOKEN SEP_SYM_TOKEN pitch
                     SEP_VALUES_TOKEN length
              |  GRACE_NOTE_TOKEN SEP_SYM_TOKEN pitch
                     SEP_VALUES_TOKEN length
```

```
extensions   : SEP_VALUES_TOKEN ties SEP_VALUES_TOKEN fermata
             | SEP_VALUES_TOKEN ties
             | SEP_VALUES_TOKEN fermata
pitch_list   : SEP_SYM_TOKEN pitch pitch_list
             | SEP_SYM_TOKEN pitch
pitch        : diatonic accidentals integer
figure       : FIGURE_TOKEN
length       : figure dots
dots         : %empty
             | DOT_TOKEN dots
fermata      : FERMATA_TOKEN
ties         : TIESTART_TOKEN
             | TIEEND_TOKEN
             | TIEBOTH_TOKEN
diatonic     : DIATONIC_TOKEN
clefcom      : CLEF_TOKEN SEP_SYM_TOKEN diatonic integer
minor        : MINOR_TOKEN
major        : MAJOR_TOKEN
keysig       : KEY_TOKEN SEP_SYM_TOKEN diatonic accidentals major
             | KEY_TOKEN SEP_SYM_TOKEN diatonic accidentals minor
accidentals  : ACCIDENTAL_TOKEN
             | %empty
meter        : METER_TOKEN
timesig      : TIME_TOKEN SEP_SYM_TOKEN meter
             | TIME_TOKEN SEP_SYM_TOKEN integer SLASH_TOKEN integer
opt_timesig  : timesig
             | %empty
opt_keysig   : keysig
             | %empty
integer      : INTEGER_TOKEN
```

Appendix B

Complete List of Misprinted Musical Incipits Found in the PRiMuS Data Set

All the misprinted and/or corrupted musical incipits are listed here. These are the 8 errors categorized as *Misprint/corrupted* which is described in Section 6.1.1.

The errors caused by a non-existing key signature change are not listed in this appendix, as it is not informative to look at the musical notation to understand these errors.

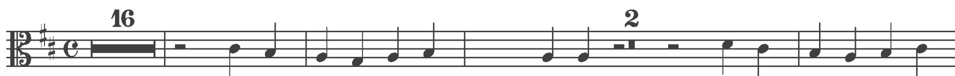


Figure B.1: Incipit 000102052-1_2_2 from package_aa. The 2 bar multirest is misplaced and there should be some bar lines.

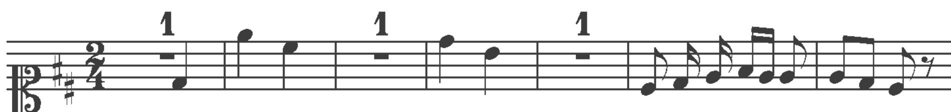


Figure B.2: Incipit 000108292-1_1_2 from package_aa. The first 1 bar multirest is misplaced and misses barlines. Does not make sense in the big picture, as there is a lone quarter note after it.

Figure B.3: Incipit 000130315-1_1_1 from package_aa. Last bar and multirest is clashed together. Transcription states it is a 1111 bar multirest.

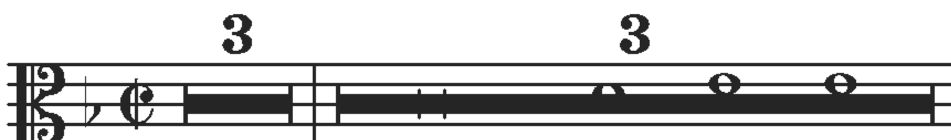


Figure B.4: Incipit 201003440-1_1_1 from package_ab. The last bar and multirest is clashed together.

Figure B.5: Incipit 201003441-1_1_1 from package_ab

Figure B.6: Incipit 201003475-1_1_1 from package_ab

Figure B.7: Incipit 210097416-1_5_2 from package_ab

Figure B.8: Incipit 212001010-1_4_1 from package_ab

Joakim Omberg Lier

Applying Language Technology in Error Detection for Optical Music Recognition

NTNU
Norwegian University of Science and Technology