Informatics

Komposition polyphoner Musik mit Grammatiken

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Lukas Eibensteiner, BSc
Matrikelnummer 01225627

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Mag. Martin Ilčík

Wien, 15. April 2021

Lukas Eibensteiner

Michael Wimmer

Informatics

Polyphonic Music Composition with Grammars

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Lukas Eibensteiner, BSc
Registration Number 01225627

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Mag. Martin Ilčík

Vienna, 15th April, 2021

Lukas Eibensteiner

Michael Wimmer

Erklärung zur Verfassung der Arbeit

Lukas Eibensteiner, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 15. April 2021

Lukas Eibensteiner

Danksagung

Mein tiefster Dank gilt Martin, der bereits mehrere Jahre mit mir zusammenarbeitet und seine Zeit, seine Erfahrung und sein Wissen mit mir teilt, und meinen Eltern, Gerda und Friedrich, die mich immer bedingungslos unterstützen.

Acknowledgements

I thank Martin, who worked with me for years and shared his time, his experience, and his knowledge with me, and my parents, Gerda and Friedrich, for their unconditional support.

Kurzfassung

In dieser Arbeit präsentieren wir einen theoretischen Ansatz für automatische Musikkomposition mit formalen Grammatiken und einem Fokus auf polyphonen Strukturen. Eine polyphone Struktur beschreibt in diesem Zusammenhang eine Menge musikalischer Entitäten (Noten, Akkorde, Takte, usw.), die nicht ausschließlich sequentiell in der Zeit angeordnet sind. Nachdem das Ergebnis einer Grammatik üblicherweise als Sequenz dargestellt wird, ist die Erzeugung sequentieller Strukturen wie Melodien, Harmoniefolgen und rhythmischer Muster naheliegend und wurde bereits in früheren Arbeiten erforscht. Im Gegensatz dazu assoziieren wir jede musikalische Entität mit einer unabhängigen Zeitspanne, was die Darstellung von sowohl sequentiellen als auch parallelen Anordnungen ermöglicht. Mit überlappenden Entitäten können wir Akkorde, Schlagzeugmuster, und unabhängige Stimmen modellieren—Polyphonie im kleinen und großen Maßstab. Neben einer grundlegenden Diskussion funktionaler Techniken für polyphone Komposition mit nicht-deterministischen, kontextfreien Grammatiken präsentieren wir die Implementierung und praktische Anwendung eines automatisierten Kompositionssystems, das auf diesen Prinzipien basiert.

Abstract

We present a theoretical framework for automatic music composition with formal grammars and a focus on polyphonic structures. In the context of this thesis, a polyphonic structure is any arrangement of musical entities (notes, chords, measures, etc.) that is not purely sequential in the time dimension. Given that the natural output of a grammar is a sequence, the generation of sequential structures, such as melodies, harmonic progressions, and rhythmic patterns, follows intuitively and has already been explored in prior works. By contrast, we associate each musical entity with an independent temporal scope, allowing the representation of both sequential and parallel arrangements. With overlapping entities we can model chords, drum patterns, and parallel voices—polyphony on small and large scales. Beyond a foundational discussion of functional techniques for polyphonic composition with non-deterministic context-free grammars, we demonstrate the implementation and practical application of an automated composition system developed on these principles.

Contents

CHAPTER 1

Introduction

Music is a phenomenon that may[1] involve sound. Sometimes these sounds are arranged according to rhythmic and harmonic principles, and sometimes their arrangement is deliberately chaotic. Music can be meaningful and invoke deep emotion. It can also be a commodity and serve as a comforting backdrop to our everyday lives. Whatever words we use to describe music, we will hardly do it justice. When we talk about music we remove its character; maybe especially so when we talk about it as theorists do, in terms of notes, bars, scales, and meters. When we talk about music we have to do so in an abstract sense because we cannot talk about it in any other way. This is not a strange or problematic thing to do, as long as we accept that the concept of language is not itself inherently strange and problematic.

In this thesis the concepts of language and music are strongly interweaved since we will use a formal model of language to automate the composition process. The automation of musical composition certainly has interesting implications. We can already see its success in computer games, which, due to the nature of the medium, cannot have a static soundtrack. Themes and sounds are triggered by user actions and are combined into something that has never been heard before. Similar systems could be built for video platforms, where we could offer generative soundtracks that adapt to visuals, dialog, cuts, and camera movements. We might someday have procedural radio, that can be configured to produce a lifetime of music in a personalized style, that reacts to the listener's mood, and from time to time plays variations of their favorite melodies. Procedural music can also be integrated into concerts, where the artist operates as the programmer and conductor for an orchestra of virtual musicians. These things already exist to different degrees, and countless musicians and composers rely on automation in their everyday work. Generative grammars—the subject of a large body of research, including this thesis—are one of the tools in the algorithmic and computer-aided composition toolbox.

[1] Consider Wikipedia's list of silent musical compositions.

A core function of language is that it allows us to put the world into simple terms, strip away the concrete and discover abstract relationships between its entities. Words are placeholders for real things, for imaginary things, and sometimes for other words. They signify something that has been replaced and they must be replaced again so we can understand their meaning. The theory of formal grammars is a theory about systems of symbolic replacement. It is an abstraction of language, the very tool that we use to build abstractions in the first place. Here we no longer talk about the meaning of a word in relation to something real or imaginary, but the relationship of words with each other. Meaning only exists in the rules of the system—arrows from one arrangement of symbols to another. What this theory reveals are the patterns, the repetitions, and the variations. Like a story, music contains recurring themes, characters, and motifs. Unlike a story it does not necessarily relate to human experience, or anything external at all. It is poetry without semantics, an art of syntax.

When we use grammars to syntactically analyze music, we break it into its smallest components, then group them into notes, bars, phrases, themes, or whatever higher-level patterns we find. The premise of using grammars generatively is that we can reverse this process of reduction, beginning at an abstract representation and replacing the abstractions until we get something concrete. For music this could mean starting with a particular style or song structure, successively adding themes, phrases, bars, notes, and finally transforming these back into an audible signal. In order to create something that corresponds to our abstraction, we have to make a series of informed decisions. Some of these decisions will be guided by the model, for example the rules of the musical genre. Other decisions will depend on subjective taste. A composer is aware of these options, yet is forced to make a decision once they write their score. A system, such as the one developed in this thesis, uses a special notation that allows the composer to state the options, but defer the decision. The system, aware of all options, can automatically collapse this superposition according to a set of parameters or random chance. The result is a new song, unheard even by its composer.

The application of formal grammar theory for music analysis and composition has a long history. The use of abstractions in musical notation certainly predates the first formalization of grammar theory by Chomsky [Cho56]. Schenker's work [Sch35] is an early example of a generative approach, where music is reduced to the *Ursatz*, a hidden structure beyond the concrete musical surface. One of the most popular works in the field is *A Generative Theory of Tonal Music* by Lerdahl and Jackendoff [LJ+83], who model music as four hierarchical aspects governed by strict well-formedness rules and soft preference rules, but has mostly seen application for music analysis. Applications of grammars for particular music generation tasks include works on melodies [LS70, LG73], harmonic progressions [Roh07, Ste96], and tabla syntax [BK92a]. More general solutions for music generation with grammars include the *generative grammar definition language* by Holtzman [Hol80], a work by McCormack [McC96], and more recently a polyphonic generator by Tanaka and Furukawa [TF12], probabilistic temporal graph grammars (PTGG) by Quick and Hudak [QH13a], and extensions to PTGGs by Melkonian [Mel19].

While these works provide evidence for a connection between language and music and their corresponding theories, there is an important difference: we usually see language as something strictly linear. A symbol comes after a symbol, a word comes after a word, and a sentence comes after a sentence. This might explain why existing research on grammars for composition focuses on the generation of melodies, harmonic progressions, rhythms—all purely sequential structures. Yet, within a sequential model of time the representation of polyphonic aspects such as chords, parallel voices, or exotic sound effects is difficult. Note that we use the term *polyphony* to refer to musical structures that are not exclusively sequential, thus essentially meaning *non-monophonic*, in contrast to its more specific use for music that has multiple leading voices. The generation of polyphonic structures in prior works was either not possible within the grammar, requiring a separate processing step [QH13b], or at least limited to chords on the terminal level [McC96] or a fixed number of voices on the global level [TF12].

In this work we build a perspective where polyphony is the norm and tightly integrated into the generation process. We achieve this by explicitly associating each musical entity with a time interval, which means they can be divided, stretched, and moved independently from each other. This representation is already common practice in musical data formats such as MIDI, but has, to our knowledge, not been explored in conjunction with generative grammars. Since the placement of entities is truly unrestricted, our model is able to generate polyphony on any level of the composition. The use of arbitrary interval arrangements leads to additional complexity for the composer, which we alleviate with operators for splitting, tiling, trimming, and querying time intervals. The latter provides access to local context, such as an underlying harmonic progression, and allows us to synchronize multiple voices. This synchronization is especially important in a polyphonic model, as the parallel parts can be generated separately, but will still fit together in the end.

We will proceed in Chapter 2 with an elaboration of existing works and conclude it by further contrasting the state of the art with our own work. In Chapter 3 we construct a functional framework on the principles of context-free grammars, and discuss the use of attributes, rule application strategies, structural, and parametric variation. On top of this generic framework we define *probabilistic temporal-split grammars* in Chapter 4, and demonstrate their practical application for music composition at the end of the chapter. In Chapter 5 we transform the theory into a working program and present a browser-based interface and playback environment, followed by a discussion of the results and further comparison in Chapter 6. Conclusions will be presented in Chapter 7.

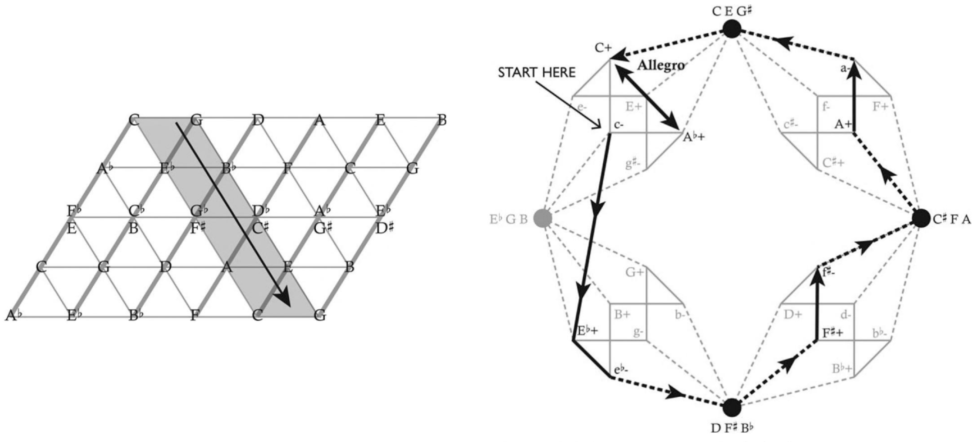
CHAPTER 2

Related Work

At its core, music composition appears to be a fundamentally free expression of human creativity. Yet, musicologists can provide us with a variety of formal models for describing different aspects and styles of music. Works on formalizing tonal music include the *Tonnetz*, first proposed by Euler [Eul74], which places pitches on a triangular grid and relates them by thirds and fifths, as shown in Figure 2.1a. It is part of a category of works on the definition of musical spaces, which reduce the complexity of navigating between related musical objects. *Tonal Pitch Space* by Lerdahl [Ler04] is another example, in which the author defines metrics for computing the distance between pitches and pitch sets. And Douthett and Steinbach [DS98] propose the *Cube Dance*, shown in Figure 2.1b, which relates the twenty-four major and minor triads and four augmented chords by semitone alterations.

Instead of describing particular aspects, such as pitch relations or harmonic progressions, some authors have attempted to find more comprehensive models of music. Schenker [Sch35] modelled music as a foreground level, that contains the actual notes of a piece, and various hidden levels, which describe the foreground in terms of abstract representations. Musical set theory, proposed by Hanson [Han60], uses sets of pitch classes as its basic entity, that can manifest as various concrete musical objects, such as chords or short motifs. Lerdahl et al. [LJ$^+$83] use four hierarchical layers to analyse music, and we will later describe their model in more detail. Looking again at the *Cube Dance* in Figure 2.1b we can imagine a layman, or a simple probabilistic algorithm, that traces random paths through the graph and comes up with new harmonic progressions. The level of abstraction is so high that any move along an edge will result in a valid, or even pleasant musical idea.

The notion of validity in music is of course highly subjective, but within a piece, or group of pieces of a particular style, it is usually possible to find higher-level organization. This organization is sometimes so heavily implied by the context that a subversion can seem erroneous. Some musicologists have noted parallels between our ability to



(a) *Tonnetz* visualization. (b) *Cube Dance* visualization.

Figure 2.1: Figure 2.1a and Figure 2.1b describe the harmonic progression of the opening measures of Schubert's Overture to *Die Zauberharfe* in terms of the *Tonnetz* and *Cube Dance* respectively. The discretization of the sonic experience allows us to reduce a complex musical arrangement to a sequence of primitive moves along the edges of a graphical structure. Both graphics were borrowed from Chapter 5 of *Audacious Euphony* [Coh12].

recognize a valid sentence in a natural language and recognizing valid compositions, when compared to random arrangements of their respective elements [Ste84]. Many authors agree that music can be described—at least to some extent—by linguistic rules, which allow us to recognize larger structures in some combinations of musical entities, but not in others [Coo59, Win68, Kei78]. Potential evidence for this connection was found by a study on neuron activation in the human brain, which shows very similar responses for the perception of music and language [BMP06].

In this work we use a linguistic approach towards musical modelling that builds on the theory of formal grammars. Naturally, we are most interested in works that use grammars for musical analysis and generation. The broader class of methods for algorithmic composition includes, besides grammars, self-similar systems, evolutionary algorithms, Markov models, and artificial neural networks. Note that these are only the most popular methods and those that we will discuss in the first half of this chapter. Due to the broad spectrum of approaches and general difficulty of assessing objective quality of a composition, attempts have been made towards a structured evaluation of algorithmic composition systems. Some authors have defined quantitative metrics [PW01, YL20], others rely on qualitative models [EBPM13]. For a critical discussion on how a failure to specify concrete goals has led to methodological problems in automated composition research see Pearce et al. [PMW02].

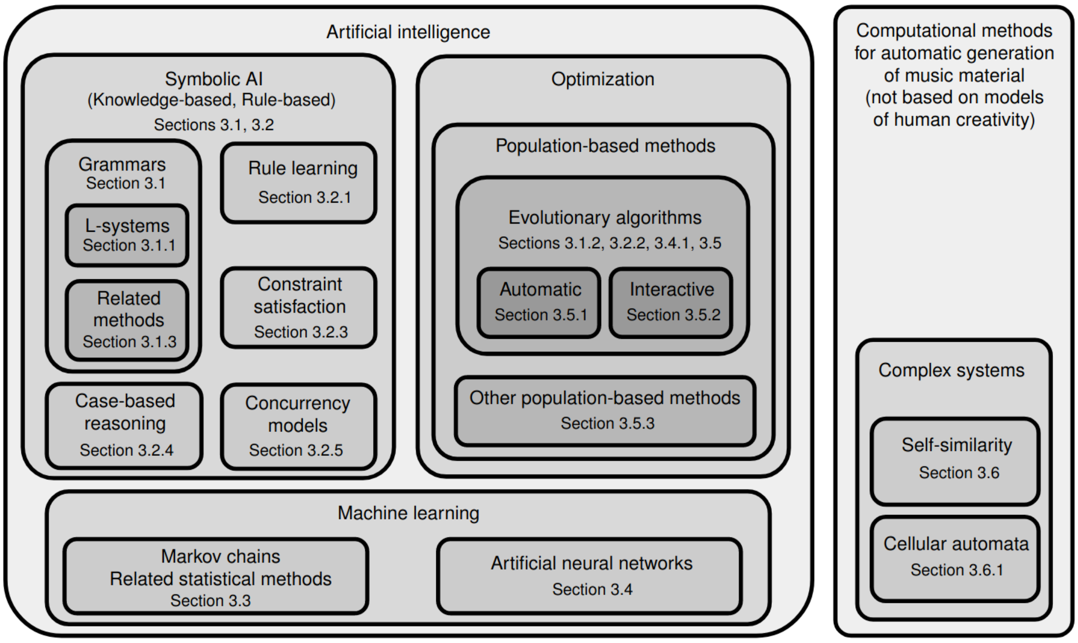


Figure 2.2: A visualization of the taxonomy of algorithmic composition approaches, as provided by Fernández and Vico [FV13]. The section numbers relate to their survey, not to this thesis.

2.1 Algorithmic composition

We can look at the content of this thesis in the context of algorithmic composition research, where grammars are just one particular model for the composition process. Nierhaus [Nie09], as well as the earlier survey by Papadopoulos and Wiggins [PW99], use a flat classification system, where generative grammars, evolutionary methods, and machine learning are the common clusters. Fernández and Vico [FV13] propose a detailed hierarchical taxonomy of methods and provide a very useful visualization, which we show in Figure 2.2. Herremans et al. [HCC17] use a taxonomy that focuses on the particular challenges of the composition process, such as providing an emotional narrative, and the components of a piece, in particular melody, harmony, rhythm, and timbre.

A comprehensive survey on the field of algorithmic composition is beyond the scope of this thesis and again we point to the authors above. Nevertheless, in this section we want to present what we see as the largest methodological clusters, summarize their core ideas and relate them to the grammar approach. We will use a flat classification, beginning self-similar systems and their relation to formal grammars through Lindenmayer systems (L-systems). We will then look at methods that treat the creative process as a search problem, in particular solutions involving evolutionary algorithms since they are often used in hybrid systems and have been used for discovering new grammars. Finally, we will present two machine-learning approaches: Markov models—since historically

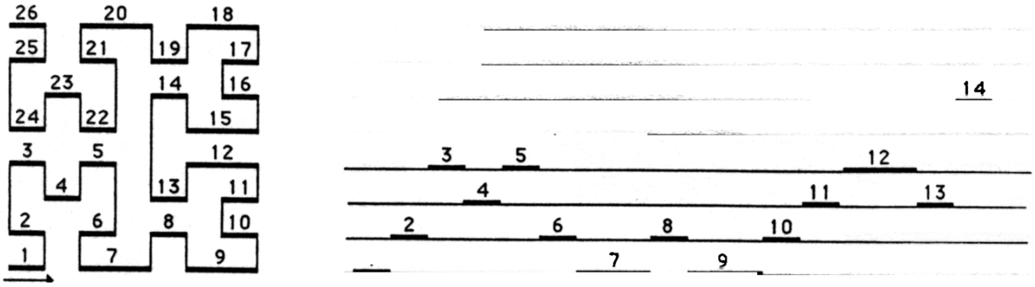


Figure 2.3: Space filling curves, such as the Hilbert curve (left), can be interpreted musically. In this case, the horizontal movements are mapped to the time dimension, and the vertical movements to the pitch dimension, resulting in the contour of a melody (right). This is one of the earliest uses of L-systems for composition, and it exploits the self-similar nature of the result, rather than using rules tailored to the music domain. The above graphic was reproduced from a work by Prusinkiewicz [Pru86b].

there has been a lot of research, and the core idea runs, in a way, orthogonal to that of grammars—and ultimately artificial neural networks. While the methodological divide between deep learning and grammars is quite large, we include it due to the circumstance that many works in that field emerged only recently.

2.1.1 L-systems

L-systems, originally proposed by Lindenmayer [Lin68], are a type of formal grammar where rules are applied in multiple parallel passes, as opposed to grammars in the Chomsky hierarchy, where all replacement happens sequentially. While originally developed to describe cellular plant growth, they have been applied to graphics and music generation. The classification of L-systems in the context of algorithmic composition varies, with some authors grouping them with grammar-based approaches [FV13], and others treating them as self-similar systems [Nie09]. An argument for the latter can be made, that some authors focus more on creative mappings of the terminal string and even musically reinterpreting L-system results from other domains, rather than finding domain-specific rules. While not as relevant as the works in the latter half of this chapter, L-systems share many features with other grammar-based approaches and are thus still important in the context of this thesis.

Prusinkiewicz showed how L-systems can be used to generate fractal curves [Pru86a] by interpreting the generated strings as input for a turtle [AD86] and later applied this principle to score generation [Pru86b]. Rather than using the path of the turtle as graphical output, they converted the movement to notes, by interpreting horizontal movements as note lengths and vertical movements as scale steps. An example is visualized in Figure 2.3. Other authors also used mapping functions based on space filling curves, such as Mason and Saffle [MS94] and Nelson [Nel96]. Wilson [Wil09] mapped the branch points and angles of a generated tree graphic to note events.

Some authors used interpretations that do not rely on graphical mappings. Langston [Lan89] proposes the use of sentences generated by bracketed L-systems as keys to a lookup structure of predefined melodic and rhythmic patterns. McCormack [McC96] used pitch letters as an alphabet to generate melodies and even added a parameter for note volume. Pestana [Pes12] also uses pitch letters together with a post-processing step for developing the melodies into polyphonic pieces. Morgan [Mor07] also mapped the L-system output to sets of fixed patterns and assembled them into a multi-instrument piece. DuBois [DuB03] and Manousakis [Man06] present various interpretations and examples of L-systems for music composition. Kaliakatsos-Papakostas et al. [KPFKV12] define *finite L-systems*, which allow the generation of rhythms that are constrained to a certain metric structure, as opposed to most other works, were the music grows freely and without temporal bounds.

Concepts from music theory have been used to find appropriate mappings and rules. Worth and Stepney [WS05] compare plant generation and music generation and propose a Schenkarian inspired [Sch35] rendering approach of background, middleground, and foreground, where only the latter can be heard and consists of the leaves of the tree structure. Gogins [Gog06] uses a turtle that moves within chord and voice-leading spaces. As we have seen, some authors put effort into finding sophisticated mappings of non-domain-specific terminals, others try to use a vocabulary close to musical structures. Especially in the latter case, which is similar to our own approach, the creative effort goes into finding the right rules. The next section includes some examples of evolutionary optimization for discovering rules for musical L-systems.

2.1.2 Evolutionary Methods

We can see composition as the process of finding aesthetic combinations in the space of all possible combinations of sounds or notes. To make this huge search-space manageable, various researchers have used a cycle of evaluation, selection, and reproduction to raise the average quality of a set of candidate solutions. The domain knowledge is encoded in the population and introduced by eliminating bad individuals according to the *fitness function*. The definition of this fitness function is crucial and often difficult, due to the subjectivity in the judgement of musical quality. According to Fernández and Vico [FV13] evolutionary methods are commonly used in hybrid systems. For example, rather than selecting and recombining the notes of multiple melodies, we can select and recombine the rules of multiple grammars that generate melodies.

Horner and Goldberg [HG91] use genetic algorithms for transforming one melody into another, leading to the generation of intermediate variations. McIntyre [McI94] harmonize melodies according to rules of baroque harmony. A more general approach towards harmonization was proposed by Phon-Amnuaisuk et al. [PATW99], who introduced explicit domain knowledge to steer the evolution towards correct harmonization in arbitrary keys. Evolutionary algorithms have been applied to generating rhythmic patterns of one bar [Hor94] and four to sixteen bar lengths [TI00]. Another application is the generation of counterpoint, attempted by Polito et al. [PDBB97] with a multi-agent genetic programming system, and Acevedo [Ace04], who generated voices for Fugues.

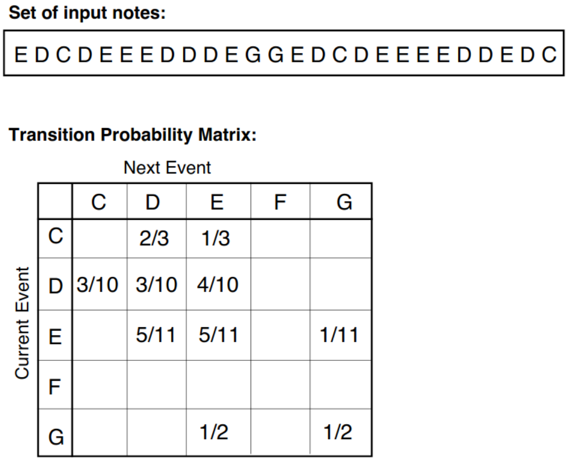
A popular work by Biles et al. [Bil94] describes *Genjam*, an interactive jazz solo generator. The system receives metric structure and harmonic progression as input, over which it generates melodic phrases that are rated by a human mentor. They describe human mentors as a *fitness bottleneck* because their limited bandwidth, subjectivity, and inevitable fatigue prevents the system from quickly converging towards good solutions. They later trained an artificial neural network to perform the evaluation [BAL96]. Papadopoulos and Wiggins attempted to solve the fitness bottleneck with a set of objective criteria [PW98]. Similarly, Towsey et al. [TBWD01] propose the use of global statistics based on a corpus of existing music as input for the fitness function. They further identify three types of fitness functions: the human critic, the rule-based critic, and the learning-based critic. They conclude that the fitness function ideally incorporates multiple sources of domain knowledge.

Lourenço [LRB09] and Kaliakatsos-Papakostas et al. [KPFKV12] apply evolutionary programming on a more abstract level and use it to find rules for L-systems, which they represent as strings of turtle commands and note events respectively. While this works well enough for simple rules, it may be difficult to define mutation and crossover operators that adhere to more complex syntactic systems. For example, the grammar that describes the syntax of our own grammar definition language is at least context-free. The operators would have to replace and swap nodes in the corresponding syntax trees, a potentially much larger search space. Nevertheless, solutions for automatic grammar inference are very interesting since the grammar formalism on its own does not help with the discovery of the rules of a domain. This is a major disadvantage of grammars, when compared to evolutionary algorithms, and the machine learning methods that we will discuss next.

2.1.3 Markov Chains

A Markov chain represents sequential data as a probabilistic transition between states, and can be used to predict the next state based on one or more prior states. For example, if the states are notes, chords, or rhythmic intervals, the Markov chain could generate new melodies, chord progressions, or rhythms. The transition probabilities are most commonly represented as state transition matrices, such as the one shown in Figure 2.4, which can either be learned from a corpus, or designed by hand-picking the weights. The timeline of the generation process runs parallel to the timeline of the piece. This differentiates their mode of operation from formal grammars, where we usually consider a top-down perspective and can develop multiple sections of a piece simultaneously. According to Fernández and Vico [FV13] and our own literature research, Markov chains are quite commonly used in composition, and we will only cite a small subset of the works.

Ames [Ame89] provides an early survey of Markov based composition processes. Another survey by Conklin [Con03] looks at statistical models for music generation and discusses



Set of input notes:

E D C D E E E D D D E G G E D C D E E E E D D E D C

Transition Probability Matrix:

Next Event

Current Event

	C	D	E	F	G
C		2/3	1/3		
D	3/10	3/10	4/10		
E		5/11	5/11		1/11
F					
G			1/2		1/2

Figure 2.4: The melody of the nursery rhyme *Mary had a little lamb* as a sequence of pitch letters is shown above. Below, the derived transition probabilities for a first order Markov chain are visualized as a matrix. The original graphic can be found in McCormack's paper on grammar based music [McC96].

different types of Markov models. Concrete uses include Farbood and Schöner [FS01], who generate counterpoint solutions with probability tables estimated from existing works. Allan and Williams [AW05] as well as Yi and Golsmith [YG07] used Markov decision processes for harmonization. Chuan and Chew [CC07] and Simon et al. [SMB08] generate accompaniment, the latter with *hidden Markov models*, where only the outcomes are known, but not the states. Hawryshkewich et al. [HPE10] and Tidemann and Demris [TD08] generate drum patterns. And Herremans et al. [HWSC15] generate music for bagana, a traditional Ethiopian instrument, using Markov chains and variable neighborhood search. There are also some interesting hybrid approaches, in particular Gillick et al. [GTK10], who apply them to grammar inference. These references only scratch the surface, but one can see that Markov models have been applied to a wide variety of tasks in composition.

Clement [Cle98] evaluated the capabilities of Markov models trained on various regular languages and conclude that Markov chains are powerful enough to generate certain harmonic progressions, but might be insufficient for modelling more complex musical aspects. Ames [Ame89] argues that mathematical linguists have unfairly dismissed Markov chains as lesser grammars, while their expressive power is comparable to that of context-sensitive grammars. We would argue that it depends on the particular formalism since the simplest type, where only the previous state determines the next state, can be trivially modelled by a probabilistic type-3 grammar, but higher-order Markov chains, where at least two prior states are considered, would generally be context-sensitive. On that topic, Moorer [Moo72] mentions how lower order Markov chains trained on existing music lack necessary structure in their generations, while higher order chains tend to over-fit. We earlier classified Markov chains as a machine-learning approach. Yet, while the transition probabilities are often derived from a corpus, lower order chains are still transparent enough to be configured manually. In contrast, artificial neural networks are exclusively trained on examples, and we will discuss them in the next section.

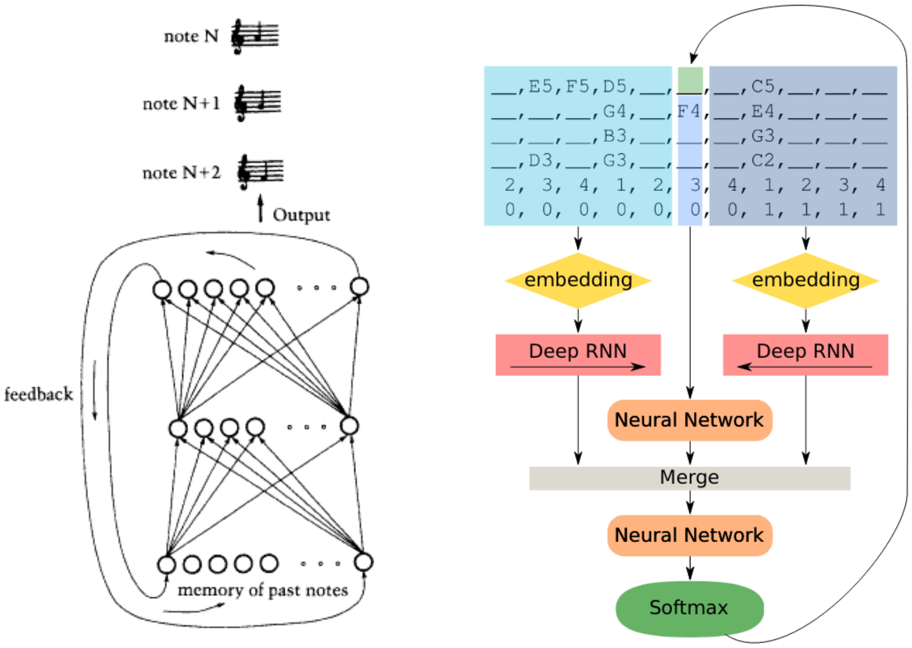
2.1.4 Artificial Neural Networks

Intuitively, artificial neural networks (ANNs) try to replicate the learning capabilities of biological neural networks. The idea is to approximate a complicated function with a highly interconnected and general graphical structure, in which particular connections are incrementally strengthened or weakened, depending on the difference between desired and actual output. Once trained, such networks can be used, not only to parse, but also generate new data. The application of artificial neural networks to problems traditionally related to grammars has been very successful. Consider, for example, the recently released GPT-3 model for natural language processing [BMR$^+$20], which can consistently generate syntactically and often even semantically valid output in an impressive range of natural and formal languages. Given that musical structure is largely syntactical, successful applications of deep learning for music generation are to be expected.

There are several challenges involved with setting up such networks. Theoretical challenges include finding a suitable representation for the data the network should process and a corresponding definition of an input and output surface, as well as the configuration of the internal network layout. Practical challenges include the gathering of examples and ultimately training the network, which can be computationally expensive. In recent years deep learning research has picked up pace, not least because hardware capabilities are finally catching up to the incredibly demanding training procedures. While ANNs have been applied to composition for over thirty years, starting with networks such as the one proposed by Todd [Tod89] and sketched in Figure 2.5a, a disproportionate number of works have surfaced only over the past decade. Again, we will present only a subset of the available literature and refer the reader to the dedicated surveys.

Once again, we refer to Fernández and Vico [FV13] for a discussion of works that were published before 2013. For example, Hild et al. [HFM92] present a neural network called *HARMONET* for approximating harmonic structure of chorales, which was later combined with *MELONET*, a network for melody generation [Hör98], and *CHORDNET*, a network for finding chord sequences based on learned voice-leading constraints [Hör04]. Mozer [Moz94] uses a recurrent neural network (RNN) called *CONCERT* to predict notes and chords from a given sequence and managed to generate repeating structures to a limited extent. Plain RNNs had problems with keeping track of global structure, so long short-term memory (LSTM) recurrent networks were proposed as an alternative by Eck and Schmidhuber [ES02] and applied to Blues improvisation. Franklin [Fra04] compared the effectiveness of various pitch representations for use in LSTMs, also noting the superiority of LSTMs over plain RNNs, and later used them to generate Jazz melodies [Fra05, Fra06].

Over the past decade the field of ANN research expanded drastically, with new works



(a) Todd [Tod89].

(b) Hadjeres et al. [HPN17].

Figure 2.5: Figure 2.5a shows a sketch of an ANN from an early work and Figure 2.5b shows a modern network design, which consists of various stages, many of which host sub-networks that contain orders of magnitudes more neurons than the early network. While the newer architecture is significantly more sophisticated and computationally expensive, they are similar in that they both recurrently predict notes based on prior network output. This is also reminiscent of higher-order Markov chains, at least from an abstract operational perspective.

being published at an increasing rate, even though music generation is still a relatively minor focus compared to applications in image and natural language processing. Briot et al. [BHP17] wrote an extensive two-hundred page analysis, detailing and comparing various techniques and network types. They classify works in terms of five interdependent dimensions: target output, musical representation, network architecture, limitations in terms of creativity and variability, and the strategy that controls generation. Notable works in the modern era of deep learning research include Google's *Melody-RNN*, which was built in the context of their musical AI research project *Magenta*, *Song from PI* by Chu et al. [CUF16], a hierarchical RNN that generates multi-track pop music, and *WaveNet* by Oord et al. [ODZ$^+$16], which is an example of a network that generates raw audio, rather than symbolic music. As an alternative to RNNs, generative adversarial networks (GANs) have been used by Yang et al. for *MidiNet* [YCY17], which generates melodies, and for *Musegan* by Dong et al. [DHYY18], which generates multi-track music.

A recurring objective is the generation of Bach music, likely due to the large and readily available corpus and consistent style. A prominent example is *DeepBach* by Hadjeres et al. [HPN17], which can generate polyphonic pieces and comes with an interactive graphical interface. Recently, the *Bach Doodle* by Huang et al. [HHR$^+$19], which can harmonize short melodies, has received wide-spread attention when it was used on the Google search homepage as an interactive widget. Liang [Lia16] demonstrated the strengths of deep learning for composition when they tested the *BachBot* in an online Turing test. Participants were asked to differentiate between artificial and actual Bach music and only one out of ten was able to do so reliably. They further studied network activations, revealing the network independently learns concepts from music theory, mirroring earlier work in the field of image recognition [ZF14], where it was found that the layers of convolutional networks learn increasingly abstract representation of the data.

Undoubtedly, deep learning already achieves very impressive results and, unlike grammars, does not require explicit knowledge of music theory. Unfortunately, the models learned by the network are often inaccessible to humans and, once the expensive training process has completed, the network is inert and limited to the subset of knowledge it gathered from the training set. In a way, we can place ANNs and grammars on opposite ends of a spectrum, where ANNs require minimal domain knowledge, but also offer a very low level of control, and grammars, unless inferred by other means, require lots of domain knowledge, but offer ultimate control. Control, in this sense, is strongly related to transparency. It takes significant analytic effort to determine the abstract reason for why a neural network arrived at a particular result, while the same question is trivially answerable for most grammars. As such, ANNs and grammars are able to fulfill opposing, but not at all conflicting needs, where the former is useful if we want to replicate a model without having to understand it, and the latter is useful when we already have a model and want to test and improve it through its application.

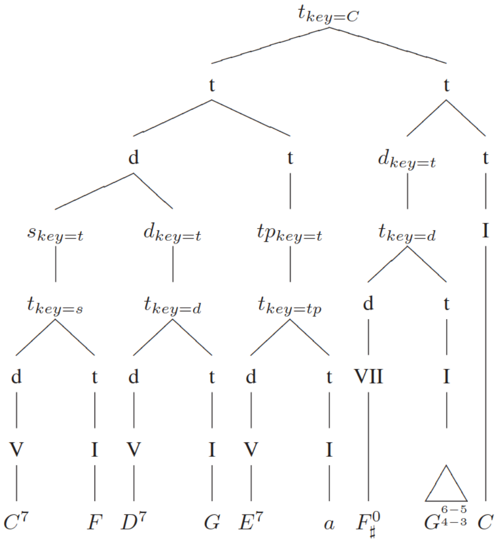
2.2 Grammar-based approaches

With generative grammars we use a set of formal rules to develop an abstract symbolic structure through iterative refinement into concrete and detailed output. For example, consider Figure 2.6, which shows the generation of a harmonic progression starting from a single tonic key. A prominent and early generative approach was conceived by Schenker [Sch35]. His assumption was that we can reduce many forms of music to the *Ursatz*, a sort of fundamental, hidden structure, not unlike the *axiom* in formal grammars. In practice, the Schenkerian approach is mostly used for analysing existing scores, with some success towards automating this analysis [Smo80, KU08, Mar10], but it is still recognized by numerous works in the field. The importance of the works by Chomsky with regard to formal grammar theory cannot be overstated since his definition of a grammar hierarchy allows us to classify and assess the expressive power of particular grammars [Cho56]. There are further subcategories and variants of these grammar types, which we will discuss in the context of particular works.

The basis for the grammar approach is a recursive model of musical structure. One

$$
\begin{array}{rcll}
t & \rightarrow & t_{key=x} & (1)\\
t & \rightarrow & t \;\; t & (2)\\
t & \rightarrow & t_{\llcorner}\; d & (3)\\
t & \rightarrow & d \;{}_{\lrcorner}t & (4)\\
d & \rightarrow & s \;{}_{\lrcorner}d & (5)\\
\\
t & \rightarrow & I & (6)\\
t & \rightarrow & tp & (7)\\
tp & \rightarrow & \begin{cases} VI & \text{if key is major}\\ \{VI, III\} & \text{if key is minor}\end{cases} & (8)\\
d & \rightarrow & \begin{cases} \{V, VII\} & \text{if key is major}\\ \{V, \natural VII\} & \text{if key is minor}\end{cases} & (9)\\
s & \rightarrow & \{IV, II, VI, \flat II\text{(in minor)}\} & (10)
\end{array}
$$

(a) A grammar for harmonic progressions.



(b) A derivation tree.

Figure 2.6: Figure 2.6a shows a parametric context-free grammar as a set of ten rules, each describing the replacement of a particular musical entity on the left side of the arrow with the entities on the right side. The recursive application of these rules leads to the development of a tree structure. Figure 2.6b shows how the grammar might generate the harmonic progression of Bortnianski's *Tibje Pajom*. The original graphics can be found in Rohrmeier's work [Roh07].

concrete implementation of such a hierarchy is provided by Buxton et al. [BRBM78], who defined a score representation for use in the *Structured Sound Synthesis Project*. Roads and Wieneke [RW79] review early uses of grammars as music representation and discuss the appropriateness of formal grammars for modelling music in general. They argue, that the use of grammars forces a discretization of music and is biased towards hierarchical, multi-level structures, due to the nature of the derivation process. Nevertheless, they think context-free grammars are a sufficiently powerful tool to model many musical structures. Regular grammars were deemed too limited, while more general grammar types with full context-sensitivity were judged as too complex for practical use. Bod [Bod02] provides further perspectives on the role of hierarchies and tree structures in human perception of language and music, and Yust [Yus09] explores the mathematical features of hierarchical geometric structures in relation to melody, rhythm, and harmony.

A Generative Theory of Tonal Music (GTTM) by Lerdahl and Jackendoff [LJ$^+$83] is one of the most popular works in the field of generative music theory. The authors provide a formal framework consisting of four hierarchical aspects. (1) Grouping structure describes how neighboring events are perceived as a whole as motives, phrases, and larger structures. (2) Metrical structure provides a regular grid of strong and weak attack points, on which events are located. (3) Time-span reduction combines the grouping and metrical structure into a hierarchy of time intervals. Finally, (4) prolongational structure describes the perception of harmonic progression, tension, and release, and mends disconnects that arise from using time-span reduction on its own. Each of these four aspects is encoded as sets of strict *well-formedness rules*, *preference rules* for accommodating a listener's perceptions, and *transformational rules* for resolving distortions. Efforts have been made to automate musical analysis with GTTM [HHT06, HHT07] with some success in synthesis through alteration of the derivation tree.

Grammars can be used to describe the transitions between abstract and concrete structures in music and therefore lend themselves to the automation of both generation and analysis. Since the domain knowledge is encoded in the rules, finding appropriate rules and associated probabilistic weights to resolve ambiguities is the creative step. Many authors resort to hand-picking rules, but one can also automatically derive rules and weights from existing works, a process which is called *grammar inference*. Kippen and Bel [KB89] used grammar inference to learn Indian drum music. They used an iterative approach of generalization, variation, and verification through human experts. Cruz-Alcázar and Vidal-Ruiz [CAVR98] compared multiple grammar inference algorithms and musical coding schemes and used them to analyze and synthesize melodies in different musical styles. Sidorov et al. [SJM14] treat music as a smallest grammar problem and approximate the shortest grammars for representing the temporal structure of fugue voices. Various authors compute production probabilities for a given set of rules and corpus of existing pieces [GC07, Qui14].

2.2.1 Grammars for Composition

The works presented so far relate music and grammars in a general sense, with a focus on analysis over synthesis. We will now look at some concrete works and applications of grammars for composition and their particular features. The references in this section are definitely the most relevant in the context of our own work. In an early work, Lindblom and Sundberg [LS70] analyzed the melodies of various nursery rhymes and derived production rules. Due to the fixed structure of eight bars, they were able to tailor rules to particular parts, for example, guaranteeing a concluding half note and tonic harmony at the end of every piece. Most of the rules are defined through tables, sketches, or plain text, but they appear to be at least context-sensitive. Another early work by Lidov and Gabura [LG73] present rules for modelling melodies as a hierarchy of semantic labels from the phrase level down to the measure level and a set of rules for generating melodic movement within measures.

Holtzman [Hol80, Hol81] defined the *Generative Grammar Definition Language* (GDDL) and applied it to music generation. Beyond standard features for non-deterministic derivation, such as production probabilities, GDDL provides interesting meta-level features, which allow the selection of alternatives based on prior rule applications. For example, one can control the probability of going from one alternative to another using a finite-state transition matrix, or guarantee that all alternatives are selected at least once before one is repeated. They demonstrate these capabilities by replicating a piece by Arnold Schönberg, who is known for using similar restrictions in his compositions. For repetition, GDDL uses special rules, which they call *meta-productions*, that replace all occurrences of an LHS with a particular RHS .

McCormack [McC96] surveys multiple approaches for algorithmic composition, including L-systems, Markov chains, and formal grammars. They propose an advanced musical grammar system that utilizes various meta-level features, such as stochastic rule selection, numeric parameters, and nested grammars. Symbols in their grammar carry pitch, duration, timbre, and control parameters. While the generations are primarily sequential, limited polyphony can be achieved by marking multiple notes as a chord. Preceding notes can be used for matching, which allows for limited context-sensitivity.

Steedman [Ste84] defined a grammar for generating chord sequences for 12-bar Blues, which consist of three phrases of four bars in common time. Each node in the derivation tree has a duration, a harmonic function, and a chord type parameter, each of which can be inherited or removed on the RHS. The replacement entities divide the time interval of the original entity into equal parts, guaranteeing a monophonic and bounded temporal organization. The grammar is context-sensitive, which allows for powerful matching criteria, for example, matching the second chord of a cadence. They later revised their grammar and showed that it can be expressed by an equivalent context-free grammar [Ste96]. Chemillier [Che04] discusses the integration of this grammar in a real-time system and analyzes particular classes of generated progressions. Recently, Melkonian [Mel19] replicated this grammar, which will be discussed later.

On the topic of generating harmonic progressions, Rohrmeier [Roh07, Roh11] derived harmonic substitution rules from various works on harmonic theory. The rules match on harmonic functions and mode, as shown in Figure 2.6a, and the right-hand side allows parametric embedding of the replaced symbol in a new context, relative harmonic changes, and the addition of accidentals. The grammar is context-free, except for a special rule for pivot chords, which they later reformulate without context-sensitivity. De Haas et al. [DHRVW09] remodelled Rohrmeier's earlier grammar and applied it to automatic parsing of jazz pieces.

There has been some interest in replicating improvisation on the tabla, a Northern Indian two-piece drum set with formal grammars [Bel89, BK92a, BK92b, Bel92, Mel19] and Markov chains [CSŞ11]. The music is traditionally communicated using *bols*, which are comparable to syllables in natural language, and follow strict syntactic rules. Bel and Kippen [BK92b] used these syllables as a vocabulary for generating words of a type-2 language and used a type-0 grammar for combining these words into larger sequences. The results of the type-2 system were also replicated by Melkonian [Mel19].

Gilbert and Conklin [GC07] defined *probabilistic context-free grammar* (PCFG) for melody generation which uses pitch intervals as non-terminals. The use of intervals, each of which encodes information about two adjacent notes, allows them to reformulate rules that would normally be context-sensitive in a context-free manner, for example a rule for generating passing notes. Keller and Morrison [KM07] defined another PCFG for generating jazz melodies. The terminals of the grammar encode various semantic roles which are realized in a post-processing step. For example, an *approach tone* forces the next note to be a tone of the chord. Additional constraints allow the specification of a pitch range and leap intervals. They use the note type as a parameter so rules can be defined that apply to non-terminals of varying durations.

Quick and Hudak generally required the use of a time parameter in their *temporal generative graph grammars* (TGGG) [QH13b] and later *probabilistic temporal graph grammars* (PTGG) [QH13a]. Besides duration, the harmonic function and a modulation indicator are also provided as parameters. Additionally, they allow the reuse of node identities in multiple places on the RHS, which elegantly solves the problem of synchronizing repeating sequences. PTGGs are used in the composition tool *Kulitta* [Qui14, Qui15] to generate harmonic progressions, based on production probabilities that were learned from existing music. Melkonian [Mel19] later extended them to melody and rhythm generation and generalized the harmony generation using a Schenkerian approach. They demonstrated this by encoding various grammars from musicologist literature, including the context-free variant of the Steedman grammar [Ste96] and tabla syntax [BK92b].

PTGGs and *Euterpea*, a music generation library by the same authors, as well as the extensions by Melkonian, were implemented in Haskell [HQSWC15, HQ18, Mel19], showcasing the effective use of functional programming for defining the rules of a generative grammar. This is preceded by works on a music notation framework [HMGW96] and algebraic treatment of temporal media [Hud04, HJ14], in which the authors discuss various functional techniques for algorithmic music composition and representation. Functional programming has further been used for modelling harmony [MdH11] and melody harmonization [KMDH13].

We move on to a work by Tanaka and Furukawa [TF12], who tackle polyphony on the grammar level, rather than as a post-processing step. They model polyphonic music as a list of voices, where every voice is a list of notes, and present two models for replacement. The naive model applies a rule at a single point in time and replaces the selected time span in all voices. In the asynchronous model replacement happens at different points in time in each voice. Rules are inferred using a genetic algorithm and matching happens only if the rule application preserves certain constraints. While the generated output is polyphonic, rules still operate on a sequential voice model, replacing one monophonic sequence with another, usually longer one.

Giraud and Staworko [GS15] used context-free parametric grammars to model Bach inventions. They use short motifs as parameters for a simple pattern language and demonstrate how an equivalent non-parametric grammar is more verbose. While the motifs are not generated by the grammar, the ability to use sequences of notes as parameters is a feature we also implemented for the grammar developed for this thesis. Additionally, they propose a distance metric for computing how closely a particular derivation tree matches a piece.

Other types of grammars have been used for musical applications. Petitjean [Pet12] used

MetaGrammars for declaratively constructing a lexicon of chords. Some authors have explored categorial grammars for music analysis [GW13], as well as generation [You17]. The latter used a vocabulary of domain functions, such as transposition and augmentation, higher-order combinators, such as list mappings and Cartesian products, and a hierarchy of musical types, such as rhythms, pitches, and scales. By adhering to the type relations implied by the type constraints on the input and output of the functions, the system automatically combines these elements into working programs, which in turn generate melodies.

2.2.2 Grammars in Graphics

In the broader field of content creation with generative grammars, the music domain might no longer rank as highly in popularity as it once had. At least from a commercial perspective, procedurally generated graphics seem to enjoy much broader application, considering how infeasible it would be to generate the effects and environments used in modern video games and CGI-heavy cinematography without significant amounts of automaton. The relationship between graphics and music has been of some interest in algorithmic music research, primarily for approaches based on self-similarity, such as L-systems, which can be interpreted graphically and musically. Our work has another possible relation to graphics: the treatment of time as a spatial dimension and, generally, the interpretation of symbolic music as points on a multi-layered lattice.

Split grammars, proposed by Wonka et al. [WWSR03] and preceded by the work on shape and set grammars by Stiny [Sti80, Sti82], were a significant inspiration for this thesis. In a split grammar, shapes are generated by recursively dividing spatial volumes, similar to the splitting of the time dimension that is used in many works on algorithmic composition. The initial split grammar later evolved into the popular *CGA Shape* grammar [MWH$^+$06] and its successor *CGA++* [SM15]. The latter introduces shapes as first-class citizens, allowing operations on generations of sub-grammars, which is a feature we partially implemented for this thesis. Also, compare Giraud and Staworko[GS15] in the music domain, who pass generated motifs as parameters. Various papers have been written on the generation of building facades [ZXJ$^+$13, IMAW15, JCS16], a problem that has some overlap with the synchronization of voices in a polyphonic piece.

2.3 Comparison with our approach

In the previous sections we presented the state of the art in the field of algorithmic composition, with a focus on grammar-based solutions. Now we will take a preliminary look at what separates this work from others and where we have drawn inspiration. Since the concrete goal of this thesis is the generation of polyphonic pieces, the treatment of the time dimension was a primary concern. The majority of existing grammar-based solutions do not model polyphony on the grammar level. Their representation of time is purely sequential, closely mirroring the implicit structure of the sentence. This is of course the simplest, most intuitive approach, if one wants to model melodies, rhythms, and harmonic progressions, neither of which require parallelism in the time dimension. One exception is Tanaka and Furukawa [TF12], who do use a polyphonic model to combine multiple voices, but the voices within are still sequential.

The need for a more flexible model of time lead us to split-grammars, which were the initially inspired this thesis. In fact, many generative models of music use some form of recursive temporal division; although, our own split operation might be more expressive than what we have seen in other works. So, it is not the eponymous *split* operation that makes us draw this parallel. Rather, it is the association of objects with an independent spatial or (in the case of music) temporal scope, that not only describes the size of an object, but also its absolute position. This allows us to freely alternate between parallel and sequential placement in the time dimension, move or resize notes without changing the overall structure, and even sample voices that were generated in earlier passes. The last point is facilitated by the ability of running isolated derivation passes and reusing the results, which has been used by Schwarz and Müller [SM15] for graphics, and to a limited degree in music by Giraud and Staworko [GS15].

Another aspect that makes our work unique is the ubiquitous use of custom attributes. Most of the existing grammars for composition are prescribe the types of information that can be propagated through the derivation tree. As a consequence, such grammars are limited to a particular musicological theory or composition task. In our system the grammar author is not bound by any particular semantics beyond the hard limits set by the derivation algorithm itself. We try to leave the choice of a domain model up to the user and instead supply them with a set of useful operators and output mappings to compensate for the increased generality.

Finally, our domain-specific language is hosted in a general purpose programming language and users can develop *ad hoc* extensions, giving them even more freedom in defining custom semantics. Our method of defining rules as functions is very similar to that of Quick and Hudak [QH13a], who also host their system within a general purpose functional language (Haskell). On the topic of embedded DSLs, consider the early survey on computer-aided composition by Loy and Abott [LA85], in which they describe a divide between programs with predefined interfaces and those with a more flexible programming interface, where the former tend to evolve the latter, due to incredibly varied requirements. This supports an argument by Moorer [Roa82], in which he states that music composition requires general purpose computing and consequently, once composers are sufficiently familiar with programming, they will require the power of a general purpose language. With this we conclude our discussion of related works and move on to the next chapter, in which we define the theory and various abstract features of our musical grammar definition language.

CHAPTER 3

A Functional Framework for Context-Free Grammars

Many phenomena can be understood in terms of formal language theory, including computer programs [Sip97], the way plants grow [PL90], collections of virtual 3D objects [MWH$^+$06], and certain aspects of music, as we have seen in Chapter 2. We can think of a formal language as a set of sentences, each consisting of an ordered sequence of words from a vocabulary. Assuming our vocabulary contains English words, we can form English sentences. Assuming our vocabulary contains musical notes, we can form melodies, or even complete scores. Instead of a *sentence*, *word*, and *vocabulary* metaphor, some authors prefer *word*, *letter*, and *alphabet*. I personally like *sentence* because of the connection between sentence structure and grammar in natural languages. Since *word* is used ambiguously, we will use the more abstract term *symbol* (later *entity*) to refer to the parts of a formal sentence.

In the hypothetical language of *minuets in G major* over the vocabulary of musical notes, the score of *Minuet in G Major, BWV Anh. 114 by Petzold* (formerly attributed to Johann Sebastian Bach) is a valid sentence. The score of *Gymnopédie No.1 by Eric Satie* is not a valid sentence since it is neither a minuet, nor written in G major. One could define a formal language by exhaustively enumerating all of its sentences, ignoring for a moment that this would entail the incredible task of composing every possible minuet. Yet, if we think of natural languages such as English, we can tell whether a sequence of words is valid without knowing every possible sentence. There is an underlying pattern, a set of structural constraints, which we can use to analyse almost any sequence of words and judge it as a valid sentence, or reject it as incorrect. These are the rules of the language.

Formal grammars are a way of defining formal languages in terms of rules. A formal rule for the English language could say that a **simple-sentence** may consists of a **subject**, which stands for any subject phrase, followed by a **verb**, standing for any verb phrase. When we read a subject phrase followed by a verb phrase, we can infer that the sequence of words must be a **simple-sentence**, at least according to that rule. Similarly, in music we could say that a **C-major-triad** is a **C**, **E**, and **G** note played simultaneously or in short succession, and an **authentic-cadence** may be a **C-major-triad**, preceded by a **G7-chord** consisting of the notes **G**, **B**, **D**, and **F**. Using these rules we can decide whether a particular piece contains an **authentic-cadence**. Notice how we did not precisely specify what a **subject** or **verb** is, nor did we specify how long **C**, **E**, or **G** must be, or their absolute tone height. Just as **C-major-triad** can be seen as a placeholder for **C**, **E**, and **G**, the latter three can be seen as abstract placeholders for concrete pitches. **C** could be a placeholder for **C1**, **C2**, **C3**, up to **C8**, which in turn could be placeholders for *the physical phenomenon of air pressure caused by vibrations at various multiples of 32.7 Hz*. It is exactly these placeholder relations which are expressed by the rules of a formal grammar.

This chapter is a step by step construction of the system we developed, with a very strong focus on the meta-language, rather than the domain-specific, musical perspective. While significant work went into the selection and integration of the various meta-level features, we cannot point to any particular aspect that seems sufficiently innovative to claim as our own, except for what was already mentioned in Section 2.3. However, the specific combination of features and their integration in a functional framework is not something we have seen before, thus it makes an important part of the contribution of this thesis. In this chapter, we define the semantic and syntactic foundation on which we will later base the domain-specific discussion.

3.1 Basics

Definitions: T, N, S, P, V In this thesis we build on the definition of a formal grammar established by Chomsky [Cho56]. Two components were already mentioned in this chapter's introduction: symbols and rules. A formal grammar $\langle T, N, S, P \rangle$ is defined by a set of terminal symbols T, a set of non-terminal symbols N, a starting symbol $S \in N$, and a set of production rules P, which we will address in Section 3.1.1. The two sets N and T are disjoint, and their union forms the vocabulary V. Only terminals may be found in the sentence. Non-terminals, on the other hand, are what we earlier referred to as *placeholders*, symbols that stand for other symbols. The starting symbol S, also known as the axiom, is a dedicated placeholder and can be seen as the polar opposite of the sentence. While a sentence contains no placeholders and can be said to be maximally concrete, the axiom is the most general placeholder and represents the highest level of abstraction in our language. Every sentence can be reduced to the axiom and in turn the axiom can stand for any sentence.

Example: Basics

In this example we will define the vocabulary for a simple musical grammar. We will use the seven pitch classes of the C major scale as our set of terminals T. Consequently, a sentence would be some sequence of notes in the C major scale. We could just as well use a set of absolute pitches (**A1**, **B1**, **C2**, **D2**, etc.) or use all pitches of the chromatic scale. The choice is rather arbitrary and depends on who or what may need to interpret the sentences of our language. We will use the non-terminals **G7-chord** and **C-major-triad** to build an intermediate abstraction layer on top of the pitch classes. The final non-terminal **authentic-cadence** is used as the axiom S. Again, the boundary is drawn arbitrarily, and we could easily imagine higher levels beyond a mere cadence, such as a **measure**, **phrase**, or whole **minuet**.

In this chapter we will use a very literal interpretation of generative grammars, where the order of the symbols in a sentence corresponds to their organization in the time dimension. Since many musical aspects are inherently sequential—think of melodies, rhythm, or harmonic progression—this is a very intuitive and commonly used approach. Yet, the sequential nature of the sentence is not an inherent limitation, which is one of the core premises of this work. As a metaphor, think of a cooking recipe: a list of ingredients does not mean that we must eat one after another. Instead, a cook will interpret the recipe and prepare a meal; the structure of the recipe is not apparent in the final result.

$$\begin{aligned} T &\coloneqq \{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}, \mathbf{F}, \mathbf{G}\} \\ N &\coloneqq \{\textbf{authentic-cadence}, \textbf{G7-chord}, \textbf{C-major-triad}\} \\ S &\coloneqq \textbf{authentic-cadence} \end{aligned}$$

3.1.1 Rules

A random sequence of notes from the C major scale may, with some luck, sound pleasant to us, but only some sequences can be described as a C major triad, and it is even more unlikely that it will contain an authentic cadence. Rules define the relations between symbols in such a way that we can move between levels of abstraction, finding cadences in sequences of notes, or generating sequences of notes that have a cadence.

Definitions: $P, X^*, \longrightarrow, [\ldots]$ Each production rule in P describes, in its most general form, a replacement of a sequence of non-terminal symbols with a sequence of symbols from our vocabulary. We can write a rule concisely with an arrow: $LHS \longrightarrow RHS$. The sequence that is replaced is described by the LHS (left-hand side). The sequence that replaces it is described by the RHS (right-hand side). Separated by a colon, we may specify an optional label that identifies the rule. For example, the rule $l : x \longrightarrow y$ has the label l. A very simple way of defining a rule is to list the symbols on both sides of the arrow in the form $N^* \longrightarrow V^*$, where for any set X, X^* is the set of arbitrary length sequences of its elements. For example, $\{x\}^*$ is equivalent to the set $\{[], [x], [x, x], [x, x, x], \ldots\}$. Note the use of the bracket notation $[\ldots]$, for which we will later define some additional semantics. For example, we do not differentiate between a plain element x and the singleton sequence $[x]$, and consider nested sequences such as $[x, [x, [x]]]$ as equivalent to the corresponding flat sequence $[x, x, x]$.

Example: Rules

We can now formally express the relationship between the symbols of our vocabulary with three rules $P \coloneqq \{p_1, p_2, p_3\}$. Rule p_1 establishes a two-chord progression for the cadence. Rules p_2 and p_3 specify the notes for the two chords. A chord progression is sequential in the time dimension, but the notes of a chord are usually played at the same time. In this chapter we will glance over this important distinction and may either assume an implicit grouping, or interpret the note sequence as an *arpeggio*, meaning the chord notes are played in short succession; the choice is up to the reader. A detailed discussion of our handling of the time dimension can be found in Chapter 4. Of course, describing the structure of an authentic cadence lacks artistic expression, but using a familiar and clearly defined musical pattern makes the validity of the abstraction immediately obvious.

$$
\begin{aligned}
p_1 :&\quad \textbf{authentic-cadence} \longrightarrow [\textbf{G7-chord}, \textbf{C-major-triad}] \\
p_2 :&\quad \textbf{G7-chord} \longrightarrow [\textbf{G}, \textbf{B}, \textbf{D}, \textbf{F}] \\
p_3 :&\quad \textbf{C-major-triad} \longrightarrow [\textbf{C}, \textbf{E}, \textbf{G}]
\end{aligned}
$$

3.1.2 Derivation

The idea that rules replace sequences of symbols with other sequences is very general. Without further restrictions, such grammars can be Turing complete. The rules in the previous example did not fully exploit this potential since each of them used exactly one non-terminal on the LHS. This means our grammar is at least type-2 in the hierarchy of grammar types defined by Chomsky. Each type puts some constraints on the expressive power of the LHS and RHS, with *unrestricted* grammars (type-0) being the least restrictive, and *regular* grammars (type-3) the most restrictive variants. Lower numbered types include all capabilities of the higher types, but the computational and perceived complexity increases with higher generality.

The remaining grammar types are the *context-sensitive* (type-1) and the *context-free* grammars (type-2), the latter of which will serve as the fundamental theoretic model for the system developed in this thesis. Type-2 grammars have been used by various researchers for procedural generation, arguably because they strike a good balance between power and simplicity. For example, Steedman's first grammar for Blues progressions was context-sensitive [Ste84], but they were later able to reformulate it as a context-free grammar [Ste96]. A general discussion about different grammar types for music generation can be found in the early survey by Roads and Wieneke [RW79]. From here on, all theoretic models and examples assume that the underlying grammar is context-free.

In context-free grammars a rule replaces exactly one non-terminal with zero or more symbols, which means the form of the rule is restricted to $N \longrightarrow V^*$. If one of the replacement symbols is itself a non-terminal, it can be replaced again. This recursive one-to-many relationship is closely modelled by a tree structure, where each rule application defines edges between a replaced non-terminal (*parent* or *predecessor*), and a sequence of replacements (*children* or *successors*). The root of the tree is the axiom S, the internal nodes are non-terminals N, and the leaves are terminals T. The replacement process is also called *derivation*, and the resulting tree structure is called the *derivation tree*. Since the derivation does not rely on context, every subtree can be seen as an isolated derivation, where the root of the subtree is the axiom of a sub-grammar, and the leaves are its terminals. Consequently, any symbol in the vocabulary may be used as a starting symbol, resulting in a sentence of the language defined by the particular sub-grammar.

The derivation algorithm constructs a derivation tree incrementally, starting from the axiom. For each non-terminal leaf node a rule will be selected, where the LHS *matches* the symbol. When the rule is applied, the output nodes are added as children to the input node. The intermediate sequence of leaves is also called a *sentential from*, and a sentential form that contains no placeholders is a sentence. Assuming every non-terminal in the sentential form must be replaced before the derivation terminates, we must assume that there exists a rule that matches it. If there is no such rule, the grammar is invalid. Any set of rules from a valid grammar therefore implicitly defines non-terminals as those symbols that may be matched by an LHS, and the terminals as those symbols that are not matched by any rule. Most grammar examples in this thesis will not explicitly define terminals and non-terminals and assume they can be inferred from the rules. Instead of N and T we will mostly use the combined set V in our definitions. The distinction is primarily necessary for the definition of type-1 and type-3 grammars.

Example: Derivation

We will use the definitions of V, S, and P from the prior examples and generate a sentence by going through the derivation algorithm step by step. Table 3.1 shows the numbered steps, and the resulting derivation tree is visualized in Figure 3.1. At step 0 no rule is yet applied, and the sentential form contains only the axiom **authentic-cadence**. In step 1 we replace the axiom with two new non-terminals according to rule p_1. In step 2 we could either replace **G7-chord** or **C-major-triad**, but arbitrarily chose to replace **G7-chord** with rule p_2. In context-free grammars it does not matter which non-terminals are replaced first, so the order of steps 2 and 3 has no effect on the result.

step	rule	sentential form
0	–	**authentic-cadence**
1	p_1	[**G7-chord**, **C-major-triad**]
2	p_2	[[**G**, **B**, **D**, **F**], **C-major-triad**]
3	p_3	[[**G**, **B**, **D**, **F**], [**C**, **E**, **G**]]

Table 3.1: Derivation of a context-free grammar as a sequence of numbered steps. The nested square brackets in the sentential form illustrate the tree structure, but serve no further purpose.

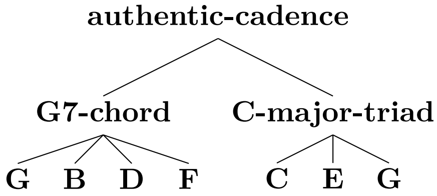


Figure 3.1: Derivation of a context-free grammar as a tree.

3.2 Parameters

If we wanted to generate the authentic cadence from Section 3.1.2 for D major instead of C major, we would have to add three new rules: one for the new chord pair and two for the new chords. Assuming we wanted to generate a cadence in any of the twelve major keys of the chromatic scale, the number of rules would increase by a factor of twelve. This rightfully seems excessive for such a simple pattern. A music theorist, given a key, can come up with the major triad and seventh chord of the dominant harmony without having to learn all possibilities by heart. They know that the dominant harmony is one fifth above the tonic, that a seventh chord is a major triad together with a minor seventh, and that a major triad consists of the tonic, major third, and fifth. Rather than absolute pitch values, the theorist thinks in terms of relative intervals. Yet, so far the vocabulary contains only atomic labels and pitch is not something that can be manipulated independently.

A grammar with atomic entities, such as the one we have constructed in the previous section, can be seen as a special case of a *parametric grammar* with a single parameter. We used that parameter to encode multiple separate concepts. For example, the **G7-chord** combines a **7-chord** with a tonic pitch G. Similarly, the symbol **C** is a **note** with a fundamental pitch C. Separating the pitch from the semantic label enables us to independently consider these aspects when we match and replace symbols. Our vocabulary becomes a set of tuples of the form (x_1, x_2), where x_1 stands for a label, such as **7-chord** or **note**, and x_2 is the associated pitch value, such as G or C. For example, we can define a rule that matches on an arbitrary **7-chord** and use relative intervals to calculate a pitch for the four **note** symbols.

Definitions: V In general, the vocabulary of a parametric grammar with n parameters consists of elements from an n-dimensional space $V = X_1 \times \ldots \times X_n$. Since the term *symbol* implies some degree of indivisibility, we will use the term *entity* from here on, when referring to the elements of the vocabulary. Barring some early and primitive examples, most existing works use entities with multiple parameters. For example, Steedman [Ste96] and Rohrmeier [Roh07] use the key, harmonic function, and chord type as parameters. The grammars by Quick and Hudak [QH13b] have an explicit time parameter for rhythmic variation. A fixed set of parameters allows one to optimize the grammar definition language, and most authors use a compact glyph notation for their entities. For example, Quick and Hudak [QH13b] would write $III^{\frac{1}{4}}$ for a third with the duration of a quarter of the original time interval. We use a generic tuple notation, in order to highlight that the parameter space can be freely adapted to a particular task or model.

Example: Parameters

We will construct our new vocabulary using two parameters: label and pitch. The label is defined over a set of semantic labels L, similar to the symbols from the previous example, but without pitch information. So **C-major-triad** becomes **major-triad**, **G7-chord** becomes **7-chord**, and **note** will be used as a generic name for our terminals. As before, the label will be used for matching. The pitch parameter is defined over the set of chromatic pitch-classes $\mathcal{C} = \{A, Bb, B, C, \ldots, G, Ab\}$. In order to make the grammar independent of a particular key, we will only refer to the pitch by the variable x_2. For relative movement we define a set of pitch intervals $I = \{m3, M3, P4, P5, m7\}$ and a binary operation $+ : \mathcal{C} \times I \rightarrow \mathcal{C}$. As an example, the term $C + P5$ calculates the pitch class one perfect fifth above C, which is G. With these parameters and the + operator, we can now define a general authentic cadence generator. There is also a new abstraction in rule p_2, where we are able to reuse rule p_3 for the definition of the seventh chord. The derivation tree of this grammar is visualized in Figure 3.2.

$$\begin{aligned}
p_1 : \quad (\textbf{authentic-cadence}, x_2) &\longrightarrow [(\textbf{7-chord}, x_2 + P5), (\textbf{major-triad}, x_2)] \\
p_2 : \quad (\textbf{7-chord}, x_2) &\longrightarrow [(\textbf{major-triad}, x_2), (\textbf{note}, x_2 + m7)] \\
p_3 : \quad (\textbf{major-triad}, x_2) &\longrightarrow [(\textbf{note}, x_2), (\textbf{note}, x_2 + M3), (\textbf{note}, x_2 + P5)]
\end{aligned}$$

3.2.1 Parametric Matching

Already, the authentic cadence generator is becoming quite powerful, but there is still very little variation besides a change of pitch. In order to musically express our sentiment towards the progress so far, we could introduce a new parameter with two states: *happy* and *sad*. If the mood is happy, we want to resolve into a major triad as we did before. If the mood is sad, we want to generate a minor chord instead. There are at least two ways in which we can control the flow of the derivation based on a parameter. Given

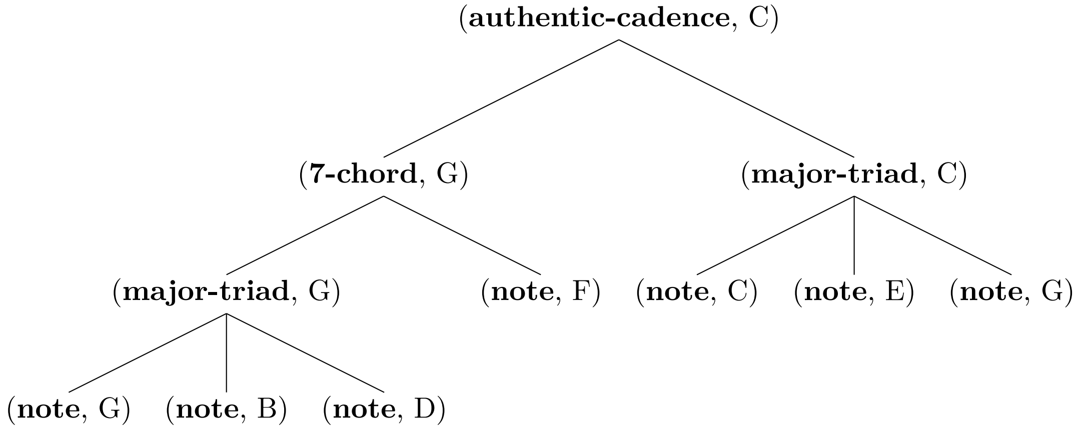


Figure 3.2: Derivation tree of a parametric grammar with two parameters. Since we use C for the second parameter of the root entity, the output is the same as in Section 3.1.2. Due to the reuse of the **major-triad** for the generation of the **7-chord**, the shape of the tree is slightly different compared to the non-parametric example.

an entity $v \in V$, a predicate $c : V \to \{\top, \bot\}$, and two right-hand side replacements $\alpha, \beta \in V \to V^*$, we want to apply $v \longrightarrow \alpha(v)$ if $c(v) = \top$, and $v \longrightarrow \beta(v)$ otherwise. A simple case differentiation is one possible solution:

$$v \longrightarrow \begin{cases} \alpha(v), & \text{if } c(v) = \top \\ \beta(v), & \text{otherwise} \end{cases}$$

While the above works, there is another way of testing entities, that keeps the RHS simple and utilizes the structural capabilities of a formal grammar. The purpose of the LHS is to describe possible replacement candidates. So, if replacement should only happen under a certain condition, we should test this in the LHS. If the test fails, we can reject the rule outright, without considering the RHS at all. Another advantage is that we do not have to provide an alternative branch since the derivation process will either move on to the next rule, or terminate for this entity. We have already used matching conditions implicitly. For an LHS such as $(\textbf{major-triad}, x_2)$ for rule p_3, we assumed that the derivation algorithm would interpret it as the condition $x_1 = \textbf{major-triad}$. Since we now also want other parameters to affect the matching behavior, we will make this condition explicit. The general scheme above can be rewritten with two rules, where $\neg c$ is the negated condition:

$$\begin{aligned} c(v) &\longrightarrow \alpha(v) \\ \neg c(v) &\longrightarrow \beta(v) \end{aligned}$$

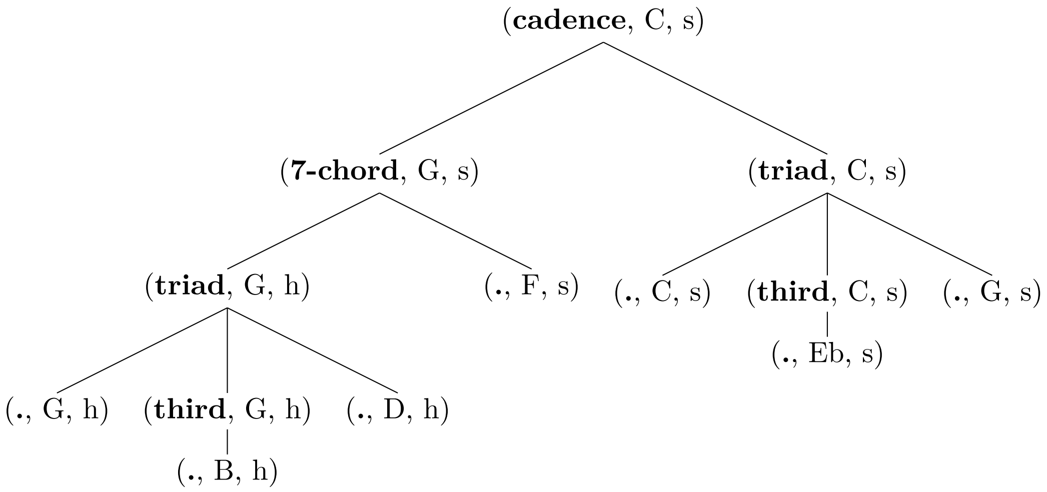


Figure 3.3: Derivation tree of a parametric grammar with three parameters (x_1, x_2, x_3). The two entities named **third** were matched by different rules and the pitch was increased by a major third if $x_3 = h$, and a minor third if $x_3 = s$. The **note** label for the leaves is abbreviated to "**.**" due to spatial constraints.

Example: Parametric Matching

We will now define a parametric cadence generator with a three-dimensional vocabulary $(x_1 \in L, x_2 \in \mathcal{C}, x_3 \in \{h, s\})$, where h stands for *happy*, and s stands for *sad*. In p_2 we set $x_3 = h$ because a dominant seventh is always a major chord. Everywhere else x_3 is passed down to the children unchanged. In rule p_3 the triad is generated with the new label **third**. In rules p_4 and p_5 we use an additional condition on x_3 to generate a major or minor third offset depending on the value of x_3. Note that we are assuming an implicit binding of the three variables x_1, x_2, and x_3. The derivation tree is visualized in Figure 3.3.

$$\begin{aligned}
p_1 :& \quad x_1 = \textbf{cadence} \longrightarrow [(\textbf{7-chord}, x_2 + P5, x_3), (\textbf{triad}, x_2, x_3)] \\
p_2 :& \quad x_1 = \textbf{7-chord} \longrightarrow [(\textbf{triad}, x_2, h), (\textbf{note}, x_2 + m7, x_3)] \\
p_3 :& \quad x_1 = \textbf{triad} \longrightarrow [(\textbf{note}, x_2, x_3), (\textbf{third}, x_2, x_3), (\textbf{note}, x_2 + P5, x_3)] \\
p_4 :& \quad x_1 = \textbf{third} \wedge x_3 = h \longrightarrow (\textbf{note}, x_2 + M3, x_3) \\
p_5 :& \quad x_1 = \textbf{third} \wedge x_3 = s \longrightarrow (\textbf{note}, x_2 + m3, x_3)
\end{aligned}$$

3.3 Attributes

At this point we already designed three different versions of a simple cadence generator, one with just labels, one with an additional pitch parameter, and one with an additional mood parameter. This workflow of iterating on an existing model, changing, adding, generalizing features, is quite natural when inventing musical grammars. From an artistic point of view, there are simply no hard criteria for the correctness of the results, and our decisions will often depend on system feedback. Form a technical standpoint, it can be very difficult to predict the results in advance, especially once non-deterministic features come into play, and one must make incremental changes to understand the effect. So, if continuous modifications of our grammar are guaranteed in practice, the underlying framework should make changes as painless as possible.

So far, changing the value space of a parameter is straight-forward. We did this multiple times already with the label parameter x_1, when we added new labels or discarded obsolete ones. In comparison, adding and removing an entire parameter is far from painless since it requires us to update each parameter tuple. In case of the mood parameter x_3, the problem became especially evident because it is only used once in the RHS and twice in the LHS. In general, a rule will depend on a subset of the parameters and any other parameters will be passed to the children unchanged. Making such changes would be easier if we could decouple the rule definition from the exact structure of the entity space, allowing the rules to only access what they need. This leads us to the concept of *attributes*, which were invented by Wegner [Knu90].

Definitions: K, X_k, k_{get}, k_{set} We define the set $K = \{k_1, \ldots, k_n\}$ of parameter identifiers and associate them with the dimensions of our n-dimensional parameter space $V = X_{k_1} \times \ldots \times X_{k_n}$. For every $k \in K$ we define a getter function $k_{get} : V \to X_k$ and a setter function $k_{set} : X_k \to (V \to V)$. For a particular $k \in K$ and $\forall l \in K \setminus \{k\}$ the functions are defined in such a way that for any $v \in V$ and $x_k \in X_k$ the following equivalencies are satisfied:

$$\begin{aligned} &f := k_{set}(x_k) \\ &k_{get}(f(v)) = x_k \\ &l_{get}(f(v)) = l_{get}(v) \end{aligned}$$

This just means that a setter only changes the value of one particular parameter, working as an identity function for the rest, while a getter retrieves the latest value. If $k_{get}(v) = x_k$ for some entity $v \in V$, we say v has or defines an *attribute* with the key or name k and a value x_k. k_{set} and k_{get} are also called the *accessors* of the attribute.

Definitions: $k, \langle \ldots \rangle$ Note how k_{get} and $k_{set}(x_k)$ are both functions over the domain V. We call such functions *expressions*, and in Section 3.4 we will discuss them in detail. We can simplify our syntax by assuming that expressions are evaluated implicitly on the current entity $v \in V$. So, on both sides of a rule we may write k_{get} or just k instead of $k_{get}(v)$. On the right-hand side we will write $k_{set}(x_k)$ instead of $(k_{set}(x_k))(v)$ and $\langle k_{set}(x_k), \ldots, l_{set}(x_l) \rangle$ for setting multiple attributes. The exact semantics of the angle bracket notation $\langle \ldots \rangle$ will be defined in Section 3.4.2, but it is essentially a higher-order expression that applies the first expression within the brackets to the current entity, and each subsequent expression to the output of the prior one. Since the setter leaves all but one attribute untouched, attribute values will automatically propagate downwards in the derivation tree—from the parent to the child. This type of behaviour is known as *attribute inheritance*, and means that we do not have to pass invariant parameters explicitly.

Example: Attributes

In this example we will associate our three parameters (x_1, x_2, x_3) with three attribute names $K = \{label, pitch, mood\}$ with the same values as before. Using descriptive attribute names helps us understand the semantics of a rule at a glance. Since we must set the *label* attribute for every new entity, we have slightly optimized the syntax in these examples. Whenever a label $\mathbf{x} \in L$ appears within the RHS, it actually stands for $label_{set}(\mathbf{x})$. For *pitch* and *mood* we use the normal setter syntax. Now notice that by using attributes instead of positional parameters, rules p_1 and p_3 no longer depend on *mood* and will be unaffected by future changes to it. Compared to the previous example, the overall behavior of our grammar remains the same, and the structure of the derivation tree has not changed.

$$\begin{aligned}
p_1 : &\quad & label = \textbf{cadence} &\longrightarrow [\langle \textbf{7-chord}, pitch_{set}(pitch + P5)\rangle, \textbf{triad}] \\
p_2 : &\quad & label = \textbf{7-chord} &\longrightarrow [\langle \textbf{triad}, mood_{set}(h)\rangle, \langle \textbf{note}, pitch_{set}(pitch + m7)\rangle] \\
p_3 : &\quad & label = \textbf{triad} &\longrightarrow [\textbf{note}, \textbf{third}, \langle \textbf{note}, pitch_{set}(pitch + P5)\rangle] \\
p_4 : &\quad & label = \textbf{third} \wedge mood = h &\longrightarrow \langle \textbf{note}, pitch_{set}(pitch + M3)\rangle \\
p_5 : &\quad & label = \textbf{third} \wedge mood = s &\longrightarrow \langle \textbf{note}, pitch_{set}(pitch + m3)\rangle
\end{aligned}$$

3.3.1 Optional Attributes

For the latest example, we defined initial values for all parameters. This is necessary since we must select a particular element from V as the axiom. Yet, our definition of attributes does not inherently require that every entity defines every parameter, as long as we do not use its getter before we used its setter. This can be useful in practice when we model more complex structures with multiple semantic levels. For example, we could encode the chord type as a special *chord* attribute that is determined once the derivation reaches the chord level of the piece. Under our current definition every entity, including the axiom, would have to define the *chord* attribute, but semantically it does not make sense to say a cadence is a type of chord.

Definitions: $\emptyset$ With optional attributes we formalize the situation where a particular dimension of an entity cannot be meaningfully defined. An attribute k is optional when its value space contains the special value $\emptyset$. When the attribute value is $\emptyset$, we say the attribute is *undefined*, otherwise we say it is defined or, more informally, that it exists. Unless a value is explicitly specified, we generally assume that an optional attribute is undefined for the axiom. We further define four new accessors:

$$
\begin{aligned}
&k_{has} : V \to \{\top, \bot\} : v \mapsto k_{get}(v) \neq \emptyset \\
&k_{def} : X_k \to (V \to V) : x_k \mapsto v \mapsto \begin{cases} v, & \text{if } k_{has}(v) \\ (k_{set}(x_k))(v), & \text{otherwise} \end{cases} \\
&k_{or} : X_k \to (V \to X_k) : x_k \mapsto v \mapsto \begin{cases} k_{get}(v), & \text{if } k_{has}(v) \\ x_k, & \text{otherwise} \end{cases} \\
&k_{tag} : V \to V : k_{set}(\top)
\end{aligned}
$$

Definitions: $k_{has}, k_{or}, k_{def}, k_{tag}$ Again, k_{has}, $k_{or}(x_k)$, $k_{def}(x_k)$, and k_{tag} are expressions and will be implicitly applied to the current entity. k_{has} is used to test whether an entity defines a particular attribute. Within the LHS, we may also just write k instead of k_{has}. k_{def} works like k_{set} if k is undefined, otherwise it has no effect. k_{or} retrieves the current value if k is defined, otherwise it retrieves the specified fallback value. Finally, k_{tag} can only be used for attributes that include $\top$ in their value space. We will use $\{\top\}$ as the value space for optional attributes that do not need a particular value.

Example: Optional Attributes

In this example we assume *pitch* and *mood* are optional. In p_1 we use the fallback value C for computing the fifth and initialize it for the triad. Rules p_2 to p_5 all depend on *pitch* and check for its existence in the LHS. p_4 and p_5 check the *mood* attribute using an equality, which will fail if its value is $\emptyset$. For this reason, we now guarantee that *mood* is defined in p_3 using the fallback value h.

$$
\begin{aligned}
p_1 :&\quad & \mathit{label} = \textbf{cadence} &\longrightarrow [\langle \textbf{7-chord}, \mathit{pitch}_{set}(\mathit{pitch}_{or}(C) + P5)\rangle, \\
&&& \langle \textbf{triad}, \mathit{pitch}_{def}(C)\rangle] \\
p_2 :&& \mathit{label} = \textbf{7-chord} \wedge \mathit{pitch} &\longrightarrow [\langle \textbf{triad}, \mathit{mood}_{set}(h)\rangle, \\
&&& \langle \textbf{note}, \mathit{pitch}_{set}(\mathit{pitch} + m7)\rangle] \\
p_3 :&& \mathit{label} = \textbf{triad} \wedge \mathit{pitch} &\longrightarrow [\textbf{note}, \langle \textbf{third}, \mathit{mood}_{def}(h)\rangle, \\
&&& \langle \textbf{note}, \mathit{pitch}_{set}(\mathit{pitch} + P5)\rangle] \\
p_4 :&& \mathit{label} = \textbf{third} \wedge \mathit{pitch} \wedge \mathit{mood} = h &\longrightarrow \langle \textbf{note}, \mathit{pitch}_{set}(\mathit{pitch} + M3)\rangle \\
p_5 :&& \mathit{label} = \textbf{third} \wedge \mathit{pitch} \wedge \mathit{mood} = s &\longrightarrow \langle \textbf{note}, \mathit{pitch}_{set}(\mathit{pitch} + m3)\rangle
\end{aligned}
$$

3.3.2 Goal Fulfillment

The introduction of optional attributes ushers in a philosophical shift in our system. Entities are no longer structurally the same, but may represent different types of musical objects depending on the available attributes. Up to now we differentiated between entities based on the nominal *label* attribute, which has very limited semantic meaning. For example, we could encode the labels **7-chord** and **triad** as a *chord* attribute that has the particular chord type as its value. Semantically, this would be a richer and arguably cleaner model of our domain. A rule that generates a chord could then match on the *chord* attribute, rather than testing the overloaded *label* attribute. The problem is that this *chord* attribute will be inherited to the children, and the same rule would match again and again. We avoid this with *label* by only matching on particular values and then changing it for every child. For example, we must set **note** as a dummy label for our terminals, even though we never actually use it for matching. Ideally, we want structural matching, but without risking an infinite recursion.

When the derivation algorithm encounters an entity with the *chord* attribute, it should select a rule that can make a chord, for example, by replacing it with the chord notes. We could also say *chord* is a *goal* for the entity, and a rule that specifies it in the LHS represents a claim, of sorts, that the rule is able to *fulfill* this goal. The idea is that a rule can only be applied if it can fulfill at least one goal. Once the rule has been applied, the goal can be considered fulfilled, and the rule will not be applied to the same entity again. There are multiple possible ways to integrate goal fulfillment mechanism into our system. Do we declare goals on the entity, or do they depend on the rule? Are goals implicitly or explicitly marked as fulfilled? How can we allow the repeated fulfillment of the same goal for deliberate recursion? The particular solution we describe below is just one possibility and not necessarily optimal by every metric.

Definitions: $G, k^!, k_!$ In the LHS of a rule $p \in P$, we may write $k_!$ to add k to the set $G(p) \subseteq K$ of goals that p can fulfill. For example, if the LHS of p is $k_! \wedge l$, then $G(p) = \{k\}$. The logical operators are irrelevant at this stage. The set of $G(v) \subseteq K$ is the set of unfulfilled goals on the entity $v \in V$. When we want to test whether some rule p matches entity v, we first intersect $G(p)$ with $G(v)$. If the intersection is empty, no goal can be fulfilled, and the rule does not match. Otherwise, if at least one goal can be fulfilled *and* the condition is satisfied *and* the rule is selected, all goals in the intersection are marked as fulfilled. Within the RHS we can reset the fulfillment state by writing $k^!$, which we can use to cause deliberate recursion. Analogously, we can write $k_!$ to manually fulfill k. Automatic fulfillment happens before the RHS is applied since we may want to immediately change the fulfillment state.

With this model we can decide on a per-rule basis which attributes are treated as goals and which are used only for conditional matching. This is useful because sometimes attributes should become goals at a later point, and some attributes should not be fulfilled at all. Consider the *pitch* and *mood* attributes, which are potentially propagated down from the axiom. Goal semantics do not seem to be meaningful for these two, as they are better understood as shared parameters of various musical abstraction levels. Regarding deliberate recursion, resetting the fulfillment state in the RHS rather than in the LHS is more expressive since it allows us to specify both the recursion and the conditional termination within a single rule.

Example: Goal Fulfillment

In this example we will use goal fulfillment instead of a nominal *label* attribute for matching. We remove the *label* attribute altogether and add the valueless *cadence* and *third* attributes and a *chord* attribute with the values $\{7, \mathit{triad}\}$. We fulfill exactly one goal in each LHS, but we could potentially specify an arbitrary number, as long as that number is not zero. The **note** label does not need a replacement because we no longer need to explicitly mark the terminals, which leads to the degenerated expression $\langle\rangle$ for the chord root in rule p_3. The expression $\langle\rangle$ works as an identity function and replaces the input entity with itself. The use of $\mathit{chord}^!$ in rule p_2 immediately resets the fulfillment state, so that p_3 can fulfill it again. Since we use two different rules and p_3 does not reset the goal again, there will be no infinite recursion. As intended, we no longer need to use a label for the terminals. The *pitch* attribute seems slightly out of place since we use it in almost every rule. We will discuss this phenomenon in the next section.

$$
\begin{aligned}
p_1 :\quad && \mathit{cadence}_! &\longrightarrow [\langle \mathit{chord}_{\mathit{set}}(7), \mathit{pitch}_{\mathit{set}}(\mathit{pitch}_{\mathit{or}}(C) + P5)\rangle,\\
&& &\qquad \langle \mathit{chord}_{\mathit{set}}(\mathit{triad}), \mathit{pitch}_{\mathit{def}}(C)\rangle]\\
p_2 :\quad && \mathit{chord}_! = 7 \wedge \mathit{pitch} &\longrightarrow [\langle \mathit{chord}^!_{\mathit{set}}(\mathit{triad}), \mathit{mood}_{\mathit{set}}(h)\rangle,\\
&& &\qquad \mathit{pitch}_{\mathit{set}}(\mathit{pitch} + m7)]\\
p_3 :\quad && \mathit{chord}_! = \mathit{triad} \wedge \mathit{pitch} &\longrightarrow [\langle\rangle, \langle \mathit{third}_{\mathit{tag}}, \mathit{mood}_{\mathit{def}}(h)\rangle, \mathit{pitch}_{\mathit{set}}(\mathit{pitch} + P5)]\\
p_4 :\quad && \mathit{third}_! \wedge \mathit{mood} = h \wedge \mathit{pitch} &\longrightarrow \mathit{pitch}_{\mathit{set}}(\mathit{pitch} + M3)\\
p_5 :\quad && \mathit{third}_! \wedge \mathit{mood} = s \wedge \mathit{pitch} &\longrightarrow \mathit{pitch}_{\mathit{set}}(\mathit{pitch} + m3)
\end{aligned}
$$

3.3.3 Attribute Semantics

The *pitch* attribute is overloaded with four different meanings: at first it describes the tonic of the cadence, then the root for a chord, then an alterable note for the third, and for the terminals it is simply the fundamental frequency of the note. We could use different attributes for each of these meanings, but they seem to coexist quite naturally within the *pitch* attribute. There are various reasons why the concept of pitch works well across abstraction levels, not least because it is a core dimension of music, and many musical objects can be reduced to or derived from a fundamental pitch. Anyway, *pitch* is queried and written quite often in our grammar and whenever this happens we need to guarantee its existence, either by checking for it in the LHS or providing a default value. At this point, working with the grammar is still rather tedious, reminiscent of programming in a very low-level language. We have built a custom model of a cadence by defining attributes, matching conditions, and primitive mutations, but the behavior of the derivation algorithm is not inherently musical, except for the $+$ operator for pitches and intervals. We will soon introduce additional abstract features, but this will only get us so far. Domain-specific features are essential, otherwise we will not end up with a system for music generation, but with a generic programming language. Still, flexibility is important too, considering Moorer's stance on the value of general purpose programming for composition [Roa82].

So, how will we define domain-specific features in a way that is unobtrusive and does not impose unnecessary limitations? For one, we will provide these features as composable functional models. These functions take certain musical structures as input, and generate new structures as their output. The structures are defined in terms of attributes, some of which the function will consume, others it will add or replace. For example, an instrument model that synthesizes a sound may consume a *pitch*, *time*, and *volume* attribute and generate a sequence of audio samples. Ideally, the semantics of these attributes overlap with those used by other models. If multiple models assume identical semantics for a particular attribute, we call it an attribute with *canonical semantics*. The reason why the *pitch* attribute works well is that it is part of a common interface that is shared by all of our rules. If each rule defined its own identifier for the role of the *pitch* attribute, we would have to do some renaming between representations. Further, if we commit to the semantics of the *pitch* attribute, we can define other functions that depend on them. For example, we could reformulate the $+$ operator for pitches and intervals as a domain-specific $\mathit{pitch}_{+}(x)$ function, which adds the interval x to the current pitch value in a very concise way.

We will often attempt to find canonical semantics in order to simplify the interaction between the various domain-specific models that we employ. Canonical attributes are attributes that are associated with canonical semantics, though there is nothing special about these attributes in terms of syntax or behavior. In the context of our cadence generator the *pitch* and *mood* attributes could be called canonical since they transcend the boundaries of multiple rules. Ultimately, this classification is primarily useful to differentiate them from *ad hoc* attributes, that are specific to a particular example. In practice, the initialization of these attributes is an important consideration. For example, what happens if we use pitch_{+} on an entity that does not define *pitch*? We will usually avoid undefined behavior by providing fallback values, but raising an error, or at least a warning, are viable alternatives. In the future we may want to rely on a type system for recognizing such problems early.

Example: Attribute Semantics

In this example we commit to the *pitch* attribute as a fundamental component of our model. We will define the function $\mathit{pitch}_{+} : I \to (V \to V)$, which returns a setter expression via the mapping $x \mapsto \mathit{pitch}_{\mathit{set}}(\mathit{pitch}_{\mathit{or}}(C) + x)$. This allows us to streamline the notation for the pitch modification since the *pitch* getter is used implicitly with a guaranteed default value C. We also remove the existence check from the LHS, since our rules no longer require an explicit *pitch*. Obviously, we would still specify a pitch for the axiom, but the grammar is now at least correct without it, albeit C is an arbitrary choice.

$$\begin{aligned}
p_1 :&& \mathit{cadence}_{!} &\longrightarrow [\langle \mathit{chord}_{set}(7), \mathit{pitch}_{+}(P5)\rangle, \mathit{chord}_{set}(\mathit{triad})]\\
p_2 :&& \mathit{chord}_{!} = 7 &\longrightarrow [\langle \mathit{chord}^{!}_{set}(\mathit{triad}), \mathit{mood}_{set}(h)\rangle, \mathit{pitch}_{+}(m7)]\\
p_3 :&& \mathit{chord}_{!} = \mathit{triad} &\longrightarrow [\langle\rangle, \langle \mathit{third}_{tag}, \mathit{mood}_{def}(h)\rangle, \mathit{pitch}_{+}(P5)]\\
p_4 :&& \mathit{third}_{!} \wedge \mathit{mood} = h &\longrightarrow \mathit{pitch}_{+}(M3)\\
p_5 :&& \mathit{third}_{!} \wedge \mathit{mood} = s &\longrightarrow \mathit{pitch}_{+}(m3)
\end{aligned}$$

3.4 Expressions

The fundamental feature of a context-free derivation is a one-to-many relationship between entities. As a consequence, many components of our grammar are parametrized by a single entity. For example, given an input entity, the LHS returns the result of the matching condition, the RHS generates a sequence of replacement entities, the setter copies the entity and adds an attribute, and the getter retrieves the value of an attribute. Taking the predominance of this type of function into consideration, we generally try to control the flow of entities implicitly. We have used a declarative notation in the previous sections, where we avoided references to the input entity, but we have not yet provided an explanation of the mechanism. In this section we will find a unifying functional model for the features so far, which is naturally quite abstract, but will help us keep our definitions simple and our syntax declarative.

Definitions: Y We call a function of the form $V \to Y$, where V is the set of entities, an *expression* over the *constants* Y. An important subtype are expressions over sentences, which would be any function $V \to V^*$. For example, the RHS and setter expression are both sentence expressions, albeit an RHS may generate sentences of arbitrary length, while a setter generates exactly one entity. We have to differentiate between the setter function k_{set} and the setter expression $k_{set}(x_k)$ for some attribute k and value x_k. The former is a function that returns an expression, the latter *is* an expression. Expressions can be the result of a function, but they are especially useful as function arguments. Often, the value that we pass to the setter function depends on the input entity $v \in V$. For example, we may want to copy the value of an attribute l to attribute k by writing $k_{set}(l)$, where l is short for l_{get}.

A *parametric expression* is an expression that depends on a parameter, and we could define it as a function of the form $X \to (V \to Y)$, where X is the set of parameters. The setter function is one example since we defined it as $k_{set} : X_k \to (V \to V)$ for some attribute k with a value space X_k. Yet, using X_k as the domain of our setter function is inconvenient because unless we just want to set a constant $x_k \in X_k$, we must explicitly evaluate any expression $V \rightarrow X_k$. Instead, we define a parametric expression as $(V \rightarrow X) \rightarrow (V \rightarrow Y)$, which covers the general case, where we specify the parameter x in terms of the input entity v. This allows us to use $k_{set}(l)$ directly, where l is an expression $V \rightarrow X_k$. A constant x_k can be set via $k_{set}(v \mapsto x_k)$, where v is simply ignored. Note that the evaluation of an expression is always the responsibility of the receiving function. In the example our new setter function receives an expression l, and $(k_{set}(l))(v)$ will internally assign $l(v)$ to attribute k.

Definitions: Y_λ, λ For setting a constant we could either use our earlier setter function that was defined over X_k, or manually wrap every constant $x_k \in X_k$ in a dummy expression $v \mapsto x_k$, as shown above. Since neither solution is ideal we generalize the two concepts. Given a set of constants $Y = Y_\lambda^0$ as a basis, we recursively define the set of nth-order expressions $Y_\lambda^n = Y_\lambda^{n-1} \cup (V \rightarrow Y_\lambda^{n-1})$, where $n \in \mathbb{N}$ is a finite number. For example, the first-order expressions Y_λ^1 are equivalent to $Y \cup (V \rightarrow Y)$. Nth-order expressions include any chain of the form $V \rightarrow \ldots \rightarrow V \rightarrow Y$ with at most n occurrences of V. When the exact value of n is not important, which is the common case, we will just write Y_λ instead of Y_λ^n. We further define a function $\lambda : Y_\lambda \rightarrow (V \rightarrow Y)$, that reduces any higher-order expression y_λ to a plain expression by recursively applying y_λ to the input entity $v \in V$:

$$\lambda(y_\lambda) = v \mapsto \begin{cases} y_\lambda, & \text{if } y_\lambda \in Y \\ (\lambda(y_\lambda(v)))(v), & \text{otherwise} \end{cases}$$

We will usually assume an implicit application of λ, which means we can just write $y_\lambda(v)$ instead of $(\lambda(y_\lambda))(v)$. For example, with $0 \in \mathbb{N}_\lambda$ we can write $0(v)$, which is equivalent to $(\lambda(0))(v) = 0 \in \mathbb{N}$. Note that since expressions are evaluated recursively, we cannot differentiate between the sets $(Y_\lambda)_\lambda$ and Y_λ. Consequently, defining expressions over expressions can lead to ambiguity and should be avoided. Due to type variance rules, we can redefine any function $(V \rightarrow X) \rightarrow Y_\lambda$ as a function $X_\lambda \rightarrow (V \rightarrow Y)$ without losing generality. Yet, it can be convenient to use the general higher-order form of the parametric expression $X_\lambda \rightarrow Y_\lambda$, especially for ad-hoc definitions, where we do not want to think about the order of the expressions on either side of the function arrow. For example, the setter can be defined as a function $k_{set} : (X_k)_\lambda \rightarrow V_\lambda$, and we may freely use $k_{set}(x_k \in X_k)$ or $k_{set}(l \in (V \rightarrow X_k))$. We can redefine it as a function $k_{set} : (X_k)_\lambda \rightarrow (V \rightarrow V)$ without restricting its use, but the implementation must now guarantee a simple expression as the result.

Definitions: $M, R, \longrightarrow$ Finally, we establish that an LHS is an expression $m_\lambda \in M_\lambda$ that evaluates to a pair $(w \in \{\top, \bot\}, G \subseteq K) \in M$, where w indicates the success of the matching condition, and G are the goals that would be fulfilled by the rule's application. Respectively, an RHS may be any sentence expression $v_\lambda^* \in V_\lambda^*$. The star-operator takes precedence, which means $V_\lambda^* = (V^*)_\lambda$. At first glance the application of a rule appears to be a two-step process, where we evaluate the LHS and then—if it matched—evaluate the RHS. Yet, the evaluation of the LHS already determines the result of the RHS, as the input entity for both steps must be the same. We can represent the results of both sides with the set $R = M \times V^*$ and define the set of possible rules as R_λ. We use our familiar arrow notation to combine an LHS m_λ with an RHS v_λ^* to get a rule in R_λ:

$$m_\lambda \longrightarrow v_\lambda^* \equiv v \mapsto \begin{cases} (m_\lambda(v), v_\lambda^*(v)), & \text{if } m_\lambda \text{ matches } v \in V \\ (m_\lambda(v), []), & \text{otherwise} \end{cases}$$

Example: Expressions

In this example we show how our various notational constructs can be described in terms of expressions. We use plain expressions wherever possible, as the unnecessary use of higher-order expressions can obfuscate the definitions. For the accessors we define an attribute k with value space X_k. The RHS, LHS, and rules were already explicitly defined. Note that P are the rules of a particular grammar, while R_λ is the set of all possible rules. The final three lines describe the derivation algorithm as a whole and our two types of bracket notation. How these three functions work in detail will be discussed in Sections 3.4.1 and 3.4.2.

$$\begin{aligned}
k_{has} &: V \to \{\top, \bot\} \\
k_{get} &: V \to X_k \\
k_{or} &: (X_k)_\lambda \to (V \to X_k) \\
k_{set} &: (X_k)_\lambda \to (V \to V) \\
k_{def} &: (X_k)_\lambda \to (V \to V) \\
k_{tag} &: V \to V \\
LHS &\in M_\lambda \\
RHS &\in V_\lambda^* \\
R_\lambda &= (M \times V^*)_\lambda \\
P &\subset R_\lambda \\
derive &: \mathcal{P}(R_\lambda) \to (V \to V^*) \\
[\ldots] &: (R_\lambda)^* \to (V \to V^*) \\
\langle \ldots \rangle &: (R_\lambda)^* \to (V \to V^*)
\end{aligned}$$

3.4.1 Sub-Grammars

A cadence and a chord exist on different levels within the semantic structure of a piece. The cadence describes an abstract relation between tension and release, which is at a lower level expressed as a concrete progression of chords. Let us consider the latest version of our cadence generator, which we defined in Section 3.3.3. While the fulfillment of the cadence is well encapsulated within the first rule, the four remaining rules expose a lot of details about the chord generation process. For example, the *third* goal is fulfilled within the global scope of the derivation, but will hardly become relevant outside of chord generation. In this section we introduce nested derivation, which allows us to encapsulate rules in sub-grammars. Grammars with sub-grammars are also known as *hierarchical grammars* and have been used for music generation by McCormack [McC96].

Definitions: $derive, axiom$ Context-free derivation over a particular set of rules can be understood as a mapping from an axiom, which is a single entity, to a sentence, which is a sequence of entities. We define a function $derive : \mathcal{P}(R_\lambda) \rightarrow (V \rightarrow V^*)$, which takes a set of rules as its parameter and returns a sentence expression that performs context-free derivation in the familiar manner. With *derive* we can use a context-free sub-grammar as the RHS of a rule. In order to remove the need for a dedicated starting symbol, we define the special predicate *axiom*, which only matches the direct input entity for the expression returned by *derive*. In terms of goal-fulfillment, we will treat a sub-grammar as a black-box, where attributes can pass freely between the boundaries, but the fulfillment state is local to the particular derivation. Every application of *derive* pushes a clean fulfillment context onto the fulfillment context stack of the axiom, and removes this context from the terminals once derivation has finished. Any operation that affects fulfillment uses the top-most fulfillment context of the entity. In practice this means that all goals on the axiom of a nested grammar are unfulfilled. Any internal fulfillment will be discarded once it terminates and not leak into the scope of the super-grammar.

Example: Sub-Grammars

In this example we will encapsulate the rules for chord generation into a CHORD sub-grammar and use this sub-grammar in our main grammar CADENCE. Note that we use the rule labels as variables in order to syntactically simplify the example. It would be equivalent to replace the CHORD label with the body of the chord rule. The most important part of this example is the use of the *derive* function as the RHS of rule p_2. Rule p_1 is almost exactly the same as in Section 3.3.3, except that we now use the *axiom* predicate, rather than the *cadence* goal, which served no purpose beyond being an entry point. Rule p_2 fulfills the *chord* goal by applying the CHORD sub-grammar in its RHS . The four rules q_1 to q_4 correspond to rules p_2 to p_5 of the example in Section 3.3.3. We do not use the *axiom* here since different rules should be used, depending on the value of the *chord* attribute. Since each grammar has its own goal fulfillment context, rules q_1 and q_2 of the CHORD grammar can fulfill the *chord* goal, even though it was already fulfilled in p_2 of the CADENCE grammar.

$$
\begin{aligned}
&\text{CADENCE} : \mathit{derive}(\\
&\quad p_1 : \mathit{axiom} \longrightarrow [\langle \mathit{chord}_{set}(7), \mathit{pitch}_{+}(P5)\rangle, \mathit{chord}_{set}(\mathit{triad})]\\
&\quad p_2 : \mathit{chord}_{!} \longrightarrow \text{CHORD}\\
&)
\end{aligned}
$$

$$
\begin{aligned}
&\text{CHORD} : \mathit{derive}(\\
&\quad q_1 : \mathit{chord}_{!} = 7 \longrightarrow [\langle \mathit{chord}^{!}_{set}(\mathit{triad}), \mathit{mood}_{set}(h)\rangle, \mathit{pitch}_{+}(m7)]\\
&\quad q_2 : \mathit{chord}_{!} = \mathit{triad} \longrightarrow [\langle\rangle, \langle \mathit{third}_{set}, \mathit{mood}_{def}(h)\rangle, \mathit{pitch}_{+}(P5)]\\
&\quad q_3 : \mathit{third}_{!} \wedge \mathit{mood} = h \longrightarrow \mathit{pitch}_{+}(M3)\\
&\quad q_4 : \mathit{third}_{!} \wedge \mathit{mood} = s \longrightarrow \mathit{pitch}_{+}(m3)\\
&)
\end{aligned}
$$

3.4.2 Fork and Pipe

The *derive* function implements the most general strategy for combining rules in our system, but in many cases we do not need the complexity that comes with a recursive derivation process. Instead of a complete sub-grammar consisting of four rules, we could generate the chord notes in parallel and use conditional clauses to modify the third and add the seventh. The steps of this process have a fixed order, and we would have to carefully enforce this order with goals. In this section we will look at two particular functions that can be used as a simpler alternative for defining deterministic, non-recursive sub-grammars. We know them from the prior examples as our two types of bracket notation, but until now they lacked a formal definition and name: *fork* and *pipe*.

Definitions: $m_\top$ The core mechanism behind fork and pipe can be explained with a restricted definition as two functions $(V_\lambda^*)^* \to (V \to V^*)$ over sequences of sentence expressions, which we will later generalize to functions $(R_\lambda)^* \to (V \to V^*)$ over sequences of rules. In order to treat the set of right-hand sides V_λ^* as a subset of the set of rules R_λ, we define a trivial LHS $m_\top = (\top, \{\}) \in M$ and assume that an RHS $v_\lambda^* \in V_\lambda^*$ is equivalent to the rule $m_\top \longrightarrow v_\lambda^*$. Note that this would also allow the use of a plain RHS as a rule for the *derive* function. Since *derive* additionally requires that at least one goal must be fulfilled, a rule with a trivial LHS will simply be ignored.

Definitions: $[\ldots]$ The fork function performs parallel application and evaluates each input rule directly on the input entity. The results are concatenated into a flat sequence of children. We use our familiar square bracket notation as a special syntax. Assume $f, f_i \in V_\lambda^*$ and $f = [f_1, \ldots, f_n]$, then $f(v) = f_1(v) \cdot \ldots \cdot f_n(v)$ for any entity v, where $\cdot$ is the concatenation operator. This definition is fully compatible with our prior use of square brackets, but we are now using arbitrary sentence expressions within. For example, we can nest forks, but since the resulting sequences will be concatenated, the nested fork $[[[f_1], f_2], f_3]$ is equivalent to the flat fork $[f_1, f_2, f_3]$. The *empty fork* $[]$ removes the branch from the sentential form since it describes a replacement of the parent entity with zero children. Additional examples are shown in Table 3.2.

Definitions: $\langle \ldots \rangle$ The pipe function applies its input rules sequentially. Only the first rule is evaluated on the input entity. The second rule is evaluated on the results of the first rule, the third rule is evaluated on the results of the second rule, and so on. We use angle brackets as a special syntax for a pipe. Using the same variables and concatenation operator as before, if $f = \langle f_1, \ldots, f_n \rangle$ and $f_1(v) = (w_1, \ldots, w_m)$, then $f(v) = [\langle f_2, \ldots, f_n \rangle(w_1), \ldots, \langle f_2, \ldots, f_n \rangle(w_m)]$. For the base case, where $n = 0$, we define $\langle\rangle(v) = v$, which we call the *empty pipe*. The evaluation of a pipe can be modelled by a derivation tree, where v is the root at depth 0, $(w_1, \ldots, w_m)$ are its children at depth 1, and each entity in $f(v)$ is a leaf node at depth n. A rule f_i is applied to all nodes at depth $i - 1$ and generates a new tree level at depth i. Again, the notation is fully compatible with the prior examples, and we may use any rule within the brackets. We can even nest pipes, but—under our restricted definition—a nested pipe $\langle\langle\langle f_1 \rangle, f_2 \rangle, f_3 \rangle$ will be equivalent to the corresponding flat pipe $\langle f_1, f_2, f_3 \rangle$. Additional examples and interactions with the fork function are shown in Table 3.2.

In order to generalize fork and pipe from $(V_\lambda^*)^*$ to $(R_\lambda)^*$, we must find a meaningful interpretation for rules with a non-trivial LHS. In a fork, a failing rule will be interpreted as the empty fork $[]$. For example, given two sentence expressions $f, g \in V_\lambda^*$, $[\bot \longrightarrow f, g] = [[], g] = g$. For the pipe, we declare that a failing rule will terminate the current branch. For example, $\langle \bot \longrightarrow f, g \rangle$ is equivalent to the empty pipe $\langle\rangle$. When a non-trivial LHS is involved, nested pipes may have different semantics, as the nested pipe can terminate, but evaluation will continue in the outer pipe. For example, $\langle\langle \bot \longrightarrow f \rangle, g \rangle$ is equivalent to $\langle\langle\rangle, g\rangle = \langle g \rangle$. The behaviour of the pipe becomes more interesting when the LHS matches certain entities, but not others. Given two entities $w_1, w_2 \in V$ and an LHS $m_\lambda \in M_\lambda$ that matches w_1, but not w_2, $\langle [w_1, w_2], m_\lambda \longrightarrow f \rangle = [f(w_1), []] = f(w_1)$. Finally, goal fulfillment inside fork and pipe is generally allowed, but they do not create their own fulfillment context.

Example: Fork and Pipe

In this example we will define a grammar that is equivalent to the one from Section 3.4.1, but we will use fork and pipe instead of derive. For the CADENCE grammar we use a pipe at the top-level. As usual, p_1 creates two entities with the *chord* goal, where the first has the chord value 7 and is transposed by a fifth. The *triad* goal is not longer necessary. In p_2 we feed each of the two entities into the CHORD grammar. On a high level, CADENCE corresponds to example 10 in Table 3.2, and p_1 corresponds to example 7. Note how we no longer have to use an LHS here, since the application order is fixed.

#	$f \in (V \to V^*)$	$f(v \in V)$
1	$[]$	$()$
2	$\langle\rangle$	(v)
3	$[a]$	$(a(v))$
4	$\langle a\rangle$	$(a(v))$
5	$[a, b]$	$(a(v), b(v))$
6	$\langle a, b\rangle$	$(b(a(v)))$
7	$[\langle a, b\rangle, c]$	$(b(a(v)), c(v))$
8	$[a, \langle b, c\rangle]$	$(a(v), c(b(v)))$
9	$\langle a, [b, c]\rangle$	$(b(a(v)), c(a(v)))$
10	$\langle [a, b], c\rangle$	$(c(a(v)), c(b(v)))$
11	$[\langle a, b\rangle, \langle c, d\rangle]$	$(b(a(v)), d(c(v)))$
12	$\langle [a, b], [c, d]\rangle$	$(c(a(v)), d(a(v)), c(b(v)), d(b(v)))$

Table 3.2: This table shows basic combinations of fork and pipe. We define four expressions $a, b, c, d \in V_\lambda$, which describe arbitrary one-to-one mappings between entities. We do not use sentence expressions V_λ^* because it is important that each of these expressions generates exactly one entity. For example, if a could return an arbitrary number of entities, writing $b(a(v))$ in the third column would be invalid. In summary, the fork function provides concatenation for sentence expressions, while the pipe function can be used for function composition and mapping over sentences. Consider examples 10 and 12, where the first expression in the pipe generates two entities, and the second expression is applied to both.

The CHORD grammar has changed significantly. Each of the four rules q_1 to q_4 generates one note of the chord, which are concatenated using a top-level fork. Rule q_1 generates the bass note, and since the pitch should not change, we simply use an identity mapping expressed as an empty pipe. In q_2 we generate the third, by first shifting the pitch by a minor third interval. We then use a guarded expression that only matches if the chord is a seventh chord or the mood is happy. If either is the case, we shift the third up by another half-step to create a major third, otherwise pipe ignores the RHS and stops after the first expression, yielding a minor third. Rule q_3 generates the fifth, which is rather straight-forward. Finally, in rule q_4 we only want to generate the fourth note if the chord is a seventh chord. As fork replaces a failing rule with an empty fork [], this branch will be ignored if the chord is a triad. Even though we use two LHS in this grammar, we did not need any goal fulfillment since fork and pipe, unlike *derive*, are not inherently recursive and will terminate either way.

CADENCE : $\langle$
$\quad p_1 : [\langle chord_{set}(7), pitch_{+}(P5)\rangle, chord_{tag}]$
$\quad p_2$: CHORD
$\rangle$

CHORD : [
$\quad q_1 : \langle\rangle$
$\quad q_2 : \langle pitch_{+}(m3), chord = 7 \vee mood = h \longrightarrow pitch_{+}(1)\rangle$
$\quad q_3 : pitch_{+}(P5)$
$\quad q_4 : chord = 7 \longrightarrow pitch_{+}(m7)$
]

3.4.3 Sentences

Although we can now define sub-grammars of varying complexity and use them as the RHS of a rule, their application is not truly encapsulated. With the *derive* and pipe functions a sentence generated by a sub-grammar immediately replaces an entity in the outer grammar's sentential form, where they are visible to its rules. Yet, sometimes we will want to transform or aggregate the result of the sub-grammar before we integrate it into the scope of the outer grammar. The derivation of the transformed entities can then resume in the familiar manner. Beyond encapsulation, this allows us to run context-sensitive passes over sub-grammar results, significantly expanding the space of possible operations. Schwarz and Müller [SM15] were the first to use sub-grammar results as first-class citizens in shape grammars.

Definitions: $count, filter, k_{set}(v^*)$ Using sentences as first-class citizens simply means that we have functions that take sentence expressions as their input. Fork and pipe are examples of such functions and can be used for concatenation and mapping. Another example is the function $count : V_{\lambda}^{*} \to (V \to \mathbb{N}) : v_{\lambda}^{*} \mapsto (v \mapsto |v_{\lambda}^{*}(v)|)$, which calculates the length of the sentence generated by v_{λ}^{*} for the input entity v. A function $filter : V_{\lambda}^{*} \times \{\top, \bot\}_{\lambda} \to (V \to V^{*}) : (v_{\lambda}^{*}, c) \mapsto v \mapsto \ldots$ can be used for selecting entities from the sentence $v_{\lambda}^{*}(v)$ that match the condition c. Our implementation provides additional aggregation functions, which we will not discuss in detail. It is further possible to retain a sentence for later use. Given an attribute k over the value space V^{*} of sentences, we can use our familiar setter $k_{set}(v_{\lambda}^{*})$ to store the result of $v_{\lambda}^{*} \in V_{\lambda}^{*}$ under k. We can access the generated entities via the getter of k at any later point.

Definitions: *append*, *select* Finally, we define a generic way of integrating a sentence into the sentential form. Assume we have a sentence expression $v^*_\lambda \in V^*_\lambda$ and an input entity $v \in V$. We only need a certain set of attributes from $v^*_\lambda(v)$, and at the same time we want to preserve some attributes of v. We do not know anything about v^*_λ, so it may or may not preserve the attributes we need. How can we guarantee that the internals of v^*_λ do not affect the main grammar? We define a function $\mathit{append} : V^*_\lambda \to (V \to V^*)$, that copies the attributes of every entity v_i in the sentence $v^*_\lambda(v)$ to a clone of v. The new entities will preserve all attributes of v that do not appear in the corresponding entity v_i. Naturally, both $v^*_\lambda(v)$ and $(\mathit{append}(v^*_\lambda))(v)$ generate the same number of entities. Further, we define $\mathit{select} : \mathcal{P}(K) \to (V \to V)$, which marks every optional attribute on v as undefined, unless it appears in the specified subset of K. This allows us to discard implementation-specific attributes and define a clean output interface for sub-grammars.

Example: Sentences

In this example we will use isolated evaluation of sentence expressions to define a variant of our earlier CHORD grammar. This time, we want to set the value of a new attribute *gain* for the chord notes, which describes the relative loudness of a note. *gain* is defined over the non-negative real numbers, where 0 is complete silence, and, for any two gain values x, y, we assume x is louder than y iff $x > y$. Depending on the value of the *chord* attribute, the CHORD grammar will generate a triad or a seventh chord. If we use the same *gain* value for each note, the four notes of the seventh chord will naturally be louder than the three notes of the triad. Instead, we want both chords to have the same total gain, with 1/3 for each triad note, and 1/4 for the seventh chord notes.

We define the EQUALIZED_CHORD grammar as a pipe that is wrapped in an *append* function. This outer *append* call will guarantee that the attributes of the input entity will be preserved, regardless of where we use this grammar. In p_1 we evaluate the CHORD grammar and store the resulting entities in the *notes* attribute, which is defined over the sentences V^*. After p_1 there is still only one entity that gets processed by the pipe, since the notes generated by CHORD are isolated inside the attribute. In p_2 we use the *count* function to calculate the number of chord notes and set the *gain* attribute accordingly. Next, in p_3 we use the getter *notes* again to fetch the chord notes and use *select* to isolate the *pitch* attribute. After all, we may not know whether the output of CHORD defines its own *gain* value that would overwrite the one we have calculated in p_2. The *append* call combines the single input entity that carries the *gain* with the three or four *pitch* values that we selected. Finally, we use *select* to remove all attributes except for *pitch* and *gain*.

$$
\begin{aligned}
&\text{EQUALIZED\_CHORD} : \mathit{append}(\langle\\
&\quad p_1 : \mathit{notes}_{\mathit{set}}(\text{CHORD})\\
&\quad p_2 : \mathit{gain}_{\mathit{set}}(1/\mathit{count}(\mathit{notes}))\\
&\quad p_3 : \mathit{append}(\langle \mathit{notes}, \mathit{select}(\{\mathit{pitch}\}\rangle)\\
&\quad p_4 : \mathit{select}(\{\mathit{pitch}, \mathit{gain}\})\\
&\rangle)
\end{aligned}
$$

3.4.4 Ranges

The fork function allows us to replace the parent entity with a sequence of children, but we have to explicitly specify these children in one way or another. We are still lacking a function for expanding a single entity into multiple ones. Cloning is arguably the simplest operation for generating an entity sequence from a single entity. We can define a *clone* function that generates a specified number of copies of the input. As opposed to fork, this function would allow us to specify the length of the generated sequence dynamically. Yet, truly identical clones are of course not very useful since they will all be derived in the same way. Instead of generating exact copies of the input, we want to associate each output entity with a unique value.

Definitions: *range* We define a function $\mathit{range} : \mathbb{N}_\lambda \times (\mathbb{N} \to V_\lambda^*) \to (V \to V^*)$ which performs the mapping $(n, f) \mapsto [f(0), \ldots, f(n-1)]$. The input n is the number of copies, and f is a function that takes the index of the copy and maps it to a sentence expression. First, we generate indices $(0, \ldots, n-1)$ and apply f to each, then we concatenate the resulting n sentence expressions with a fork. We definitely want to generalize *range* to take any numeric expression in $\mathbb{N}_\lambda$, rather than a constant number. It may further be convenient to supply the evaluated value n as an additional parameter to f, so that one can transform the index based on the total. One use-case for this is the generation of a reversed range $(n-1, \ldots, 0)$. Within a sentence, the index of a particular entity is implicit, and our implementation provides helper functions for iterating over sentences.

Example: Ranges

In this example we use *range* to generate a dynamic number of measures, which is supplied via a parameter n. In p_1 we pass the value n to *range*. Instead of $i \mapsto \mathit{index}_{\mathit{set}}(i)$ we can directly pass the setter function to write the index to the *index* attribute. Rule p_2 matches on even and odd numbered measures to generate an alternating pattern of a dominant seventh and a tonic triad. It also demonstrates how a fork with guarded expressions can emulate a case differentiation. Finally, in rule p_3 we apply the EQUALIZED_CHORD grammar from the previous example. CHORDS is actually just another parametric expression. For example, to generate 8 chords we could write CHORDS(8), which is an expression $V \to V^*$.

```
CHORDS : $(n \in \mathbb{N}_{\lambda}) \mapsto \langle$
    $p_1 : range(n, index_{set})$
    $p_2 : [$
        $even(index) \longrightarrow \langle chord_{set}(7), pitch_{+}(P5) \rangle$
        $odd(index) \longrightarrow chord_{tag}$
    $]$
    $p_3$ : EQUALIZED_CHORD
$\rangle$
```

3.5 Variation

A human composer uses their creative intuition to locate a piece in a space of possibilities bounded by whatever they consider to be a valid musical idea. Up to now we side-stepped the problem of modelling the creative aspect by avoiding decisions in the derivation process altogether. For every non-terminal there was only a single matching rule and consequently the resulting sentence was fully determined from the start. Yet, abstraction is more than an efficient encoding. We can see a *minuet in G major* as a placeholder for many pieces, not just as an identifier for one particular minuet. Variation is the missing ingredient, which elevates formal grammars from a complex tool for musical notation to a powerful tool for automatic composition.

When we expand the possibility space by offering alternatives for the replacement of some non-terminal, we also need a strategy for deciding between these alternatives. Some classic composers published musical games that allow the composition of new pieces by randomly selecting from a framework of predefined bars, most notably Mozart with his minuet generator [Zas05]. The player of the game needed no musical knowledge, only a pair of dice. For this work we will use a similar approach, where the grammar is the framework, and the replacement decisions are delegated to a non-deterministic selection process. Leaving the decisions up to randomness is arguably not very sophisticated, and we could imagine alternative strategies which incorporate empirical analysis of existing music or direct user intervention. On the other hand, the chance-based approach is trivial to implement and will nonetheless produce interesting results. It also forces us to actively shrink the possibility space around parts of the process that should have less variance, which hopefully leads to grammars that consistently generate satisfying output for various starting conditions.

Definitions: $rng, seed$ At the core of all non-deterministic features of the grammar lies the canonical rng attribute, which is a function that generates a new random number from the interval $I = [0, 1] \subset \mathbb{R}$ each time it is evaluated. Note that such a function needs mutable internal state, for example a counter that gets incremented after each evaluation. When the rng attribute is inherited, the internal state is not cloned, which means child entities are desynchronized by default. We define $seed : (\mathbb{N}_\lambda)^* \to (V \to V)$, which uses the specified numbers to initializes a new random number generator and sets it as the value of the rng attribute. The $seed$ function can be used to synchronize children, which will be discussed in Section 3.5.1.

Definitions: $rand, uniform$ Based on the rng attribute we can define additional non-deterministic functions. We define $rand : V \to I$, which simply evaluates the rng function of the input entity and returns the random number. For example, the expression $[k_{set}(rand), k_{set}(rand)]$ will generate two entities with attribute k that holds a random number. Since the state of an inherited random number generator is shared, both evaluations of $rand$ will yield a different result. We further define $uniform : (\mathbb{R}_\lambda)^2 \to (V \to \mathbb{R})$, which takes a tuple of number expressions $(a_\lambda, b_\lambda) \in (\mathbb{R}_\lambda)^2$ and returns an expression that samples from a uniform distribution between $a_\lambda(v)$ and $b_\lambda(v)$, when evaluated with some entity $v \in V$.

Definitions: $pick, choice, derive$ In a non-deterministic grammar a non-terminal may be matched by multiple rules. In order to make a decision, we define a discrete choice mechanism for rules with the function $pick : \mathcal{P}(R_\lambda) \to (V \to R)$. It evaluates each input rule and discards those that did not match. The remaining ones are the candidates Q, of which one is selected at random using a number generated by rng. Note that $pick$ propagates the result of the selected rule to the caller, which is the reason why it returns an expression over R. When Q is empty, $pick$ has no available options and will not match. The $derive$ function uses $pick$ internally for non-deterministic derivation. For parametric variation, we define $choice : Y \to (V \to Y)$, which can be used to randomly select a value from an arbitrary set Y. Note that $choice$ is undefined when no candidate exists, and an implementation may simply raise an error in that case.

Definitions: $\xrightarrow{w}$ When it comes to picking an option, uniform selection is not always optimal. Deliberate use of uncommon patterns can lead to great original compositions, but their overuse might sound strange. The selection frequency can be controlled with a weighting mechanism. In Section 3.4 we defined the result of an LHS as a tuple $(w \in \{\top, \bot\}, G \subseteq K)$. We will now generalize the binary value w to a non-negative number $w \in \mathbb{R}^+$, which we call the *weight* of the rule. A rule with weight 2 will be selected twice as often as a rule with weight 1, and $w = 0$ indicates a matching failure. For some LHS $m_\lambda \in M_\lambda$ and RHS $v_\lambda^* \in V_\lambda^*$, we define a weight $w_\lambda \in \mathbb{R}_\lambda^+$ by writing $m_\lambda \xrightarrow{w_\lambda} v_\lambda^*$. For a trivial LHS, we will just write $\xrightarrow{w_\lambda} v_\lambda^*$ instead of $\top \xrightarrow{w_\lambda} v_\lambda^*$. If no weight is specified, we assume a weight of 1, so uniform selection will be the default

$part_1$	$part_2$	progression	probability
p_3	p_3	V-I-V-I	$2/3 * 2/3 = 4/9$
p_3	p_4	V-I-I-IV	$2/3 * 1/3 = 2/9$
p_4	p_3	I-IV-V-I	$1/3 * 2/3 = 2/9$
p_4	p_4	I-IV-I-IV	$1/3 * 1/3 = 1/9$

Table 3.3: Each row shows one possible combination of rules p_3 and p_4 applied to the two *part* entities generated by p_2. Rule p_3 has a relative weight of 2.0 and will be selected twice as often as rule p_4, leading to the probabilities specified in the last column.

behavior. For *choice* we may use the same syntax to assign weights to the options, albeit without an LHS.

Example: Variation

In this example we will use the $uniform$, *pick*, and *choice* functions for variation. In p_1 we use *choice* to select a random value for the *mood* attribute, h for 'happy' and s for 'sad'. Neither option has a weight, so either will be selected with the same probability. Next, rule p_2 will fulfill $bars$ and generate two new entities with the *part* goal. The *part* goal is interesting since it can be fulfilled by two rules. p_3 will generate our familiar cadence, a V-I progression in roman numeral notation, while p_4 will generate a I-IV progression. Since *derive* uses *pick* internally, the system will select a random rule from $Q = \{p_3, p_4\}$ for each of the two entities, resulting in one of the four progressions shown in Table 3.3. In p_5 we use the EQUALIZED_CHORD grammar for generating the chords and attach the valueless *note* attribute. Finally, in p_6 we add some imperfections by using $uniform$ to slightly randomize the *gain* value.

PROGRESSION : *derive*(

$p_1 : axiom \longrightarrow \langle mood_{set}(choice(h, s)), bars_{set}(2)\rangle$

$p_2 : bars_! \longrightarrow range(bars, part_{set})$

$p_3 : part_! \xrightarrow{2.0} [\langle chord_{set}(7), pitch_+(P5)\rangle, chord_{tag}]$

$p_4 : part_! \longrightarrow \langle chord_{tag}, [\langle\rangle, pitch_+(P4)]\rangle$

$p_5 : chord_! \longrightarrow \langle \text{EQUALIZED\_CHORD}, note_{tag}\rangle$

$p_6 : note_! \longrightarrow gain_{set}(gain_{or}(1) * uniform(0.9, 1.0))$

)

3.5.1 Synchronization

Variation is the antithesis to repetition, yet for music generation we generally need both. Remember the rule from the prior examples that generated two entities with the *part* goal. The *part* goal was fulfilled by a non-deterministic sub-grammar with two possible results, which we may call a and b. What if, instead of the four possible combinations $\{aa, ab, ba, bb\}$ that are listed in Table 3.3, we wanted to restrict the result to only allow $\{aa, bb\}$? The two parts should be derived randomly, but both should be derived with the same rules and parameters. In a pure context-free grammar, once two entities are separated, we can no longer access information from the former sibling. We need a way of *synchronizing* the subtrees before the separation occurs. Quick and Hudak [QH13a] solve repetition with their *let-in* construct, which allows a subtree to have multiple parents, resulting in a duplication of its leaves in the generated sentence.

We usually do not want the subtrees to be completely identical. For our handling of the time dimension, which we will present in Chapter 4, every entity gets an explicit time interval and identical copies of a subtree would all end up in the same place. So, repetition is more general than duplication since it includes translation in the time dimension. Quick and Hudak avoid this distinction by encoding the temporal offset in the order of the terminals, so the absolute temporal scope of an entity is implicit during the derivation. Another limitation of their approach is that there is no way to *desynchronize* two subtrees. In our previous example we randomized the *gain* attribute to add imperfections as they might occur when a human plays the piece. If the two parts are still synchronized at that point, both will develop the exact same loudness profile, which is not very realistic. In this section we present two solutions, one based on seeding and one that uses isolated derivation.

A pseudo-random number generator (PRNG) generates a fixed, yet seemingly random and uniformly distributed sequence from an initial source of entropy, called a *seed*. One may think of the seed as a compressed sequence, which is unpacked by the rules of the PRNG, conceptually similar to the way a deterministic grammar takes an axiom and unpacks the sentence. Synchronizing n subtrees is then as simple as using the same seed for each of their roots. For desynchronizing the subtrees we have to retain some entropy from a point after the separation, but before the synchronization, using the following six-step process: (1) We generate the shared seed s and store it in an attribute. (2) We generate roots of the n subtrees using some grammar f, and (3) add an independently generated seed $s_i, i \in (1, \ldots n)$ to each subtree root. (4) We initialize the entity's PRNG with the shared seed s, which synchronizes (5) the application of the non-deterministic sub-grammar g. (6) We reintroduce the entropy and desynchronize all leaf entities by mixing the independent seed s_i with a random number from the PRNG initialized with s and using the result as a new seed. The following rule shows how we may implement these six steps within our system:

$$\langle s_{set}(rand), f, s'_{set}(rand), seed(s), g, seed(s' * rand)\rangle$$

Using $seed(s')$ at the end does not suffice because this would actually synchronize all entities generated by g in the same way we synchronized the result of f with $seed(s)$.

Instead, we combine the independent seed s_i, which is stored in the s' attribute and constant within the subtree, with a synchronized random number generated by *rand*, which uses the PRNG initialized in step (4) and is different for all m entities generated by g. Ideally, we would use a bijective operator to combine these numbers, but since the theoretic probability of picking any random number from a real interval is zero, we can just use multiplication. After step (6) each of the n times m entities should have a PRNG initialized with a different seed. While the example is suitable to illustrate the general mechanism, it is a low-level strategy and requires lots of user attention. For practical application we can define a higher-level construct which takes the two grammars f and g as input and automatically performs the other steps. Yet, there is an alternative solution, albeit less general, which covers many use-cases.

Instead of synchronizing the roots and developing n trees in parallel, we can apply the non-deterministic grammar g once and make n copies of the resulting subtree. The difference to the solution by Quick and Hudak is that we can independently define how the subtree is integrated under the subtree roots generated by f. Note that with this strategy g is evaluated before f and must therefore not depend on any attributes generated by f. The process works as follows: (1) We evaluate g once and retain the resulting sentence. (2) We evaluate f to generate the subtree roots, or rather the nodes where we later attach the subtrees. Finally, (3) we append the result of g to each result of f using some transformation operator that adapts the subtree to its new context, for example with *append* and *select*. Note that the seeded approach is more powerful since it runs the sub-grammar n times instead of once, and while the subtrees might be synchronized in one aspect, they are not required to be synchronized in every aspect. The second approach requires fewer steps and intuitively mirrors how we might compose a piece on paper, where we come up with a short motif and use it multiple locations with minor changes. The following rule shows how we could implement the second approach using an attribute k and *append* as the operator:

$$\langle k_{set}(g), f, append(k)\rangle$$

Example: Synchronization

In this example we will synchronize the generation of our chord sequence, so that it consists of either V-I or I-IV chord pairs, but never both. We define the non-deterministic sub-grammar CHORD_PAIR, where q_1 generates our V-I cadence and q_2 the I-IV pair. In the main grammar PROGRESSION, the rules p_1 to p_3 correspond to the three steps from before. First, we generate a single chord pair and store it in the *pair* attribute. Second, we generate two subtree roots and differentiate them via the *time* attribute. Third, we append the retained pair to each subtree root. Using *append* here preserves the *time* attribute, which means we have successfully synchronized the subtrees, without duplicating them exactly. Rule p_4 is self-evident. Finally, in rule p_5 we randomize the gain. Since we never reinitialized the PRNG, we can be sure that all leaf entities will be randomized independently.

$$
\begin{aligned}
&CHORD\_PAIR : pick(\\
&\quad q_1 : [\langle chord_{set}(7), pitch_{+}(P5)\rangle, \langle\rangle] \\
&\quad q_2 : [\langle\rangle, pitch_{+}(P4)] \\
&)
\end{aligned}
$$

$$
\begin{aligned}
&PROGRESSION : \langle \\
&\quad p_1 : pair_{set}(CHORD\_PAIR) \\
&\quad p_2 : range(2, time_{set}) \\
&\quad p_3 : append(pair) \\
&\quad p_4 : \text{EQUALIZED\_CHORD} \\
&\quad p_5 : gain_{set}(gain_{or}(1) * uniform(0.9, 1.0)) \\
&\rangle
\end{aligned}
$$

3.6 Conclusion

In this chapter we presented the various features and constructs of our procedural generation framework, which is based on context-free grammars. These included the handling of multi-dimensional entity spaces with parametric and attribute-based grammars, a formalism for declarative notation, nested derivation of sub-grammars and propagation of grammar results, and finally techniques for non-deterministic variation and synchronization. While we motivated these features with musical examples, the presented concepts are general and could potentially be applied to generation tasks in other domains. This means, on the other hand, that the framework is not yet optimized for music generation tasks. In the next chapter we will solve this by extending our generic framework with domain-specific musical models.

CHAPTER 4

Probabilistic Temporal-Split Grammars for Music Composition

Consider a musical composition with two or more interrelated voices that play at the same time. They are independent, in the sense that they can define their own movement and rhythmic patterns, but are synchronized to a shared metric and harmonic framework. In music notation individual voices and instruments are often written in separate *staves*, for example, the left and right hand of the piano piece in Figure 4.1. Each stave spans the whole duration of the piece, and the individual voices within are read sequentially. To implement this model in a context-free grammar we could derive each voice individually, then combine the results by playing them at the same time. Yet, without further constraints this approach is insufficient because due to non-determinism in the derivation process the voices might become desynchronized. As soon as the algorithm branches into different voices, sharing information between them becomes difficult. Rather, we need some way of providing a shared metric and harmonic context that allows the voices to actually fit together, instead of merely coexisting in the same temporal space.

The synchronization of voices is a fundamental challenge when generating non-monophonic structures with context-free grammars, and in this chapter we will discuss our solution. First, we define a common measurement system, expressed as a set of domain-specific temporal units, which allows us to independently align voices in different subtrees. For example, using a common tempo and time signature, we can guarantee that beats of multiple voices coincide. This solution is trivial since we simply adopt features from classical music notation. Second, we allow delaying the point of separation of voices to arbitrary depths in the derivation tree. For this we associate each entity with a time interval, in a way that is reminiscent of the use of spatial volumes in split-grammars, and allow layering, recursive splitting, and space-filling with a repeat operator. Finally, we combine the meta-level features of isolated derivation and operations on sentences with the domain-specific units and split operations to generate textures of abstract musical

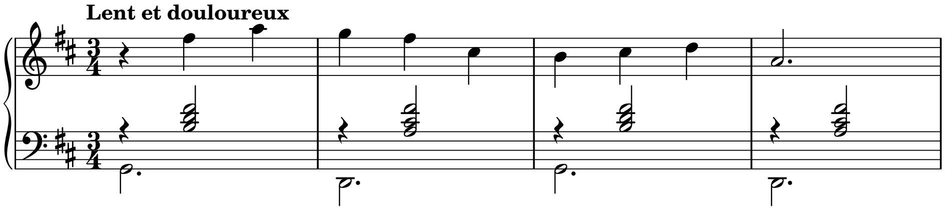


Figure 4.1: Score representation of measures five to eight of *Gymnopédie No.1* by Eric Satie.

aspects, which we can use as a common substrate for the voices of a polyphonic piece. For example, we can generate a harmonic progression in an isolated subtree and access it from an independently generated melody and bass line.

We summarize our approach as *probabilistic temporal-split grammars*, highlighting the synchronization challenges that arise from probabilistic generation and the parallels to the functional approach, parametric treatment of time, and recursive divisions of the timeline used in *probabilistic temporal graph grammars* (PTGG) [QH13a]. Additional details on how our method relates to existing works will be presented in Chapter 6. The implied connection to the concept of split-grammars in computer graphics is intentional. As explained in Section 2.3, the reason for this is not just the recursive division of the timeline with a split operator, which PTGGs already introduced to the music domain, but the use of an explicit scope. In music the sentence itself can be a sufficient representation, at least for monophonic structures. In computer graphics we usually consider at least two spatial dimensions, which means we cannot intuitively derive the placement of entities from a one-dimensional sentence. Thus, an explicit encoding of the spatial dimensions predates our use of this concept for modelling time in grammars for composition.

Musical Structures

The system we constructed in Chapter 3 allows us to delegate focus to parts of a composition and apply incremental changes to them. This focus is expressed as a mutable sequence of entities, and the resulting music will be an aggregation of that sequence's final state. Attributes allow us to attach custom semantics to the entities, which means we can conceivably build any kind of representation within the confines of our system. Rules describe how we may navigate between representations and, in the case of a generator, we move from the abstract towards the concrete. There is a vast spectrum between the top-most abstract intent of composing polyphonic music, and the bottom-most concrete sensation of that music as a sonic signal. In this section we will briefly look at the options that we may find on this spectrum and then develop a canonical representation level that we will use as a basis. One may call this level the *terminal* representation, but it is in no way a *final* representation, in the sense that this representation still has to be mapped to the desired output format.

Over the past centuries musicologists came up with a variety of abstract models for musical structure, melody, rhythm, dynamics, and harmony. As mentioned in Chapter 2, such models were often developed to aid musical analysis. This is not to say that they cannot be used in a generative context—the core premise of generative grammars is the reversal of the analytic parsing process—but implementing such models is not trivial. Consider, for example, the implementation of Lerdahl's and Jackendoff's *A generative theory of tonal music* [LJ$^+$83] by Hamanaka et al. [HHT06] and their discussion of the ambiguities in the original theory. Further, given our concrete requirement for polyphonic music, we must ensure that our chosen model is either versatile enough to cover our needs, or flexible enough to be combined with other models. Both aspects are very difficult to judge without a deep understanding of the specific theory and music theory in general. Ultimately, we concluded that for the purpose of generating polyphonic pieces we do not require a particular musicological framework, only that voices can be generated in a way that entangles them rhythmically and harmonically.

As an alternative to the top-down approach, where we derive the system from an advanced analytic model, we can construct it around our desired output, and add higher-level models as the need arises. If we intend the grammar results to be playable by an orchestra, they must be constrained by the rules of a score and physical limitations of real-world instruments. If, on the other hand, we only intend automated interpretation via software, we can pick from a broader range of encodings at different levels of abstraction. On the high-level end of the spectrum we have digital sheet music formats or the complex states of digital audio workstations. At the intermediate level we have formats such as MIDI, which reduces music to a set of time intervals on a semitone scale, but makes no assumptions about the actual sound synthesis, nor the structure of the piece. At the lowest level we have primitive waveforms and frequency space, where we can express sounds as combinations of oscillations. For our concrete goals we do not need to focus much on the physical qualities of the sounds, as they relate to timbre or audio effects. The output of the grammar can still be symbolic, in the sense that it would need interpretation by a musician or synthesizer.

After various experiments we ended up with something close to MIDI as our preferred base representation. It provides the necessary freedom for polyphony in the time dimension, while being accessible to humans and symbolic in nature. Other advantages of this model are a one-to-one association between entities and notes, a small, homogeneous set of canonical attributes, and a trivial rendering step. Familiar features from musical notation, such as meter for time measurements and diatonic scales for pitch, can be provided as optional higher-level models and will be discussed in later sections. From a musicological perspective, this model is rather primitive and should be familiar to anyone who has enjoyed a basic music education. The domain-agnostic features of the system already require learning a wealth of concepts. Having an accessible domain model seems reasonable if we intend the practical application of our work by anyone besides experts in both music and computer science.

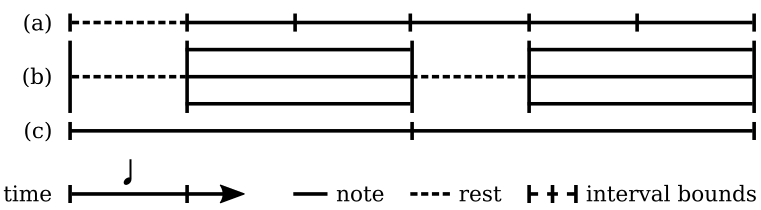


Figure 4.2: Three parallel sequences of intervals on a timeline. Together they approximate the temporal structure of the first two measures in Figure 4.1. Sequence (a) represents the melody, (b) represents the chords, and (c) the bass line.

4.1 Time

Polyphony is a musical feature that relates strongly to the time dimension, the representation of which is consequently of primary interest to us. In a score, such as Figure 4.1, we can find notes, rests, measures, tempo markings, time signatures, key signatures, dynamics—each either directly represents or associates with a point in time, a duration, or a time interval. The starting time of a note is encoded in the notes, rests, and other markings that precede it, and its length is determined by the note symbol. A musician reads the score left to right and plays note after note. The sentence generated by a grammar is a sequence, so assuming our terminals are notes, we could simply play the terminals in the order they were generated. This is arguably the most obvious interpretation of formal grammars for composition, and it was implied for the examples in Chapter 3. Just like note symbols, an entity in this model may have a duration parameter for representing rhythmic variation. For playback, we can calculate the cumulative sum of these duration values to get the absolute offset of a note on the timeline. Figure 4.2 shows a sequence (a) of six quarter note intervals and a sequence (c) of two whole measure intervals.

With a sequence we can express a monophonic melody or bass line. In order to express a *chord*, which we could see as a trivial example of polyphony, the intervals of multiple notes must coincide. We could achieve this by associating each entity with a set of pitch values, where each pitch spawns a note over the entity's time interval, analogous to multiple note heads along a single stem. This generalizes to rests (zero pitch values), single notes (one pitch value), and chords (multiple pitch values), while still using a sequential encoding. Sequence (b) in Figure 4.2 shows an interval sequence that contains rests as well as chords. Yet, for any such sequence, the bounds of the intervals are global—there cannot be a note across interval boundaries. We can only express a singular rhythmic pattern. In an earlier work [Eib16] we used this approach with an additional mechanism, where we could mark a note as *played* or *held*, to optionally hide divisions caused by superimposed patterns of finer temporal granularity. In practice this model was not very intuitive and seemed inadequate for achieving the explicit goal of generating polyphonic pieces.

We can think of a single voice as a special case of an interval arrangement, where the start of an interval coincides with the end of the preceding interval. In the same vein, parallel

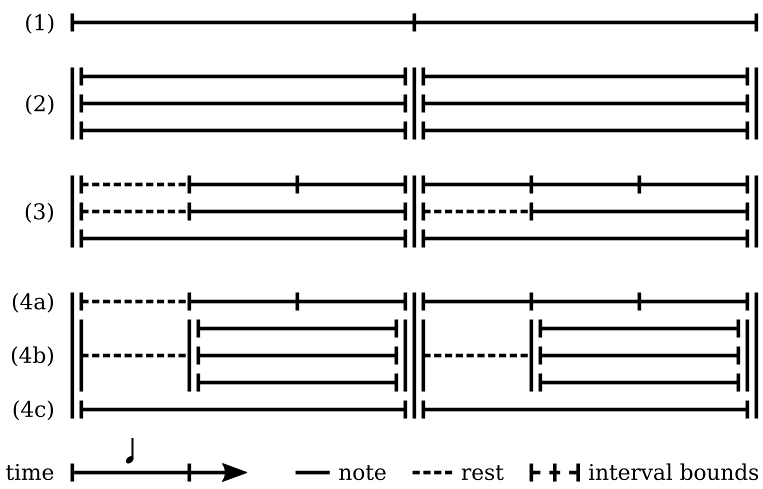


Figure 4.3: A step by step construction of the temporal structure from Figure 4.2 using an explicit interval for every entity and both sequential and parallel placement. The nesting of the interval bounds indicates how the replacement intervals are defined in relation to the replaced interval. In (1) we start with a sequence of two whole measure intervals. (2) replaces each measure with three parallel voices, which we split into notes and rests in (3). In (4b) we once again use parallel voices to generate chords.

voices would be an arrangement where the bounding intervals of the voices overlap. For example, we can interpret Figure 4.2 as at least three levels of arrangements. The initial interval contains a top-level parallel arrangement of three voices (a), (b), and (c), each of which contains a second-level sequential arrangement, and a third level of parallelism in the chords of (b). Traditional scores are mostly limited to these three levels (four if you count the measure level), but the recursive nature of the derivation process allows us to extend this principle to arbitrary depths. We can freely alternate between placement strategies on a per rule basis. Figure 4.3 shows how we may generate the two measures using a total of four levels of nested arrangements.

Definitions: *time*, *span* A consequence of mixing placement strategies is that the note duration alone no longer determines a unique arrangement. We solve this by borrowing the concept of *scope* from split-based grammars. The scope describes the boundaries of an entity and encodes its absolute placement, so we avoid dealing with context-sensitive dependencies between terminals. In graphics the scope is usually represented by a 3D transformation of space, whereas in music it would be a 1D transformation of time. We encode this transformation as a number pair $(t_0, t_1) \in \mathbb{R}^2$ and define $\Delta = t_1 - t_0$. Δ may also be negative if $t_1 < t_0$, although we will usually refrain from using negative time intervals. We integrate this model into the grammar with two real-numbered attributes *time* for t_0 and *span* for Δ. If not explicitly specified, we assume *time* and *span* are equal to zero.

Example: Time

In this example we apply our temporal model to replicate the structure of the fifth measure of *Gymnopédie No.1*, which is the first measure shown in Figure 4.1. It consists of two quarter notes for the melody, a three-note chord, and a bass note. We assume that the length of a quarter note is 1 and set the duration of the measure to 3 in p_1. In p_2 we fork our single entity into three parallel parts for the bass, chord, and melody. The bass note fills the whole measure and we can use the empty pipe $\langle\rangle$ as the identity function. In the CHORD sub-grammar we first delay the chord by 1 in q_1 and set its duration to one half note in q_2. Rule q_3 copies the entity to generate three chord notes. Finally, the MELODY sub-grammar first sets the *span* to 1 in r_1 and then generates two notes in r_2, which are offset by 1 and 2 respectively.

$$
\begin{aligned}
&\text{PIECE} : \langle \\
&\quad p_1 : span_{set}(3) \\
&\quad p_2 : [\langle\rangle, \text{CHORD}, \text{MELODY}] \\
&\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\text{CHORD} : \langle \\
&\quad q_1 : time_{set}(time + 1) \\
&\quad q_2 : span_{set}(2) \\
&\quad q_3 : range(3) \\
&\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\text{MELODY} : \langle \\
&\quad r_1 : span_{set}(1) \\
&\quad r_2 : [time_{set}(time + 1), time_{set}(time + 2)] \\
&\rangle
\end{aligned}
$$

4.1.1 Meter

A random arrangement of notes, uniformly distributed over time and frequency, will not resemble music. Since we want to use randomness for variation we must structure the entity space in a way that increases the likelihood for generating music-like arrangements. For that we need to consider the mechanism behind rhythm and harmony and find models for expressing these features effectively. The simplest rhythm is just a beat, a regular pulse of air pressure over time; think of the clicks produced by a metronome. Two beats fit together if their pulses frequently coincide. For example, two metronomes clicking at the same rate will click together in harmony, assuming they started at the same time. If one of them clicks faster than the other, for example at a ratio of 2:1 or 1000:999, sometimes they will click in unison and sometimes they will not. For the first ratio, 1 out of 2 clicks will coincide, for the second ratio only 1 in 1000 will. Musical theory revolves around frequencies that harmonize, which means that their ratios tend to be simple, such as 2:1 and not 999:1000.

Our metric model closely resembles that of classic musical notation. In a score the temporal organization of the piece revolves around three measuring systems that are related by simple ratios. It uses a regular pattern of pulses called the *beat* as the fundamental unit, and we use the *tempo* parameter $T \in \mathbb{R}_+$ to convert between beats and physical time in seconds. In digital contexts the tempo is usually specified as the number of beats per minute (BPM). Multiple beats are grouped into measures, and the number of beats within a measure is encoded in the *beat count* parameter $N \in \mathbb{R}_+$. The third measuring system expresses the duration in terms of fractions of a *whole note.* The conversion to beats is defined by the *beat type* parameter $B \in \mathbb{R}_+$, where the length of a whole note is assumed to be B beats. The pair (N, B) has the same semantics as the two numbers of the time signature in musical notation. For example, if $N = 3$ and $B = 4$, there are three beats per measure, and the duration of each beat is equivalent to $1/4$ of a whole note. We define the conversion factors of various temporal units that can be derived from these parameters in the upper part of Table 4.1.

A measuring system based on tempo and meter is very intuitive for musicians, but it has some limitations. For one, it anchors the fundamental beat unit, and by extension all other dependent units, to an absolute duration. This means that if we rely solely on these units, all our rules will be bound to a particular temporal granularity. For example, we can replace a quarter note with two subsequent eighth notes, but we cannot easily divide a note of arbitrary length into two equal parts. For this purpose we use an additional measuring system which is decoupled from the beat unit. In this system a duration $\Delta \in \mathbb{R}$ serves as the base unit. With these *relative* units we can divide arbitrary time intervals. Relative units also solve a second limitation of the classic measuring system, namely that the latter does not define any larger divisions beyond the measure level. With relative units we can define both microscopic and macroscopic temporal structures. The relative time units available in the system are shown in the lower part of Table 4.1.

Definitions: $\mathit{tempo}, \mathit{beats}, \mathit{beatType}$ We define the canonical attributes *tempo*, *beats*, and *beatType* for holding the three parameters T, N, and B. We will reuse the *span* attribute for the duration parameter Δ. While some factors in Table 4.1 can be defined as simple numeric constants, such as ‘minute’, others depend on one or more parameters. As an example, we could implement the ‘beat’ unit with a function $\mathit{beat} : V_\lambda \to (V \to \mathbb{R}) : u \mapsto v \mapsto \mathit{minute}(u(v))/(\mathit{tempo}_{or}(120))(u(v))$. Note that instead of evaluating the expressions directly on the input entity v, we apply them to the entity $u(v)$. This form allows us to specify a parameter implicitly as an inherited attribute, or explicitly as an attribute on u. For example, if we write $\mathit{span}_{set}(\mathit{beat})$, the setter will treat *beat* like a

unit	factor	base
second	1	second
minute	60	second
hour	60	minute
beat	$1/T$	minute
measure	N	beat
whole	B	beat
half	$1/2$	whole
quarter	$1/4$	whole
eighth	$1/8$	whole
sixteenth	$1/16$	whole
relative	Δ	second
percent	$1/100$	relative

Table 4.1: Conversion table for the temporal units. Reading: *A minute is equal to 60 seconds.* The computation of the conversion factors for the absolute units relies on three parameters: the tempo T, the number of beats per measure N, and the beat type B. The computation of relative units relies on a duration Δ.

second-order expression, apply it twice to v, and use either its *tempo* attribute, or the fallback value 120. If we instead write $\mathit{span}_{\mathit{set}}(\mathit{beat}(\mathit{tempo}_{\mathit{set}}(60)))$, then the new entity $(\mathit{tempo}_{\mathit{set}}(60))(v)$ will be used to supply the tempo. The above recipe for the 'beat' unit works analogously for the other units. For N and B we assume the fallback value 4.

Example: Meter

In this example we extend the previous example with our new units. We start by globally setting a slow tempo of one beat per second in p_1 and a three-quarter time signature in p_2 and p_3. Each measure has three beats, so the *measure* expression in p_4 evaluates to 3 seconds. In p_5 we fork into the bass, chord, and melody voices. For the bass note, we use the identity function. Here it would be equivalent, yet redundant, to write $\mathit{span}_{\mathit{set}}(\mathit{rel})$ instead. In the CHORD and MELODY sub-grammars we express the delay and duration of the entities in quarters and half notes, rather than absolute numbers.

$$
\begin{aligned}
&\text{PIECE} : \langle \\
&\quad p_1 : tempo_{set}(60) \\
&\quad p_2 : beats_{set}(3) \\
&\quad p_3 : beatType_{set}(4) \\
&\quad p_4 : span_{set}(measure) \\
&\quad p_5 : [\langle\rangle, \text{CHORD}, \text{MELODY}] \\
&\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\text{CHORD} : \langle \\
&\quad q_1 : time_{set}(time + quarter) \\
&\quad q_2 : span_{set}(half) \\
&\quad q_3 : range(3) \\
&\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\text{MELODY} : \langle \\
&\quad r_1 : span_{set}(quarter) \\
&\quad r_2 : [time_{set}(time + quarter), time_{set}(time + half)] \\
&\rangle
\end{aligned}
$$

4.1.2 Split

The *split* operation allows us to divide a given volume into an arrangement of sub-volumes, in our case a time interval into a sequence of sub-intervals. These sub-volumes must not overlap, and their bounds are constrained to coincide—hence the name. In the previous example we modified the *time* and *span* attributes directly to define the temporal scope of an entity relative to that of its parent. The split operation only needs to know the length of the parts and computes the absolute bounds as a cumulative sum relative to the bounds of the available space. Our general exploration of the time dimension in Section 4.1 already implied that a split operation can be useful for music generation. It is the missing sequential operation in the parallel-sequential duality in our model of a polyphonic structure. As opposed to parallel arrangements, which are trivial to generate due to attribute inheritance, the split operation needs further definition.

Let us first explore concrete ways in which we can use a split operation in music. We can start with a piece of a certain length and generate an overall structure, such as an intro measured in seconds, a sequence of equal-length themes, and an outro that is half the length of a theme. At a lower level, we can split themes into hyper-measures and measures. Below the measure level, we can use uniform splits with a division number of two and three to generate a metric structure. Rhythmic patterns can be generated with relative sizes, for example a $(1/4, 2/4)$ pattern in a three-quarter meter. Or we could specify the same pattern just with the numerators $(1, 2)$ and let the split operation infer the three-part division from their sum. For splitting the measure of an arbitrary meter, we could define a one-quarter part and a flexible part, which will fill the remaining space. From these use-cases we can deduce some basic requirements for the split operation. There can be an arbitrary number of parts, and their sizes can be defined as a fixed or flexible quantity. Fixed sizes are useful when we want to fit the parts onto our metric grid, for example when splitting a measure into quarter notes. Flexible sizes are useful for distributing remaining space. We must also be able to apply rules to the resulting parts individually. For example, we may need to mark one part as an 'intro' and another as an 'outro'.

A particular split is defined by a *scope*, which is the available volume, and a *pattern*, which describes how the available volume is divided. We define the scope as a pair of numbers $(t_0, t_1) \in \mathbb{R}^2$ and its size $\Delta = t_1 - t_0$. From the pattern we derive a sequence of sizes $(\Delta_1, \ldots, \Delta_n)$ for the resulting sub-volumes. We calculate the cumulative sum of these sizes, starting at the scope's first bound t_0, to get the split offsets $(t_0 = s_0, \ldots, s_n)$. The final result are the scopes for the adjacent scopes for the sub-volumes $((s_0, s_1), \ldots, (s_{n-1}, s_n)) \in (\mathbb{R}^2)^n$. For example, if the sizes for the sub-volumes are $(2, 3)$ and the scope is $(1, 8)$, the split offsets are $(1, 3, 6)$ and the resulting volumes are $((1, 3), (3, 6))$. Under this definition the sub-volumes only align with the first bound t_0 when the sum of their sizes is equal to Δ. In order to align the sub-volumes with the second bound t_1, we could stretch them, align them to either side, to the center, or an arbitrary pivot point, resize, duplicate, drop, insert, or redistribute individual elements, or truncate the overflowing volumes. Yet, we do not have to include this in the split model since these strategies can be implemented as stand-alone operators on the split result.

We define the split pattern as a sequence of number pairs $((\delta_1, \omega_1), \ldots, (\delta_n, \omega_n)) \in (\mathbb{R}_+^2)^n$, where δ_i is the *fixed size* and ω_i is the *flexible size* of the ith element. We calculate the total fixed size $\delta = \delta_1 + \ldots + \delta_n$, the total flexible size $\omega = \omega_1 + \ldots + \omega_n$, the remaining space $\omega_\Delta = \max\{0, \Delta - \delta\}$, and the *flexible unit* $\omega_U = \omega_\Delta / \omega$ (0 if $\omega = 0$). The final size of a sub-volume can now be calculated as $\Delta_i = \delta_i + \omega_i \omega_U$. What this means is that the fixed size is the minimum size of the resulting part, while the flexible size is a relative weight that is used to distribute the remaining space. If there is no remaining space, the contribution of the flexible size is zero. For example, the pattern $((1, 1), (0, 2))$ applied to the scope $(1, 8)$ yields $\delta = 1$, $\omega = 3$, $\omega_U = (7 - 1)/3 = 2$, and consequently two resulting volumes $((1, 4), (4, 8))$. If the scope were $(1, 2)$, no remaining space would be available and the result would be $((1, 2), (2, 2))$, where the second volume is empty.

Definitions: $split, flex, size$ We integrate the split operation into our grammar with a function $split : V_\lambda^* \to (V \to V^*)$, which takes the split pattern as a parameter and yields a sentence expression that generates the resulting entities. We will use the canonical *time* and *span* attributes of the input entity as the scope and also use them for the scope of the sub-volumes. Defining the split pattern as a sentence is useful as we can generate it using a fully-featured sub-grammar. The split operation is just a particular context-sensitive transformation on a sequence of entities. We define two new canonical attributes: $size$ for the fixed size δ_i, and $flex$ for the flexible size ω_i. Per default, if neither $size$ nor $flex$ are defined, the part will be treated as if it had a $flex$ value of one. This means a uniform split into n parts can be expressed as $\mathit{split}(\mathit{range}(n))$ because $\omega_\Delta = \Delta$ and the final length of each part will be Δ/n. When only one of the two attributes is defined, the other attribute will be treated as if it were zero. Any attributes on the entities in the pattern will be preserved, except for *time* and *span*, which will be overwritten, and $flex$ and $size$, which will be discarded. We can use *append* and *select* to integrate the split result into the sentence, as discussed in Section 3.4.3.

Example: Split

In this example, we will again replicate the temporal structure of a part of *Gymnopedie No.1*, shown as a score in Figure 4.1. With the split rule we are now able to concisely replicate the four steps shown in Figure 4.3. Rules p_1 to p_4 generate a four measure interval with a three-quarter meter. In p_5 we split this interval into four equal parts and store their indices in the m attribute. This corresponds to step (1) in the figure, with the difference that we are generating four measures instead of just two. Rule p_6 models the parallel separation of the three voices. Note that the score contains quarter rests for the melody and chord part. There are various ways of handling rests. For example, we could mark the note as silent, or we could simply delete the entity using an empty fork []. Here we use the *rest* attribute as a valueless marker.

Once again, we use the identity function for the bass note, as it just needs to inherit the scope. The CHORD sub-grammar splits the measure into a rest with the length of a quarter, and an entity with the valueless *chord* attribute. Since the *chord* entity has no size, it will behave as if we set $flex$ to one and fill the remaining space of the measure. In rule q_2 we generate three parallel entities for every part that was marked with *chord*, but not for those marked with *rest*. The melody consists of a sequence of quarter notes, except for the last measure, which we exclude via $m \neq 3$ in r_1. All other measures are split into three equal size notes numbered with the i attribute, before being passed to the second rule. Note that the pipe terminates when one LHS fails, so if we want the derivation to continue in r_2, r_1 must be wrapped in its own pipe in order to treat it as an optional step, rather than a termination criterion. The second rule only applies to the first note of the first measure, which we mark as a rest.

$$
\begin{aligned}
&\text{PIECE} : \langle \\
&\quad p_1 : \mathit{tempo}_{set}(60) \\
&\quad p_2 : \mathit{beats}_{set}(3) \\
&\quad p_3 : \mathit{beatType}_{set}(4) \\
&\quad p_4 : \mathit{span}_{set}(4 * \mathit{measure}) \\
&\quad p_5 : \mathit{split}(\mathit{range}(4, m_{set})) \\
&\quad p_6 : [\langle\rangle, \text{CHORDS}, \text{MELODY}] \\
&\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\text{CHORDS} : \langle \\
&\quad q_1 : \mathit{split}([\langle \mathit{rest}_{tag}, \mathit{size}_{set}(\mathit{quarter})\rangle, \mathit{chord}_{tag}]) \\
&\quad q_2 : \mathit{chord} \longrightarrow \mathit{range}(3) \\
&\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\text{MELODY} : \langle \\
&\quad r_1 : \langle m \neq 3 \longrightarrow \mathit{split}(\mathit{range}(3, i_{set}))\rangle \\
&\quad r_2 : i = 0 \wedge m = 0 \longrightarrow \mathit{rest}_{tag} \\
&\rangle
\end{aligned}
$$

4.1.3 Repeat

Assume that we want to split an entity into a sequence of measures. In the previous example we knew that the length of the piece was four measures, and consequently we could just use a uniform split with four parts. If we already know the desired number of measures N, we can define a split pattern $((m, 0), \ldots, (m, 0))$ with N elements, where m is the constant size of a measure, and ignore the size of the parent scope Δ altogether. But what if we want to define a rule that fills an entity with an arbitrary size? Of course, we can calculate the maximum number of measures that fit into the scope as $N = \lfloor \Delta/m \rfloor$. This way we can ‘tile’ an entity with equal length parts, which can be very useful when we want to generate structures inside of parts that are flexible. For example, we may want to generate beats in a measure of arbitrary time signature, or replace a chord of arbitrary length with an arpeggio of eighth notes. Since this is a common pattern, we provide it as a dedicated *repeat* model.

The repeat model generalizes the split model with an extra step at the beginning. In this step we make N copies of the pattern, concatenate them, and use this as the actual pattern for the split operation. Under this model the difference between a normal split and a repeating split is the calculation of N. For a normal split N would be 1. For a repeating split we first calculate δ, which is the minimum size of the pattern, and then $N = \lfloor \Delta/\delta \rfloor$, which yields the maximum number of times the pattern fits into the scope without exceeding it. For example, given the pattern $((4,0),(0,1))$ and $\Delta = 10$, the minimum size of the pattern is $\delta = 4$, and the number of repetitions is $N = 2$. The pattern will be repeated two times and yield four sub-volumes with sizes $(4,1,4,1)$. Compare that to the result of a split without repetition, where only two sub-volumes would be generated and their sizes would be $(4,6)$. Alternatively, we can calculate $N = \lceil \Delta/\delta \rceil$, which means the pattern will be repeated until the scope is covered completely. That way we can guarantee at least one repetition, even if $\delta > \Delta > 0$.

Definitions: $\mathit{repeat}_{\lfloor}, \mathit{repeat}_{\lceil}$ We can integrate the repeat model by defining two functions $\mathit{repeat}_{\lfloor}$ and $\mathit{repeat}_{\lceil}$ that are equivalent to *split* in all ways, except for the computation of N. In some situations we must access the repetition number N as well as the index i of each pattern instance. For example, when we tile a piece with measures, we might want to add a prelude to the first measure. The problem is that the result of the split is a flat sequence of sub-volumes, and information about the individual repetition instances is lost. We solve this here with a second parameter over the functions $\mathbb{N} \times \mathbb{N} \to F$, which we call the *indexer*. If an indexer is specified, the function will be invoked for every generated entity with the number pair (i, N), where i is the repetition index. Note that we use a similar mechanism for the *range* function that we defined in Section 3.4.4.

Definitions: *trim* Both *split* and *repeat* align the generated entities to the first bound of the temporal scope, but the pattern is not guaranteed to meet the second bound. For example, if we use $\mathit{repeat}_{\lceil}$ with a pattern that does not evenly divide the interval, the last entity will exceed the scope. This is undesirable in many situations, as it can cause monophonic voices to become polyphonic, or allow the harmony from one section to bleed into the next. We define a function $\mathit{trim} : V_{\lambda}^{*} \to (V \to V^{*})$, which clamps the scopes of the specified entities to the scope of the input entity. Entities that have an empty overlap with the original scope will be removed entirely. For example, given a scope $(0,2)$ and two entities with scopes $((1,3),(2,4))$, *trim* will shorten the first entity to $(1,2)$ and remove the second one.

Example: Repeat

In this example we will use the repeat function to define a division for entities that have an arbitrary duration. Once again, we start by setting a tempo and the length of the piece, but this time the length is a random number. In p_3 we define a split pattern that consists of a single beat followed by a random number of beats. Since we use the $\mathit{repeat}_{\lfloor}$ function, the system will calculate how many instances of this pattern fit into the randomly generated length of the piece without exceeding it. Alternatively, we could use $\mathit{repeat}_{\lceil}$ and wrap it in *trim*. Once the entities are generated they are passed to the indexer, which marks every generated part with the *arp* attribute, except for the last pattern instance. Finally, in p_4 we replace all entities with the *arp* attribute with a sequence of half beats, using the $repeat_{\lceil}$ function once again. In this case it does not matter which *repeat* variant we use, as the scope will be divided perfectly. We also do not use an indexer since we are not interested in the repetition number.

$$\begin{array}{l}
\text{PIECE} : \langle \\
\quad p_1 : tempo_{set}(60) \\
\quad p_2 : span_{set}(uniform(1,2) * minute) \\
\quad p_3 : repeat_{\lfloor}([\\
\quad\quad size_{set}(beat) \\
\quad\quad size_{set}(choice(1,2,3) * beat) \\
\quad], (i, N) \mapsto \langle i \neq N \longrightarrow arp_{tag} \rangle) \\
\quad p_4 : arp \longrightarrow repeat_{\lceil}(span_{set}(beat/2)) \\
\rangle
\end{array}$$

4.1.4 Query

We use the term *query* to describe a function that extracts information from a sentence. We have already discussed some of them in Section 3.4.3. For example, the $filter$ query returns only those entities in a sentence that match a certain predicate. In this section we will discuss *temporal queries*, which select entities based on their location on the timeline. With temporal queries we can treat the value of an attribute as a time-dependent function. For example, we can generate a harmonic progression and store it in an attribute. We can then fork into multiple voices, which we develop with independent sub-grammars. In a final pass, we can combine the independent rhythmic and melodic information of the two voices with the common harmony of the corresponding temporal segment in the progression and calculate a pitch for each note.

Definitions: *query* We define a temporal query as a function that takes three parameters: a *texture*, which is a sequence of intervals that should be queried, a *scope*, which defines a location on the texture, and a *binary relation* between intervals, which determines whether an interval in the texture is selected by the scope. We will only use the overlap relation between intervals. For example, given a texture $((0,2),(2,3),(2,5))$, a scope $(2,4)$, the query function will yield the intervals $((2,3),(2,5))$. Of course, the intervals on their own do not provide much information. In practice they will be associated with additional data, for example an entity. For integrating temporal queries into our system, we once again rely on the *time* and *span* attributes for encoding the intervals of both the scope and the texture elements. We define the function $query : V_\lambda^* \to (V \to V^*)$, where the parameter is a sequence of entities that is used as the texture. The resulting sentence expression returns only those entities in the texture that overlap with the scope of the input entity.

4.1.5 Example: Query

In this example we will generate a harmonic progression, store it in an attribute, and later query it from three rhythmically independent voices. First, let us look at the generation of the harmonic progression in the HARMONIC sub-grammar. Within the scope of the input entity, the rule generates a repeated sequence of two measures, where the first has the subdominant (IV) as its harmony, and the second has the tonic (I). This is similar to the alternating harmonic pattern in the first sixteen measures of *Gymnopédie No.1*. We assume that the meter, tempo, and scope are already defined for the main grammar. We start in p_1 by generating the harmonic progression in the *texture* attribute. Rule p_2 forks into our three voices. We do not care how the voices are defined, but assume that each generates an independent rhythmic division of the piece. Finally, in p_3 we use *query* to sample the *texture* attribute and append the *harmony* attribute to the entities in each voice. Note that if an entity in one of the voices overlaps with multiple measures, *query* will return multiple entities. If this is not desired, we can wrap the query into an additional operator. For example, we could randomly select one of the results using *choice*.

```
$\text{HARMONIC} : repeat_{\lfloor}(\langle$
    $size_{set}(measure)$
    $[harmony_{set}(IV), harmony_{set}(I)]$
$\rangle)$
```

```
$\text{PIECE} : \langle$
    $p_1 : texture_{set}(\text{HARMONIC})$
    $p_2 : [\text{BASS}, \text{CHORD}, \text{MELODY}]$
    $p_3 : append(\langle$
        $query(texture)$
        $select(\{harmony\})$
    $\rangle)$
$\rangle$
```

4.2 Pitch

Beyond a certain number of beats per time interval, around 20 per second for humans, we can no longer perceive them individually. We instead describe the experience as hearing a sound of a certain frequency. While only some people can accurately identify absolute frequencies, most people are able to differentiate between frequency ratios. Just as with rhythm, the use of simple frequency ratios leads to harmony. After the unison, a ratio of 1:1, the second most harmonic ratio is the octave at 2:1. Two frequencies separated by octaves sound so similar, that in theory we can sometimes ignore the octave separation entirely. This phenomenon is known as *octave equivalence*, and we refer to such an equivalence class as a *pitch class* or *chroma*. Higher numbered ratios, such as 2:3 for the perfect fifth, or 3:4 for the major third, gave rise to the twelve pitch classes commonly considered in western music theory.

For polyphony the particular sounds are not important, but we want at least to be able to use basic harmonic rules to relate the frequencies of multiple voices. In musical notation a note is associated with a single scalar attribute called its *fundamental frequency* or *pitch*, which is a reduction of a complex instrument sound to the most characteristic element of its spectrum. The *timbre*, which is the particular configuration of overtones generated by an instrument playing the note, is lost in this reduction. Our system uses the same approach and associates each terminal with exactly one pitch value. When discussing the time domain we briefly mentioned that we could use multiple pitches per entity to represent chords, but we have since shown that we can model chords with multiple parallel entities. A one-to-one mapping between entities and notes is sufficient.

Definitions: $freq, play, gain$ We define the canonical $freq$ attribute which represents the fundamental frequency of the entity in Hertz. For example, $freq_{set}(440)$ sets the frequency to A4 in standard tuning. Timbre is not relevant for our approach, so we mainly consider it a nominal quality for differentiating between voices. Yet, for the evaluation of our system we did implement a playback environment with additional parameters for fitting sounds to the abstract note representation. Various timbres may be provided as virtual instruments that map a set of frequency values to particular audio samples. We use the *play* attribute for selecting an instrument by name, and the $freq$ attribute for selecting the closest audio sample, with optional fine-tuning if the selected sample does not match the frequency. Additionally, we define the *gain* attribute for controlling the relative loudness of a note, where 0 is complete silence and 1 is an arbitrary, non-silent baseline. The amplitude of a signal can actually be considered a dimension on its own, but it does not warrant a dedicated discussion in the context of this thesis.

Example: Pitch

In this example we will apply our simple pitch model to generate a sound. For the *play* attribute we set the value “sine” which indicates that we want to generate simple sine waves, rather than the sound of a prerecorded instrument. In p_3 we generate a number of entities as indicated by the *overtones* attribute, for which we defined a fallback value in p_2. We use the generated index i to calculate the harmonic overtones of the current frequency. For example, if the number of overtones is eight, and the frequency is 440, which is A4 in scientific pitch notation, it will generate the harmonic overtones $(440, 880, 1320, 1760, \ldots, 3520)$, which correspond to $(A4, A5, E5, A6, \ldots, A7)$. Note that when the factor is a power of two, the resulting frequency has the same pitch class as the fundamental frequency. Finally, in p_5 we set the gain to the reverse index and scale it by the number of overtones. The 0th overtone, which is the fundamental frequency, will receive a *gain* of one, and the gain decreases with increasing frequency.

$$
\begin{aligned}
&\text{OVERTONES} : \langle\\
&\quad p_1 : play_{set}(\text{“}sine\text{”})\\
&\quad p_2 : overtones_{def}(8)\\
&\quad p_3 : range(overtones_{or}(8), i_{set})\\
&\quad p_4 : freq_{set}((i + 1) * freq_{or}(440))\\
&\quad p_5 : gain_{set}((overtones - i)/overtones)\\
&\rangle
\end{aligned}
$$

4.2.1 Key

Western theory of harmony commonly involves *scales*, which are particular sets of pitches. A *diatonic scale* is a scale that contains seven of the twelve pitch classes of the *chromatic scale* and is constrained in such a way that the notes are maximally spread out across the octave. We can describe this as a pattern of two *half step* intervals (one semitone) and five *whole step* intervals (two semitones), where the two half steps are separated by a minimum of two whole steps. The major scale, as one particular example, is constructed from two whole steps, one half step, three whole steps, and one half step to complete the octave. We can start this pattern from any of the twelve pitch classes, leading to a total of twelve possible major scales. We assume *equal temperament* with twelve equally spaced pitch classes within an octave, which is the predominant tuning system used in western music. This means we will regard enharmonic notes, such as *C#* and *Db*, as identical.

In this section we will define a model for calculating a frequency given any combination of key, mode, octave, harmonic function, scale degree, and accidental. The model is defined by a numeric formula that can be separated into four components: (1) an offset $d \in \mathbb{Z}$ within the seven-tone diatonic scale, (2) a discontinuous function $mode_M \in \mathbb{Z} \to \mathbb{Z}$ from the diatonic scale to the twelve-tone chromatic scale, (3) an offset $c \in \mathbb{R}$ within the chromatic scale, and (4) an exponential function $hertz : \mathbb{R} \to \mathbb{R}$ from the chromatic scale to physical frequency. The chromatic scale is a logarithmic representation of pitch, where each whole number step is equivalent to a semitone interval. The origin of this system is arbitrary, and we use the MIDI standard of assigning A4 (440 hz) to 69. We can combine these terms into a frequency f_0 using the following formula:

$$f_0 = hertz(c + mode_M(d))$$

The term d consists of two offsets within the coordinate system of the diatonic scale. The first offset is defined by the harmony parameter $H \in \mathbb{Z}$, which represents a harmonic function. It is zero for the tonic (I), one for the supertonic (II), two for the mediant (III), and so forth. To the harmonic offset we add the degree parameter $D \in \mathbb{Z}$ for selecting a scale degree relative to the harmony. Again, we assume a value of zero is the first scale degree (I), one is the second (II), etc. Both parameters are defined over $\mathbb{Z}$ since a fractional degree or harmony is not musically meaningful. This system allows us to specify offsets within a scale as the nth degree of the mth harmonic function. For example, the seventh degree (VII) of the tonic function (I) and the third degree (III) of the dominant function (V) are both equal to an offset of 6 (VII). This is not particularly complicated and can be defined as a sum:

$$d = H + D$$

The function $mode_M$ is more complex as it must perform a discontinuous mapping from the diatonic scale to the chromatic scale. A major scale can be defined by a pattern $(a_i)_{i=0} = (2, 2, 1, 2, 2, 2, 1)$ of semitone intervals, and by rotating this pattern we get the seven diatonic *modes*. We define the function $a(i \in \mathbb{Z}) = a_{i \bmod 7}$ to extend the pattern infinitely in both directions and use the mode parameter $M \in \mathbb{Z}$ as the origin within this extended pattern. We use roman numeral notation, where the major mode is canonically the first mode with $M = 0$ (I). For example, $M = 5$ (VI) starts the pattern at the sixth element and yields the rotation $(2, 1, 2, 2, 1, 2, 2)$, which is called the *Aeolian* or minor mode (VI). The input to $mode_M$ is an offset $z \in \mathbb{Z}$ in scale degrees, for example the term d defined above. For a positive diatonic scale degree z we can calculate the number of semitones using $a(M) + a(M + 1) + \ldots + a(M + z - 1)$. For example, the semitones of a third (III) in the minor mode (VI) can be calculated with $mode_{VI}(III) = mode_5(2) = a(5) + a(6) = 2 + 1 = 3$.

The above formula is not yet defined for negative values of z. Generally, we want the value $mode_M(z)$ to be equivalent to the number of semitones of z scale degrees from the tonic in mode M. For example, $mode_I{-III}$ is a diatonic third below the major tonic, which is equivalent to three semitones. For this we separate z into the modulo $p(z) = (z \bmod 7)$ and quotient $q(z) = \lfloor z/7 \rfloor$, where inversely $z = 7 * q(z) + p(z)$ for all $z \in \mathbb{Z}$. The value of $p(z)$ is the offset within the current rotation of the pattern, $q(z)$ is the number of complete rotations, and 7 is the period of the diatonic scale. The total semitone offset can be calculated as $12 * q(z) + a(M) + a(M+1) + \ldots a(M + p(z) - 1)$, where 12 is the period of the chromatic scale. For $M = 0$ and $z = -2$ this yields $p(z) = 5$, $q(z) = -1$, and a total semitone offset of $a(0) + \ldots + a(4) - 12 = 2 + 2 + 1 + 2 + 2 - 12 = -3$, a negative minor third. The general formula for $z \in \mathbb{Z}$ looks like this:

$$mode_M(z \in \mathbb{Z}) = 12 * q(z) + \sum_{i=0}^{p(z)} a(M+i)$$

For mapping intervals on the chromatic scale to physical frequency, we need to define an origin. In musical notation, a *clef* is used to associate the lines in a stave with an absolute pitch, but we may just use a semitone offset c from some arbitrary base frequency. As mentioned above, we use the MIDI convention of associating A4 with 69. For example, one semitone above A4 is $A\#4$ (70), and nine semitones below is C4 (60). We express c via three parameters. The octave parameter $O \in \mathbb{Z}$ is measured in twelve-semitone intervals, which means that incrementing O by one increases the offset by 12. Since MIDI number zero counter-intuitively corresponds to C-1 (negative one) we shift O up by one to compensate. For example, $O = 4$ maps to a semitone offset of $12 * (4 + 1) = 60$, which is equal to C4. In addition, we define the key parameter $K \in \mathbb{Z}$ and the alter parameter $A \in \mathbb{Z}$, which are offsets measured in plain semitones. For example, with $O = 4$, $K = 7$, and $A = 1$ we get $c = 60 + 7 + 1 = 68$, which corresponds to G#4. We use the general formula below:

$$c = 12 * (O + 1) + K + A$$

Finally, we define the *hertz* function, which maps a MIDI number to its frequency. There is not a lot to discuss, as this step adds no further parameters to our model. For some use-cases a conversion to physical frequency might not even be necessary, for example, if we want to generate MIDI output. The constant 2 in the formula below stands for the exponential doubling of the frequency for every octave of 12 semitones, and the constant 69 is the MIDI number of A4 at 440 Hz:

$$hertz(x \in \mathbb{R}) = 440 * 2^{(x-69)/12}$$

In conclusion our diatonic model has six whole-numbered parameters $O, K, A, M, H, D \in \mathbb{Z}$. From a theoretic perspective, the parameters O, K, A may also be defined for real numbers $\mathbb{R}$, but fractional values are only meaningful for the alter parameter A, which we can use to slightly change the tuning of a note. The pairs (K, A) and (H, D) can seem redundant since they are just added together. Yet, from a musical perspective the separation does make sense, as they cover different semantic use-cases. For example, we may define the key K globally for the piece and alter a note locally using A. Or we might generate a progression of harmonies with H and develop a melody within the harmony using the D parameter. It is only in our particular model that these heterogeneous concepts become interchangeable. For example, had we assumed *just intonation* instead of equal temperament, enharmonic notes would no longer be equivalent, and a semitone offset for modelling accidentals would be insufficient.

Definitions: *diatonic*, *octave*, *key*, *alter*, *mode*, *harmony*, *degree* We will now integrate the presented model into our meta-language by defining six canonical attributes: *octave*, *key*, *alter*, *mode*, *harmony*, *degree*. For each we assume a fallback value of zero, except for *octave*, which is 4 by default. The choice of a default octave is rather arbitrary, but it is optimally in a clearly audible range. With these canonical attributes we define the parametric expression $diatonic : V_\lambda \to (V \to \mathbb{R})$, where the parameter is an entity expression V_λ that carries a subset of the attributes above, and the result is a frequency in $\mathbb{R}$ as described by our model. Again, the use of an entity expression as an argument allows us to set the parameters explicitly or implicitly. For example, we can write $freq_{set}(diatonic(degree_{set}(2)))$ to use the second degree, or write $freq_{set}(diatonic)$ to read the degree from the input entity or use the fallback value.

We can further improve the usability of the model with predefined musical constants. As mentioned above, scale degrees, harmonic functions, and modes are often identified by roman numerals. We use a zero-based mapping, where I is equal to zero and II to one since the first degree is equivalent to an offset of zero, while music notation usually counts from one. We also provide constants for the names of the harmonic functions (e.g. tonic, subdominant, dominant), modes (e.g. major, minor, ionian, dorian), accidentals (e.g. natural, flat, b, bb), semitone intervals (e.g. M3, P4, P5), pitch classes (e.g. C, C#, Db, D), octaves (e.g. middle, bass), and absolute pitches (e.g. A4, Bb5, C#6).

Example: Key

In this example we want to play ascending scales for every diatonic mode and every pitch class. Rule p_1, p_2, and p_3 are very similar and generate numbered clones. We use these three levels of indices to assign three diatonic attributes, first the *key* in p_1, then the *mode* in p_2, and finally the *degree* in p_3. In p_4 we compute the $freq$ using the *diatonic* function. The *alter*, *harmony*, and *octave* are not specified and their default values will be used. We use our split function to arrange the notes as a sequence.

$$
\begin{aligned}
&\text{SCALES} : split(\langle \\
&\quad p_1 : range(12, key_{set}) \\
&\quad p_2 : range(7, mode_{set}) \\
&\quad p_3 : range(8, degree_{set}) \\
&\quad p_4 : freq_{set}(diatonic) \\
&\rangle)
\end{aligned}
$$

4.3 Example

Now that we have the abstract theoretic tools and domain-specific models for time and pitch at our disposal, we can transition to their practical application. In this section we will demonstrate the generation of motifs, high-level piece layouts, and chord progressions, culminating in the definition of a complete, polyphonic composition based on these components. The resulting pieces will consist of multiple voices, each with a unique layout that divides the duration of the piece into a sequence of measures. In each measure we query a shared chord progression that we generate beforehand, which synchronizes the voices to a common harmonic context. The application of our polyphonic model shows that we can decouple voices and abstract musical aspects from each other, while not compromising on their synchronization and the quality of the compositions.

Our way of constructing the example in this section—starting at lower-level grammars and progressively integrating them into higher-level ones—goes in the opposite direction of the derivation process. The reason is that the lower-level grammars are more closely related to the generated music and thus easier to understand. The separate definition of these sub-grammars should further demonstrate that they are easy to reuse, recombine, and replace when we are not satisfied with the results. Transparency and loose coupling are after all primary advantages of grammars over other methodologies. A compact version of the code in this chapter, as well as a translation to TypeScript, can be found in Appendix A. The audio files are attached to the digital version of this thesis and also available online: `https://github.com/eibens/thesis-2021`

4.3.1 Motifs

We start by defining sub-grammars that generate motifs, which is the first level of abstraction above the note level. A motif is a short arrangement of sounds, no longer than a measure, which fits into the metric and harmonic framework, but usually has no further internal organization. Within a motif we may access metric and harmonic attributes such as *key*, *mode*, *harmony*, *tempo*, *beats*, and *beatType*, but we will avoid changing them, as the motif should not deviate from its context. Attributes that we will change are *time*, *span*, *gain*, and *freq*, as long as these changes take the context into consideration. For example, we can use the metric units for measures, beats, and note fractions, but should avoid absolute and relative splits, as they are not aligned with the metric structure. We will define four different motif generators: BASS, PAD, LEAD, and DRUM.

Example: Helpers

Before we implement the actual motif generators, we define several helper functions that simplify their definition. CUT splits the entity into two parts (A, B) or (B, A), where A has a fixed size s and B fills the remaining space. Besides their temporal scope, both parts are equivalent. We additionally trim the result, in order to guarantee that it fits into the input entity. REP repeats a part of a fixed size s until the input entity is covered and assigns the index to the *index* attribute. Again, we trim the result. DEG1 and DEG2 assign a random value to the *degree* attribute. DEG1 only picks from the most important scale degrees: the tonic, third, fifth, and octave. DEG2 picks from the remaining degrees. Finally, PR can be used within a pipe to apply an optional step f with probability p. It will be used throughout this section.

$$
\begin{aligned}
&\text{CUT} : (s \in \mathbb{R}_\lambda) \mapsto trim(choice(split([size_{set}(s), \langle\rangle]), split([\langle\rangle, size_{set}(s)))))\\
&\text{REP} : (s \in \mathbb{R}_\lambda) \mapsto trim(repeat_\lceil(size_{set}(s), index_{set}))\\
&\text{DEG1} : degree_{set}(choice(I, III, V, VIII))\\
&\text{DEG2} : degree_{set}(choice(II, IV, VI, VII))\\
&\text{PR} : (p \in [0,1]_\lambda, f : V_\lambda^*) \mapsto \langle rand \leq p \longrightarrow f\rangle
\end{aligned}
$$

Example: Bass Motif

BASS generates notes for a bass line and is the first and simplest of our four motif generators. We start by repeating a random time interval. Since the primary purpose of the bass is to communicate the fundamental frequency, we will only change the degree if it is not the first note of the motif. Even then we only pick from the most important scale degrees as provided by DEG1. In the last two steps we select a random octave and calculate the corresponding frequency. A subset of the possible results is shown in Figure 4.4.

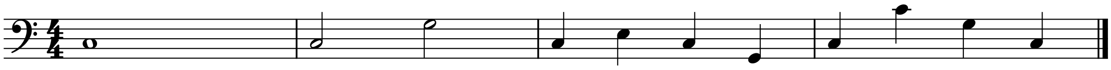


Figure 4.4: Each measure shows a possible result of the BASS grammar (File: bass.mp3). We normalized the results to C-major and common time.

$$
\begin{aligned}
&\mathrm{BASS} : \langle \\
&\quad \mathrm{REP}(\mathit{choice}(\mathit{measure}, \mathit{whole}, \mathit{half}, \mathit{quarter})) \\
&\quad \mathit{index} > 0 \longrightarrow \mathrm{DEG1} \\
&\quad \mathit{octave}_{\mathit{set}}(\mathit{choice}(1, 2)) \\
&\quad \mathit{freq}_{\mathit{set}}(\mathit{diatonic}) \\
&\rangle
\end{aligned}
$$

Example: Pad Motif

PAD generates chord notes in order to provide harmonic context for the piece. Per default the chord will be a triad, but can be extended to a seventh chord or higher by setting the *chord* attribute. The factor $2i$ calculates scale degrees that correspond to chord notes. For example, if $\mathit{chord} = 3$, the resulting degrees will be (I, III, V). For additional texture we randomly divide the scope using CUT, which splits off a half, quarter, or eighth note from the start or end of the interval. We apply it once before the chord note generation, which splits the whole chord, and once after, which splits each note individually. Unlike BASS, we determine the octave for the whole motif and not for each note individually. A subset of the possible results is shown in Figure 4.5.

$$
\begin{aligned}
&\mathrm{PAD} : \langle \\
&\quad \mathit{octave}_{\mathit{set}}(\mathit{choice}(3, 4)) \\
&\quad \mathrm{PR}(0.5, \mathrm{CUT}(\mathit{choice}(\mathit{half}, \mathit{quarter}, \mathit{eighth}))) \\
&\quad \mathit{range}(\mathit{chord}_{\mathit{or}}(3), i \mapsto \mathit{degree}_{\mathit{set}}(2i)) \\
&\quad \mathrm{PR}(0.5, \mathrm{CUT}(\mathit{choice}(\mathit{half}, \mathit{quarter}, \mathit{eighth}))) \\
&\quad \mathit{freq}_{\mathit{set}}(\mathit{diatonic}) \\
&\rangle
\end{aligned}
$$

Example: Lead Motif

LEAD is the most complex of the four motif grammars as it will generate the melody, which should ideally be distinct in its movement and rhythm. We start by applying

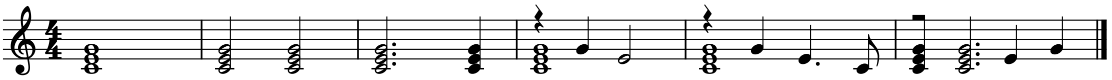


Figure 4.5: Each measure shows a possible result of the PAD grammar (File: pad.mp3). We normalized the results to C-major and common time, and use *chord* = 3. The splitting of the individual chord notes in the last three examples causes a high degree of polyphony, which is difficult to represent in a single staff.

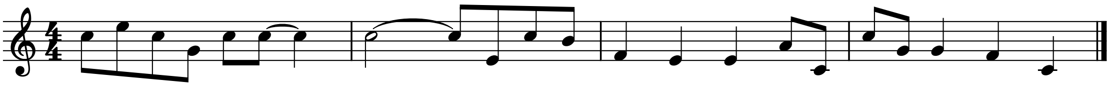


Figure 4.6: Each measure shows a possible result of the LEAD grammar lead.mp3). We normalized the results to C-major and common time.

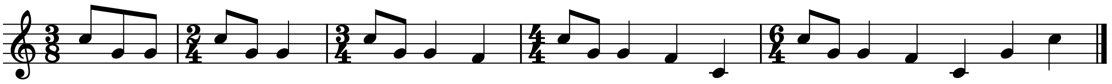


Figure 4.7: These examples show how the LEAD grammar behaves under varying metric constraints (File: lead.meter.mp3). The seed is constant for each measure.

two optional splits and assign a random degree with DEG1 to generate a preliminary structure for the motif. An optional REP further divides each note and the degree will be reassigned with a certain probability. Note that we use DEG2 with a lower probability, as it uses degrees that are less harmonic than those of DEG1. A subset of the possible results is shown in Figure 4.6 and Figure 4.7.

$$
\begin{array}{l}
\mathrm{LEAD} : \langle \\
\quad \mathrm{PR}(0.5, \mathrm{CUT}(\mathit{choice}(half, quarter, eighth)) \\
\quad \mathrm{PR}(0.5, \mathrm{CUT}(\mathit{choice}(half, quarter, eighth)) \\
\quad \mathrm{DEG1} \\
\quad \mathrm{PR}(0.5, \mathrm{REP}(\mathit{choice}(half, quarter, eighth)) \\
\quad \mathrm{PR}(0.5, \mathrm{DEG1}) \\
\quad \mathrm{PR}(0.25, \mathrm{DEG2}) \\
\quad freq_{set}(\mathit{diatonic}) \\
\rangle
\end{array}
$$

Example: Drum Motif

DRUM is our final motif grammar and generates regular pulses with a randomly selected interval. As most percussion instruments do not have pitch, we will instead modify the gain attribute to increase the variability. We scale the gain relative to the duration of

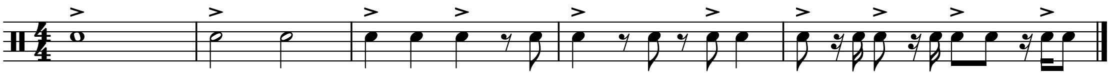


Figure 4.8: Each measure shows a possible result of the DRUM grammar (File: drum.mp3). We normalized the results to common time. The accents indicate that the notes with an even index are louder than the rest. The scaling of the loudness in relation to the duration is not depicted.

the entity and then alter the gain of every second beat to increase the fidelity. In the last step we add a random offset to beats, as long as they are equal or less than a quarter in length. A subset of the possible results is shown in Figure 4.8.

$$
\begin{aligned}
&\text{DRUM} : \langle \\
&\quad \text{REP}(\mathit{choice}(\mathit{measure}, \mathit{half}, \mathit{quarter}, \mathit{eighth})) \\
&\quad \mathit{gain}_{\mathit{set}}(\mathit{gain} * \mathit{span}/\mathit{whole}) \\
&\quad \langle \mathit{odd}(\mathit{index}) \longrightarrow \mathit{gain}_{\mathit{set}}(\mathit{gain}/2) \rangle \\
&\quad \langle \mathit{span} \leq \mathit{quarter} \longrightarrow \mathit{time}_{\mathit{set}}(\mathit{time} + \mathit{choice}(0, \mathit{span}/2)) \rangle \\
&\rangle
\end{aligned}
$$

Example: Motif Selection

Finally, we wrap the four motif generators in a single fork and use the nominal *role* attribute to select only one of the branches. We will later assign a role to each voice, which will cause them to play motifs in a consistent style. The set of roles is easily extensible with additional grammars. We can even have multiple motif grammars for a role by wrapping them in the *choice* function, for example $\mathit{choice}(\text{DRUM1}, \text{DRUM2})$. A motif, as we have defined it earlier, has no higher-level organization. Our next step is the generation of a musical layout that contains repetition. The leaves of this structure can then be filled by the MOTIF grammar.

$$
\begin{aligned}
&\text{MOTIF} : [\\
&\quad \mathit{role} = \text{'lead'} \longrightarrow \text{LEAD} \\
&\quad \mathit{role} = \text{'pad'} \longrightarrow \text{PAD} \\
&\quad \mathit{role} = \text{'bass'} \longrightarrow \text{BASS} \\
&\quad \mathit{role} = \text{'drum'} \longrightarrow \text{DRUM} \\
&]
\end{aligned}
$$

4.3.2 Layouts

The next step is the integration of the motifs into a larger structure. As the motif grammars generate a large variety of possible arrangements, repetition must be provided in the larger context. Besides the basic methods for handling repetition and variation discussed in Section 3.5, we have not yet proposed a scheme for generating the structure of a larger piece. In the context of this example we define a musical layout as a sequential arrangement of entities, where each entity is associated with a nominal *label* attribute that identifies the shape of the subtree. For example, a binary form with repeated themes can be described by four labels AABB. We will generate such patterns with a binary tree generator LAYOUT, which randomly repeats nodes using the SYNC helper function defined below. We will then synchronize multiple layouts, one for each voice, using the LAYER grammar. An optional application of the LABEL grammar allows us to select individual seeds for the motifs and generate sparse layouts by dropping measures.

Example: Probabilistic Synchronization

A simple grammar over a vocabulary of labels $\{A, B, \ldots\}$ and rules such as $\{A \longrightarrow AA, A \longrightarrow AB, \ldots\}$ could be used to generate patterns with repetitions. We consider two cases: (1) the entities on the RHS are synchronized (AA, BB), (2) or they are not synchronized (AB, BA). We isolate this choice in a helper function SYNC, which takes a synchronization probability p and a sentence expression f that generates the subtree roots. We define the attribute *sync*, which will be used as a temporary variable that holds a random number. Depending on the probability p, we will either use *sync* as the seed for each subtree, or use a random value. In the first case, the result will be completely monotonous (AAA...), and in the second case it will have no repetition at all (ABC...). SYNC on its own is not yet very useful, but this will change once we apply it on multiple levels of the derivation tree. This approach builds roughly on the synchronization workflow discussed in Section 3.5.1.

```
SYNC : $(p \in [0,1]_\lambda, f \in V_\lambda^*) \mapsto \langle$
    $sync_{set}(rand)$
    $f$
    $\langle p < sync \longrightarrow sync_{set}(rand)\rangle$
    $seed(sync)$
$\rangle$
```

Example: Recursive Layout Generation

LAYOUT recursively divides the input entity and applies SYNC at each branch point. We use the *derive* function to express a recursion and a new attribute *depth* for terminating the derivation after a certain number of levels. Consider Section 3.3.2 for details on

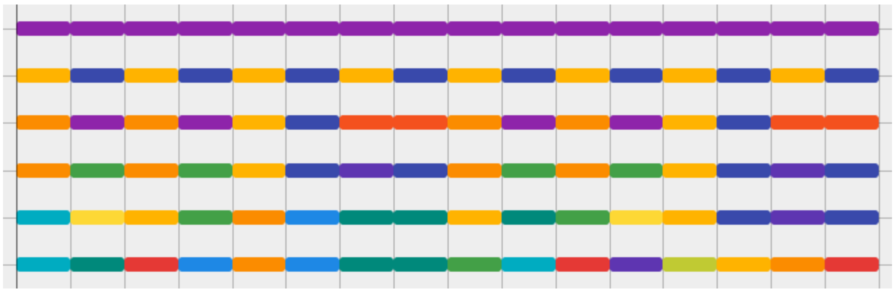


Figure 4.9: This visualization shows six generations of LAYOUT with *depth* = 4, where the monotony decreases from top (*monotony* = 1) to bottom (*monotony* = 0) in steps of 0.2. Note that the visualization is limited to 16 colors, which causes the last row to show random repetitions, even though the monotony is zero and no synchronization occurred.

recursive rules. Since we use the constant 2 for the splits, the result is a binary tree, and the total length of the pattern is 2^n, where n is the initial depth. The *monotony* attribute can be used to control the synchronization probability, and its effect is visualized in Figure 4.9. For example, with an initial depth of two, the rule will be applied once at the root and once for each of the resulting parts, yielding four entities in total. If only the first split is synchronized, the grammar generates ABAB. If all but the first split get synchronized, the pattern is AABB. With LAYOUT we can already generate interesting pieces of variable length by simply feeding its result to the motif generators. For this example we assume f is the identity function $\langle\rangle$. In the next example we will use the f parameter to further modify the split results, allowing us to synchronize layouts.

```
$\text{LAYOUT} : (f \in V_\lambda^*) \mapsto \mathit{derive}($
    $\mathit{depth}_! > 0 \longrightarrow \langle$
        $\mathit{depth}^!_{set}(\mathit{depth} - 1)$
        $\text{SYNC}(\mathit{monotony}_{or}(0), \mathit{split}(\mathit{range}(2)))$
        $f$
    $\rangle$
$)$
```

Example: Synchronization between Layouts

While LAYOUT could be used to generate the structure of a piece, the use of a single layout will synchronize the motifs across all voices. This can be desirable. For example, in a repeated binary form with the layout AABB we generate the A motif twice in all voices, then the B motif twice in all voices, leading to two pairs of perfect repetitions. But for the sake of variability it would be interesting if one voice could play AABB, another

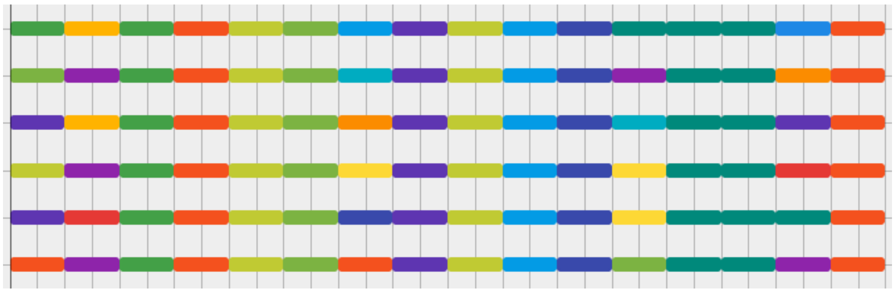


Figure 4.10: This visualization shows six generations of LAYER with $depth = 4$, $diversity = 0.25$, and a different value for the *layer* attribute in each row.

ABAB, and yet another AAAB. This can be achieved by generating a layout for each voice individually. Then again, this will lead to the voices to be perfectly desynchronized, which is interesting but may sound chaotic over longer pieces.

As a generalization of the two strategies, we define the LAYER grammar, which allows us to gradually adjust the synchronization between multiple layouts. We assume that the *layer* attribute contains a number that uniquely identifies the layout, or alternatively use a random number. Only then do we synchronize the generation of the layout using a global seed stored in the *piece* attribute. We define the *diversity* attribute, which is the probability that one branch of a layout gets desynchronized from other instances of LAYOUT that were generated with the same *piece* value. The idea is that we can generate the layout individually for each voice, but use the same *piece* value for all of them. With a diversity of zero, the result is effectively the same as if we had generated a single layout for all voices. If the diversity is one, all voices will have a unique layout, as only the unique seed in *layer* will be used. The effect of the *diversity* attribute is visualized in Figure 4.10.

$$
\begin{aligned}
&\text{LAYER} : \langle \\
&\quad layer_{or}(rand) \\
&\quad seed(piece) \\
&\quad \text{LAYOUT}(\text{PR}(diversity, seed(rand + layer))) \\
&\rangle
\end{aligned}
$$

Example: Layout Transfer Function

At last, we use LABEL as a transfer function for mapping the random numbers generated by LAYOUT to a set of predefined labels. To achieve this we set the *label* attribute with a random element from a numeric sequence stored in the *labels* attribute. This gives us greater control over the resulting layout, as we can change individual labels, while keeping everything else as it was. Another aspect that we control with the labels is

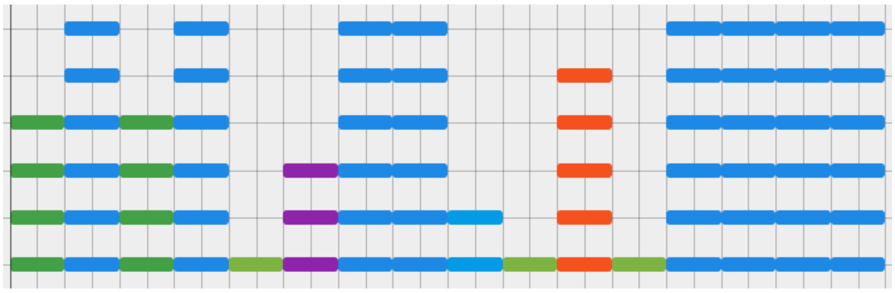


Figure 4.11: This visualization shows six generations of LAYER with *depth* $= 4$, using *labels* $= (1,0,0,0,0,0)$ for the first instance, *labels* $= (1,2,0,0,0,0)$ for the second, up to *labels* $= (1,2,3,4,5,6)$ for the last one. The sparsity of the layer increases with the number of zeros in the label pool.

sparsity. A voice does not have to cover the whole piece. In fact, the piece will sound much more interesting if certain voices only appear during certain parts of the piece. We use the label 0 as a marker for entities that should be removed. For example, if *labels* $= (0,1,2)$, on average a third of the entities will be removed. The effect of various settings for *labels* is shown in Figure 4.11. As a final step, we set the seed by combining *label* with the global *piece* seed. It is important that we add a global source of entropy, otherwise we will restrict the results to the number of unique values in the label pool.

```
LABEL : ⟨
    $label_{set}(choice(labels))$
    $\langle label = 0 \longrightarrow [] \rangle$
    $seed(piece + label)$
⟩
```

4.3.3 Chords

With the MOTIF and LAYER grammars we can already generate a wealth of pieces, but they will all sound quite monotonous. What is missing is a chord progression. Pitch and harmony, while an essential aspect of music, were not the primary concern in this thesis. We do not attempt to approximate any particular harmonic practice, as has been done by Steedman [Ste96] and Rohrmeier [Roh11]. Given that both grammars are context-free we should be able to define them within our framework, but we will not attempt this here. Instead, our progression example will be rather primitive: we first generate a chord pool with the CHORDS grammar, and after generating a temporal division we assign a random chord from the pool to each entity. This process is defined in the PROGRESSION grammar. We model the harmonic information of a chord with the attributes $K_H = \{key, mode, harmony, chord\}$, where *chord* encodes the number of chord factors, for example 3 for a triad, 4 for a seventh chord, and so forth. In addition we define the *chords* attribute over the sentence V^*, in which we will store the chord pool. On the piece level we will use it to store the complete harmonic progression, so that we can query it with the CHORD grammar and use the same chords in every voice.

Example: Chord Pool Generation

CHORDS generates the chord pool. It does not matter how the chords are arranged. We generate four chords, the tonic (I), subdominant (IV), dominant (V), and parallel minor (VI) in one of two modes: $major = I$ or $minor = VI$. Each chord will be a triad, except for the dominant, where we add the seventh by setting *chord* to 4. We could additionally alter the key, or use Jazz chords with added sevenths and ninths. Since the chords will be arranged randomly it is essential to encode harmonic constraints already in the chord pool, for example by not excessively mixing keys or modes.

$$
\begin{aligned}
&\text{CHORDS} : \langle \\
&\quad mode_{set}(choice(major, minor)) \\
&\quad chord_{set}(3) \\
&\quad [harmony_{set}(I), harmony_{set}(IV), harmony_{set}(V), harmony_{set}(VI)] \\
&\quad \langle harmony = V \longrightarrow chord_{set}(4) \rangle \\
&\rangle
\end{aligned}
$$

Example: Random Chord Progressions

PROGRESSION generates a chord progression for a sentence f using our chord pool. We first set the *chords* attribute with the result of the CHORDS grammar. Next, we apply f, which generates the temporal division for our progression. Finally, for each entity generated by f, we select a random chord from *chords* and append only the harmonic attributes K_H.

$$
\begin{aligned}
&\text{PROGRESSION} : (f \in V_\lambda^*) \mapsto \langle \\
&\quad chords_{set}(\text{CHORDS}) \\
&\quad f \\
&\quad append(\langle choice(chords), select(K_H) \rangle) \\
&\rangle
\end{aligned}
$$

Example: Chord Texture Query

CHORD is a helper function that we will use to query the *chords* attribute. Since *query* returns all overlapping entities, we use a function $first$ that returns only the first entity of a sentence. In order to guarantee that there is at least one entity, we use the identity function $\langle\rangle$ as a fallback. Since we want to preserve attributes on the input entity, we again only append the harmonic attributes K_H.

$$
\begin{aligned}
&\text{CHORD} : \mathit{append}(\langle \\
&\quad first([query(chords), \langle\rangle]) \\
&\quad \mathit{select}(K_H) \\
&\rangle)
\end{aligned}
$$

4.3.4 Pieces

We now have all the necessary components for generating a complete, polyphonic piece. Our piece will consist of multiple voices in different roles, which we define in the VOICE grammar. The harmonic information on which we build these voices comes from a singular chord progression layer that we generate with the PROGRESSION grammar, similar to the example in Section 4.1.4. The GLOBALS grammar will randomize global parameters on the axiom, and the PIECE grammar will integrate all these components and apply them in the correct order.

Example: Voices

VOICES generates the roots for the voice layers of our polyphonic piece. We define a *role* for each voice, which will be used by MOTIF to select one of the motif grammars. The *play* attribute is the name for an instrument or instrument family. How this value is interpreted is up to playback system. For the pad, bass, and drums we use the label pool $(1, 2)$, which means each has at most two different motifs per piece. The label pool for the two leading voices is more interesting. When the first voice plays motif 1, the second voice is silent, as indicated by the label 0. When the second voice plays motif 2, the first voice is silent. Motif 3 will be played by both at the same time. This only works if we additionally set the *layer* attribute of the lead voices to the same value.

$$
\begin{aligned}
&\text{VOICES} : [\\
&\quad \langle \mathit{role}_{set}(\text{'bass'}), \mathit{play}_{set}(\text{'contrabass'}), \mathit{labels}_{set}(1, 2)\rangle \\
&\quad \langle \mathit{role}_{set}(\text{'pad'}), \mathit{play}_{set}(\text{'piano'}), \mathit{labels}_{set}(1, 2)\rangle \\
&\quad \langle \mathit{role}_{set}(\text{'lead'}), \mathit{play}_{set}(\text{'violin'}), \mathit{labels}_{set}(1, 0, 3), \mathit{layer}_{set}(1)\rangle \\
&\quad \langle \mathit{role}_{set}(\text{'lead'}), \mathit{play}_{set}(\text{'piccolo'}), \mathit{labels}_{set}(0, 2, 3), \mathit{layer}_{set}(1)\rangle \\
&\quad \langle \mathit{range}(4), \mathit{role}_{set}(\text{'drum'}), \mathit{play}_{set}(\text{'percussion'}), \mathit{labels}_{set}(1, 2)\rangle \\
&]
\end{aligned}
$$

Example: Randomizing Global Attributes

GLOBALS initializes the attributes that will be constant for the whole piece. We start by setting the global attributes and the *piece* attribute, which will be used by our LAYER and LABEL grammars. Our motif generators are flexible enough that they can handle almost any meter, as shown in Figure 4.7. There is no particular reason for setting exactly these attributes either. For example, one could use a different *tempo* for different parts of a piece, or we could set the *monotony* for each voice individually. Alternatively, we can integrate these attributes as parameters in a graphical interface to give users primitive control over the generated result, which we show in Figure 5.1. For any other canonical attributes that are missing here, we assume the default values that were established in this chapter.

$$\begin{aligned}
&\text{GLOBALS} : \langle \\
&\quad piece_{set}(rand) \\
&\quad key_{set}(\lfloor uniform(0, 12) \rfloor) \\
&\quad tempo_{set}(uniform(80, 140)) \\
&\quad beats_{set}(choice(2, 3, 4, 6)) \\
&\quad beatType_{set}(choice(4, 8)) \\
&\quad monotony_{set}(uniform(0.25, 0.75)) \\
&\quad diversity_{set}(uniform(0, 0.5)) \\
&\quad depth_{set}(choice(2, 3, 4)) \\
&\rangle
\end{aligned}$$

Example: Piece

Finally, we define the PIECE grammar, which will generate the complete piece. For the overall duration of the piece we set *span* to a number of measures equal to the number of leaves in the layer. For the *chords* attribute we use the PROGRESSION grammar and pass the LAYER as its argument. This means the chord progression will have the same temporal structure as the voices, which is useful since we will later query it with the CHORD function. The last rule applies our various sub-grammars in sequence. Note that the LABEL and CHORD sub-grammars generate only one entity, while VOICES, LAYER, and MOTIF may generate any number of entities.

$$\begin{array}{l} \text{PIECE} : \langle \\ \quad \text{GLOBALS} \\ \quad span_{set}(2^{depth} * measure) \\ \quad chords_{set}(\text{PROGRESSION}(\text{LAYER})) \\ \quad \langle \text{VOICES}, \text{LAYER}, \text{LABEL}, \text{CHORD}, \text{MOTIF} \rangle \\ \rangle \end{array}$$

Figure 4.12 shows one example of the PIECE grammar. We can see how the algorithm reuses the limited set of motifs and how the two lead voices interact. The piano grammar has more variation than the other voices, as the seventh chords have an additional note. The piece lacks overall structure. For example, there is no satisfying harmonic resolution at the end, but this would be a lot to ask from our random progression generator. Nevertheless, the results are quite satisfying. Most importantly, the piece features polyphony on the large scale as a set of semi-independent voices, which are all synchronized to a hidden chord layer, and on the small scale in the form of chords and random arpeggios. We were able to achieve this within the boundaries of our theoretic framework, without compromising musical quality or our intuitions about the composition process. In the upcoming Chapter 5 we will discuss how we implemented this theoretic framework, followed by additional musical results in Chapter 6.

4.4 Conclusion

In this chapter we presented domain-specific musical extensions for the generic framework presented in Chapter 3. These included a strategy for generating both sequential and parallel temporal structures, which we called *probabilistic temporal-split grammars*, various operators that can be used to generate and transform interval arrangements, and simple models for meter and diatonic scales. We further demonstrated their application with various isolated examples and finally within a larger grammar that produces complete, multi-instrument pieces. This concludes the detailed definition of our theoretical contributions. In the next chapter we will discuss the implementation of our framework as a TypeScript library and an accompanying browser-based playback environment and score visualization.

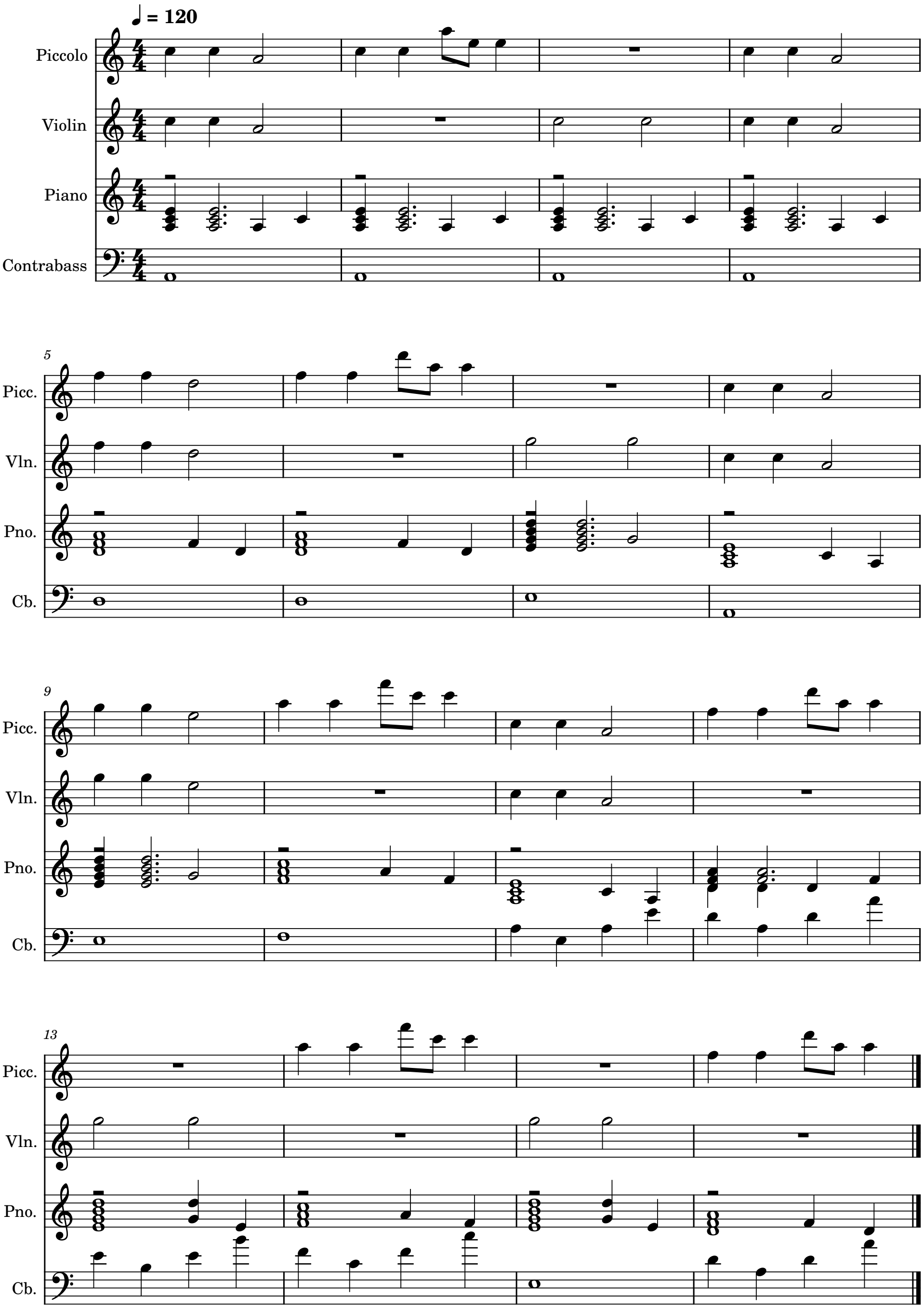


Figure 4.12: Manual transcription of one generation of the PIECE grammar. The drum voices are not included. The document attachments include two musical interpretations, one with percussion (File: piece1.mp3) and one without (File: piece1.score.mp3).

CHAPTER 5

Implementation

In this chapter we describe how we realized the theoretic ideas from the previous chapters as an executable program. We will discuss technologies that we used and show how certain parts of the theory are implemented. The decision for the final technology stack primarily depended on audio playback and visualization capabilities, available programming languages, and portability. In the end, audio and graphics requirements could be sufficiently met by a modern web browser. The framework and grammars are written in TypeScript, a strongly typed superset of JavaScript. Concerning portability we have to consider two types of clients: developers, who use the system to create musical grammars, and consumers, who want to listen to the generated music, or interact with the derivation process on a high level. While we can expect developers to install specific libraries and tools, the consumer will be reluctant to install additional software. By deploying the application as a website the consumers can access it with zero setup.

5.1 Language

We experimented with custom interpreters, but soon realized that designing a new language was infeasible. At this point robust behavior and mature tooling are more useful to us than an optimized syntax. JavaScript, the web's native programming language, can be criticized for various inconsistencies in its core design, lack of type-safety, and fragmented ecosystem. On the positive side it lends itself well to a functional programming style, it has terse syntax, and allows for a high-degree of meta programming. The latter is useful when we want to integrate our system into custom code editors and run code fragments directly in the application. As mentioned, we use TypeScript (version 4.2) for development, which alleviates much of the pain associated with untyped scripting languages. Its flexible type system is capable of statically verifying most of our components. Unfortunately, the formal TypeScript specification is locked at version 1.8 [Mic16] and the authors now rely on the TypeScript handbook [Mic21] as the main source of documentation. We will only describe the most vital parts of the code and only those aspects that were strongly influenced by the technology.

Before we introduce code examples, we provide a short summary of TypeScript's syntax. The first line below defines a constant named `f`. Its type is inferred from the value on the right side of the equal sign, which is an anonymous function. The signature of this function specifies two generic type parameters `A` and `B`, a parameter `a` of type `A`, and a parameter `b` of `B`. The return type `[B, A]` is a tuple, where the first element is of type `B` and the second of `A`. The implementation follows after the arrow and generates a tuple by reversing the order of the arguments. `f` essentially swaps its arguments. TypeScript further allows us to define types without implementation. The second line describes the type of variable `f` as the type `F`. Note that in the type definition the return type comes after the arrow rather than after a colon. The third line is similar to the second line, but binds the generic parameters directly to the type, which allows us to specify them explicitly. For example, we can write `F1<string, number>`, but not `F<string, number>`.

```
const f = <A, B> (a: A, b: B): [B, A] => [b, a]
type F = <A, B> (a: A, b: B) => [B, A]
type F1<A, B> = (a: A, b: B) => [B, A]
```

5.1.1 Expression Implementation

Expressions are an important part of our domain-specific language, as explained in Section 3.4. The idea is that whenever a non-function value is expected, we can either specify a constant, or a function over some input set, usually our entities. The names in the code do not necessarily correspond to the names used in the theory. For example, instead of an entity $v \in V$ we use a variable `x` with a type `X`, as our generic definition of an expression is practically not restricted by our definition of an entity. Restrictions on the expression input, for example the presence of certain attributes, can be specified with type constraints where necessary. We define a type shorthand `Fun<Y, X>` for a function from a set `X` to a set `Y`. The `Exp<Y, X>` type represents an expression as either a constant `Y`, or a function from `X` to the same type of expression. In this specific case we cannot use our function shorthand `Fun<Exp<Y, X>, X>` as it breaks TypeScript's recursion rules. In Section 3.4 we defined the λ function for converting a higher-order expressions to a plain function, which is called `exp` in our implementation. The `val` function is a helper function for evaluating an expression in a single call.

```
type Fun<Y, X> = (x: X) => Y
type Exp<Y, X> = Y | ((x: X) => Exp<Y, X>)
const exp = <Y, X> (f: Exp<Y, X>): Fun<Y, X> => /*impl*/
const val = <Y, X> (f: Exp<Y, X>, x: X): Y => exp(f)(x)
```

5.1.2 Attribute Implementation

An object in JavaScript is a bag of key-value pairs, which we can dynamically modify with a variety of syntactic constructs. The language essentially has attributes built into its core design. Besides dynamic access and enumeration, it has the *spread* syntax, which can be used to integrate an existing object into a new object, and *deconstruction* syntax, which takes a subset of an object's properties [Ecm20]. Consequently, we do not need an implementation for *select* and *append*, which were defined in Section 3.4.3. Further, JavaScript's prototypal inheritance model allows us to define a prototype for an object, which means that any access on a property that is not available directly on the object itself will be delegated upwards in the prototype chain. This is equivalent to native attribute inheritance, as described in Section 3.3. Although we did not do any profiling, using prototypal inheritance should be very efficient, since the parent state is reused, and a JavaScript interpreter is likely optimized for property lookup. A setter with prototypal inheritance can be implemented like this:

```
const setp = <O, X> (o: Exp<O, X>): Fun<X & O, X> => x => (
  Object.assign(Object.create(x), val(o, x))
)
```

Unfortunately, properties on prototypes are hidden from certain meta-programming features. We instead prefer using plain objects and the spread syntax for inheritance, as every native construct that works on prototypal properties also works on plain properties, but the converse is not true. For example, the spread syntax only considers properties defined directly on the object, and native serialization to JSON also ignores prototypes. The following code defines a setter using the spread syntax.

```
const set = <O, X> (o: Exp<O, X>): Fun<X & O, X> => x => (
  {...x, ...val(o, x)}
)
```

The lines below illustrate how the setter can be used in practice. The object literal syntax allows us to set multiple attributes at once, which is a major difference from the setter we used in the theoretic examples. In this concrete example, there is a dependency between `halfAnswer` and `answer`, so we cannot set both in the same call. The third line uses an alternative syntax, but is otherwise equivalent to the second line.

```
set({halfAnswer: 21})
set(x => ({answer: 2 * x.halfAnswer}))
set('answer', x => 2 * x.halfAnswer)
```

5.1.3 Grammar Definition

The generic core types and library functions are independent from a particular entity type, so that a grammar author can define a custom entity type and benefit from static type-checking. The framework provides the `Entity` type for convenience, which defines the various canonical attributes of our musical models, including the basic `time`, `span`, and `freq` attributes, metric and diatonic attributes, and additional attributes necessary for playback and visualization. The `EntityDefaults` object defines the default values for an `Entity`. Custom attributes can be added to the `Entity` type with a type union and to the `entity` object with the spread syntax. In the example below we add the custom `chord` attribute from Section 4.3.3 with a default value of `3`.

```
type ChordEntity = Entity & {chord: number}
const entity: ChordEntity = {...EntityDefaults, chord: 3}
```

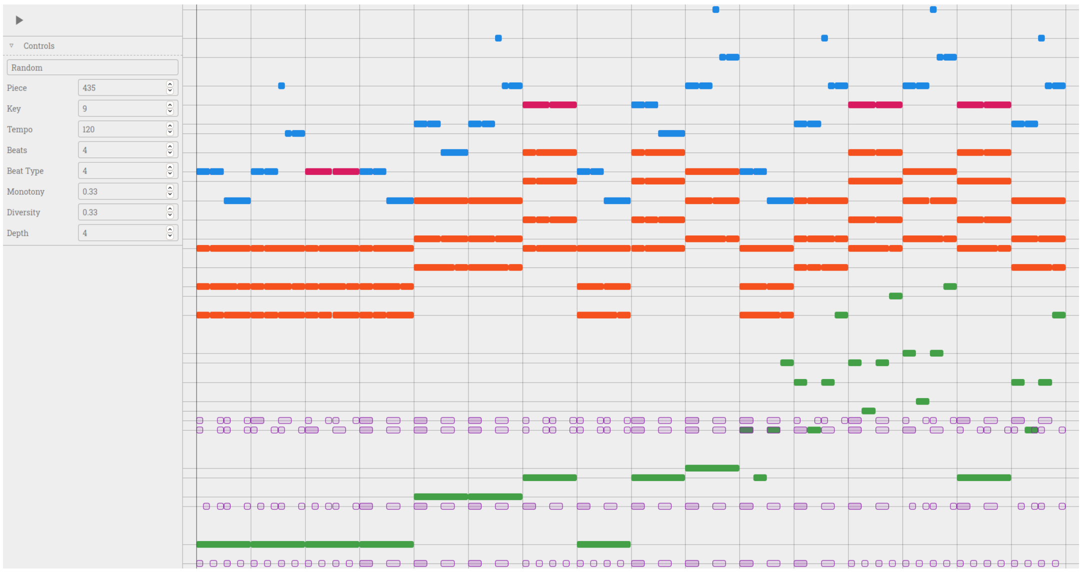
The `chord` function below is a reusable sub-grammar that takes a number expression as an argument and generates n chord notes. Ideally, the entity type for a sub-grammar can be specified as a constrained generic type argument. In the concrete example, constraining `X` to extend `Entity` is actually much stricter than necessary, as the sub-grammar only relies on the `degree` and `freq` attributes. Over time, a grammar author can build a whole library of modular functional components for their desired musical style. The equivalent theoretic notation is shown below the code.

```
const chord = <X extends Entity> (n: Exp<number, X>): Fun<X[], X> => pipe(
  range(n, i => set('degree', 2 * i)),
  set('freq': diatonic)
)
```

$$
\begin{aligned}
&\mathrm{CHORD} : (n \in \mathbb{N}_\lambda) \mapsto \langle \\
&\quad range(n, i \mapsto degree_{set}(2i)) \\
&\quad freq_{set}(diatonic) \\
&\rangle
\end{aligned}
$$

These final lines show how we can evaluate `chord` using `entity` as input. Despite using the custom entity type `ChordEntity`, the complete example is statically verifiable by TypeScript's type-checker.

```
const notes = chord<ChordEntity>(x => x.chord)
const score: ChordEntity[] = val(notes, entity)
```



Piece 435
Key 9
Tempo 120
Beats 4
Beat Type 4
Monotony 0.33
Diversity 0.33
Depth 4

Figure 5.1: This screenshot shows the interface for the example from Section 4.3 (File: piece1.mp3). The colors in the visualization indicate the instrument. Some parts of the magenta voice are hidden by the blue voice. The controls on the left side correspond to the attributes set by the GLOBALS sub-grammar.

5.2 Interface

In addition to the scripting interface, users can interact with the grammar through a graphical interface and score visualization, pictured in Figure 5.1. The visualization maps time to the horizontal and logarithmic frequency to the vertical axis. Notes are displayed as colored bars based on their `time`, `span`, and `freq` attributes. The color of a note depends on the nominal `color` attribute, which selects one of sixteen predefined colors and allows visual separation between notes. Horizontal and vertical grid lines mark specific values on the X and Y axes. Labels are only visible when the user places their cursor on or near them. Their values are measured in seconds for the X axis and MIDI numbers for the Y axis. The button in the top-left corner can start and pause playback. While the sound fonts are loading in the background, this button is disabled. Once all resources are available, a vertical playback progress indicator is displayed at the start of the score. This indicator can be moved to another point on the timeline by clicking on the corresponding vertical grid line. At the left side of the interface are the grammar controls, which can be used to modify attributes of the initial entity. They must be manually defined by the user.

For authoring new grammars, it is important that we can quickly review changes to the code and manipulate parameters. We initially attempted to integrate a code editor into the interface, and for some time even relied on the browser's built-in REPL and multi-line code editor. Yet, most of the web-based options we tested were far less sophisticated than desktop-based editors, especially in regards to code completion, navigation, and type-checking. In the end we adopted a hybrid approach, where the grammar author works in their preferred editor to develop the grammar and interface, and a file-watcher automatically rebuilds and reloads the code after a change. The latency is usually negligible, and one can use the interface controls for fine-tuning parameters. The benefit is that we have complete access to version control, package managers, type-checking, and other productivity tools that enable us to develop mature, well-organized modules.

5.3 Audio

Our requirements for audio processing are rather primitive. We only need to be able to load a large number of short audio files and schedule them for playback. With the Web Audio API we can setup complex graphs of audio data sources, filters, and sinks, that can be manipulated with low latency parameter handles. We barely scratched the surface of this rich API, as we can load, decode, and schedule an audio file with just a few lines of code. Symbolic audio formats such as MIDI usually rely on *sound fonts*, which are essentially mappings from frequencies to audio samples. The generation of a sound font is not trivial, as one needs to record or synthesize a large number of instrument sounds. Fortunately, several collections of free sound fonts have been published under permissive licenses. As the file sizes can become quite large, the player loads sound fonts lazily over the network. We use the `play` attribute with an URL that points to the font.

Once a score is generated and the sounds are loaded, the user can start the audio player. Initially, we implemented an immediate playback strategy with JavaScript's native timing mechanism. Playback would start at the earliest set of notes, and we would use `setTimeout` to wait until the start of the next set of notes. Yet, the temporal precision employed by `setTimeout` and `setInterval` turned out to be insufficient. Further, due to notes being scheduled immediately, this approach caused delays when the processor could not keep up. For that very reason, the Web Audio API provides precise timing via `currentTime` and allows scheduling notes in advance. Since scheduling the whole score is wasteful, we implemented a hybrid approach. We use an imprecise scheduling loop with `setInterval`, which accurately schedules all notes within the current time window. The time windows overlap, so that notes get scheduled early enough, even if the imprecise loop falls behind. A peeking iterator takes the events from a chronological queue, so that no note can be scheduled twice. Further, we connect each audio node to a master node in the audio graph, which we can disconnect to stop playback of all active and scheduled notes.

CHAPTER 6

Evaluation

In this chapter we will once more compare our approach to the state of the art. This comparison is not trivial, as the ultimate goal of this and similar works is the provision of *potential*, which is difficult to measure meaningfully. This potential appears on two levels, once on the level of the individual grammar and its possible results, and once again on the level of the framework and the different types of grammars we can define. The underlying model that describes both aspects is the same, as the grammar definitions themselves can be considered sentences in a meta-grammar. While we could study the distribution of pieces generated by the example in Section 4.3 and count the number of combinations of its non-deterministic choices, it will say little about the capabilities of the meta-grammar, which is more interesting to us. Yet, a similar analysis of the meta-grammar is much more difficult, as its entities have complicated relationships that need to be considered when we count not only the number of possible combinations of the grammar features, but those that are semantically meaningful. Even then, the comparison of such statistics across multiple works and their respective meta-grammars is not guaranteed to be enlightening.

Instead, we will attempt a qualitative comparison, with a focus on the polyphonic aspect. As explained in Section 2.1, machine learning and related methods are orthogonal to generative grammars and we will not consider them. For formal grammar approaches we will consider interesting features of the meta-grammar, but not the particular instances. For example, Steedman [Ste96], Rohrmeier [Roh07], and Bel and Kippen [BK92b] focus on the empirical replication of certain styles of music. As such their work is of course very valuable to us since we can attempt to integrate their grammars as sub-grammars into our own framework. For example, we could swap our naive chord progression grammar with Steedman's context-free grammar for jazz chord sequences. But from the meta perspective, their languages are optimized with very specific constraints towards a different goal.

A more meaningful comparison can be made with the works by Quick and Hudak on probabilistic temporal graph grammars (PTGG) [QH13a] and the subsequent extensions by Melkonian [Mel19], as they intend to define a theoretic framework with general applicability and use a parametric model of time. We speculate that our model of time subsumes that of PTGGs, as it is based entirely on split operations with relative units and note durations, both of which are provided in our framework. Melkonian extends PTGGs with matching on note durations, which we allow by representing the LHS as a general predicate. Subtree synchronization can be emulated by reusing a derivation result, or with a shared seed, although the implementation in Haskell with the let-in construct seems like a very elegant solution. Finally, in regards to polyphony, PTGGs do not seem to provide a native mechanism. Quick and Hudak achieve polyphony through a processing step that happens on the terminals of the grammar. Yet, the scores presented in these works do not suggest that time intervals may freely overlap even after post-processing.

The work by Tanaka and Furukawa [TF12] is very relevant, as they integrate polyphony on the grammar level. Interestingly, they do not use a recursive or parametric division of the timeline. Instead, the total duration of the notes on the RHS is greater than the duration of the replaced entity. Subsequent notes are pushed back in the song, which consequently grows with every replacement. We can emulate this insertion of entities with a split operation on the whole voice after each replacement, but it is arguably counter-intuitive within our temporal model. Tanaka and Furukawa perform this type of replacement on multiple parallel voices and only allow replacement if certain constraints are fulfilled, for example that all voices grow by the same amount. The rules themselves are generated with machine learning. We speculate that it might be challenging to design rules by hand in such a system, as it seems difficult to predict their interaction with the constraints across multiple voices. Their polyphonic model is further limited to a fixed set of voices, while our approach allows the definition of voices on every level of the derivation tree.

The generative grammar definition language (GGDL) by Holtzman [Hol81] provides a variety of interesting meta-level features. While the presented scores appear to be polyphonic, it is not explained in detail how it is achieved. It is possible that a polyphonic mapping is part of the final stage of their derivation pipeline. In that case, it can be emulated in our system with a pipe over the initial grammar and the polyphonic mapping rule. McCormack [McC96] presents a similar construction of various grammar features such as nested derivation as we did in Chapter 3. Polyphony can be achieved with both parallel and sequential placement of notes, but only with discrete symbols. This is very easy to replicate with our temporal model. They do allow context-sensitive matching, which is not available in our system. Neither work uses an explicit parametric treatment of the time dimension.

Our own model of time appears to be capable of replicating the temporal structures used in prior works. Its recursive parallelism allows us to define pieces with an unbounded number of entities within the same temporal slice, which we have not seen before in similar systems. We believe this is a very important feature of a polyphonic music generation system since many types of music require polyphony on at least two levels: voices and chords. Nested grammars allow us to integrate context-sensitive operations directly into the rules of the grammar and allow for subsequent rule application on their results, which is not possible if the same operation is applied in a post-processing step. Finally, we think that using generated sentences as a common context, as we did with the harmonic progression, is an effective and intuitive way of synchronizing parallel branches of the derivation tree that integrates elegantly with the other features of our meta-language.

When we look at computer-generated music from the outside, it can be difficult to judge how valuable it really is. How many pieces would one need to hear to be convinced that an automatic music generator does not just cycle through a set of manually composed pieces? If we want someone to appreciate our music in terms of the creative work that went into it, they need to understand the instrument, and how easy that is depends on the complexity of said instrument. In fact, it is often the use of simpler tools that is more impressive. For example, it is easy to see oneself playing a beat using pots and utensils, which means we can better appreciate when someone does it skillfully. The same beat played on an expensive drum-set may not find as much recognition because the implicit expectations for both the instrument and the player are much higher.

In formal grammars the notion of complexity is rigorously defined in terms of the Chomsky hierarchy. It was an impressive accomplishment when Mark Steedman reformulated the context-sensitive jazz chord grammar [Ste84] as a context-free grammar [Ste96] with comparable results, as the former is embedded in a framework on a fundamentally different complexity level. Inversely, it is very much expected that we can generate music with a Turing complete language, and thus the additional complexity must be justified in the sophistication of the results. Again, it is a question of potential and how much of it we actually utilize. The claim that our theoretic framework is context-free is true to an extent, but the unrestricted use of attributes and nested grammars with context-sensitive operations strongly indicates that this is not the whole truth. Further, embedding the system in a general-purpose language allows truly unrestricted manipulation. We believe we managed to strike a good balance between restrictions and freedoms in our system, without too many sacrifices in terms of simplicity and musical intuitions. Nevertheless, it is important that we consider used potential when evaluating such systems, especially when we compare them to the work of others who intentionally chose more restrictive models.

6.1 Results

At last, we provide some additional results for the grammar example from Section 4.3 in the top section of Table 6.1. The other two table sections list results of two alternative grammars with a similar setup. The difference between them is primarily the selection of the instruments, where the ‘piano’ grammar uses a single piano sound font, and the ‘synth’ grammar uses a mixture of synthesized sounds. They both feature a two-tier layout, where we first divide the piece into sections, and only then divide each section

file	key	chords	bpm	meter
piece1.mp3	A	minor	120	4/4
piece2.mp3	A	major	136	6/8
piece3.mp3	E	major	114	4/4
piece4.mp3	E	minor	93	4/4
piece5.mp3	A#	minor	108	2/8
piece6.mp3	G#	major	87	3/8
piece7.mp3	A	major	126	6/8
piano1.mp3	-	jazz	144	4/4
piano2.mp3	C	major	143	4/4
piano3.mp3	C	minor	140	3/4
synth1.mp3	-	jazz	138	2/4
synth2.mp3	-	jazz	144	4/4
synth3.mp3	-	jazz	120	4/4

Table 6.1: As selection of pieces generated with our approach. The audio files are attached to the digital version of this thesis and also available online, where we publish additional variations and results:
`https://github.com/eibens/thesis-2021`

into measures. Sparsity is used on the section level as well as the measure level, which means voices can be silent over large stretches of the song. We further use a ‘jazz’ chord pool with seventh and ninth chords, in addition to the major and minor chord pools. The chord pool can vary from one section to the next. We have now almost reached the end of this thesis. In the final chapter we will summarize our contribution and talk about potential future directions.

CHAPTER 7

Conclusion

In this thesis we developed a formal grammar approach for generating polyphonic music called *probabilistic temporal-split grammars*. Each musical entity is associated with an independent temporal scope, and replacement entities are implicitly layered due to attribute inheritance. This allows us to model parallel structures—from chords to complex voice patterns—at all levels of the derivation tree. Sequential placement of entities is facilitated by a split operator, which supports absolute, relative, and flexible part sizes, and a repeat operator, which can be used to tile intervals of arbitrary duration. Parallel voices can be supplied with common context by retaining terminal strings of sub-grammars and extracting their entities with a time-based query mechanism. We have further defined models of meter and diatonic pitch, reminiscent of classic musical notation, for calculating absolute time and frequency values from relative parameters.

The domain-specific models are embedded in a generic functional framework for defining non-deterministic, context-free grammars. Its successful application provides evidence that the functional paradigm is well suited to model this type of system and, by employing a component-based architecture, remains transparent even for complex grammars. Its implementation as a library within a general-purpose language makes it easily extensible. Further, using a generic attribute-based representation for entities instead of a fixed parametric representation allows the grammar author to work with custom musical structures. Of course, the additional generality does not come without cost, as the burden of developing higher-level musical abstractions is shifted to the user.

7.1 Limitations and Future Work

We believe our approach is intuitive and accessible in theory, but the implementation as a TypeScript library may pose serious usability problems. Scripting languages are commonly considered to be easier to learn then conventional programming languages, and JavaScript specifically is extremely popular, but their use still requires significant programming knowledge. As such, our target audience must be familiar with both programming and music theory. The former could be helped with a domain-specific or visual language, for example a node-based interface. Alternatively, we could provide a visualization of the derivation tree and integrate user input directly into the derivation process. The derivation of a terminal entity could be suspended until the user chooses the next operation, essentially allowing them to construct pieces and rules in a semi-automatic fashion. This could also be used as an alternative to random number generators, as the user could assume control over non-deterministic decisions.

While this work showed that a scope-based approach to music composition can be fruitful, other models for the time dimensions should be explored. For example, a lattice-like structure, similar to the lines of a written score, could be used to place notes within the scope of an entity. This would subsume our metric units and also generalize well to the pitch dimension and non-diatonic scales. The unification of meter and pitch could go even deeper, given that frequency is itself a rhythmic phenomenon. The former describes the interplay of periodic signals below the audible threshold of 20 Hz, the latter signals above. The linear transformation of the time dimension, defined by the entity's offset and duration, can alternatively be thought of as a wave, where the offset is the phase and the duration is the wavelength. A system that uses a wavelike representation in frequency space as the underlying model could be capable of representing both large-scale score generation and small-scale sound synthesis.

Allowing the reuse of sub-grammar results provides a basis for context-sensitive operations, as we have demonstrated with the split operator and temporal queries. The library could be improved with additional score analysis tools. For example, we could create a melody in a generative pass, extract the emerging harmonic information in an analysis pass, and use this to generate accompaniments in a second generative pass. The end goal would be a bidirectional system of score generation and score parsing, where the system state goes through multiple subsequent translations between abstract and concrete representations. With an improved interface, composers could use such a system to reverse-engineer existing scores and dynamically remix their components. This could be extended to raw audio, where Fourier analysis can reveal metric and harmonic structure. The analyzed audio data could be sliced into samples, which could be used to build new sound fonts on the fly. In the end, there remains a lot to be explored—at least until someone finds a general method for turning abstract ideas into concrete reality. For music, this does not seem like a distant possibility.

APPENDIX A

Source Code

This chapter contains the source code for the music generator described in Section 4.3. Note that only the grammar and GUI definitions are included, not the complete library. Implementation details can be found in Chapter 5.

A.1 Grammar Definition

This section lists the theoretic code from Section 4.3. Listing it here in compact form allows for convenient comparison with the corresponding TypeScript source code in Section A.2.

```
$\text{CUT} : (s \in \mathbb{R}_\lambda) \mapsto \mathit{trim}(\mathit{choice}(\mathit{split}([\mathit{size}_{set}(s), \langle\rangle]), \mathit{split}([\langle\rangle, \mathit{size}_{set}(s)])))$
$\text{REP} : (s \in \mathbb{R}_\lambda) \mapsto \mathit{trim}(\mathit{repeat}_{\ulcorner}(\mathit{size}_{set}(s), \mathit{index}_{set}))$
$\text{DEG1} : \mathit{degree}_{set}(\mathit{choice}(I, III, V, VIII))$
$\text{DEG2} : \mathit{degree}_{set}(\mathit{choice}(II, IV, VI, VII))$
$\text{PR} : (p \in [0,1]_\lambda, f : V_\lambda^*) \mapsto \langle \mathit{rand} \leq p \longrightarrow f \rangle$

$\text{BASS} : \langle$
    $\text{REP}(\mathit{choice}(\mathit{measure}, \mathit{whole}, \mathit{half}, \mathit{quarter}))$
    $\mathit{index} > 0 \longrightarrow \text{DEG1}$
    $\mathit{octave}_{set}(\mathit{choice}(1, 2))$
    $\mathit{freq}_{set}(\mathit{diatonic})$
$\rangle$

$\text{PAD} : \langle$
    $\mathit{octave}_{set}(\mathit{choice}(3, 4))$
    $\text{PR}(0.5, \text{CUT}(\mathit{choice}(\mathit{half}, \mathit{quarter}, \mathit{eighth})))$
    $\mathit{range}(\mathit{chord}_{or}(3), i \mapsto \mathit{degree}_{set}(2i))$
    $\text{PR}(0.5, \text{CUT}(\mathit{choice}(\mathit{half}, \mathit{quarter}, \mathit{eighth})))$
    $freq_{set}(diatonic)$
$\rangle$

LEAD : $\langle$
    $\mathrm{PR}(0.5, \mathrm{CUT}(choice(half, quarter, eighth))$
    $\mathrm{PR}(0.5, \mathrm{CUT}(choice(half, quarter, eighth))$
    DEG1
    $\mathrm{PR}(0.5, \mathrm{REP}(choice(half, quarter, eighth))$
    $\mathrm{PR}(0.5, \mathrm{DEG1})$
    $\mathrm{PR}(0.25, \mathrm{DEG2})$
    $freq_{set}(diatonic)$
$\rangle$

DRUM : $\langle$
    $\mathrm{REP}(choice(measure, half, quarter, eighth))$
    $gain_{set}(gain * span/whole)$
    $\langle odd(index) \longrightarrow gain_{set}(gain/2)\rangle$
    $\langle span \leq quarter \longrightarrow time_{set}(time + choice(0, span/2))\rangle$
$\rangle$

MOTIF : [
    $role =$ 'lead' $\longrightarrow$ LEAD
    $role =$ 'pad' $\longrightarrow$ PAD
    $role =$ 'bass' $\longrightarrow$ BASS
    $role =$ 'drum' $\longrightarrow$ DRUM
]

SYNC : $(p \in [0, 1]_\lambda, f \in V_\lambda^*) \mapsto \langle$
    $sync_{set}(rand)$
    $f$
    $\langle p < sync \longrightarrow sync_{set}(rand)\rangle$
    $seed(sync)$
$\rangle$

LAYOUT : $(f \in V_\lambda^*) \mapsto derive($
    $depth_{!} > 0 \longrightarrow \langle$
        $depth^{!}_{set}(depth - 1)$
        $\mathrm{SYNC}(monotony_{or}(0), split(range(2)))$
        $f$
    $\rangle$
$)$

LAYER : $\langle$
    $layer_{or}(rand)$
    $seed(piece)$
    $\mathrm{LAYOUT}(\mathrm{PR}(diversity, seed(rand + layer)))$
$\rangle$
```

```
LABEL : $\langle$
    $label_{set}(choice(labels))$
    $\langle label = 0 \longrightarrow [] \rangle$
    $seed(piece + label)$
$\rangle$

CHORDS : $\langle$
    $mode_{set}(choice(major, minor))$
    $chord_{set}(3)$
    $[harmony_{set}(I), harmony_{set}(IV), harmony_{set}(V), harmony_{set}(VI)]$
    $\langle harmony = V \longrightarrow chord_{set}(4) \rangle$
$\rangle$

PROGRESSION : $(f \in V_{\lambda}^{*}) \mapsto \langle$
    $chords_{set}$(CHORDS)
    $f$
    $append(\langle choice(chords), select(K_H) \rangle)$
$\rangle$

CHORD : $append(\langle$
    $first([query(chords), \langle \rangle])$
    $select(K_H)$
$\rangle)$

VOICES : [
    $\langle role_{set}(\text{'bass'}), play_{set}(\text{'contrabass'}), labels_{set}(1, 2) \rangle$
    $\langle role_{set}(\text{'pad'}), play_{set}(\text{'piano'}), labels_{set}(1, 2) \rangle$
    $\langle role_{set}(\text{'lead'}), play_{set}(\text{'violin'}), labels_{set}(1, 0, 3), layer_{set}(1) \rangle$
    $\langle role_{set}(\text{'lead'}), play_{set}(\text{'piccolo'}), labels_{set}(0, 2, 3), layer_{set}(1) \rangle$
    $\langle range(4), role_{set}(\text{'drum'}), play_{set}(\text{'percussion'}), labels_{set}(1, 2) \rangle$
]

GLOBALS : $\langle$
    $piece_{set}(rand)$
    $key_{set}(\lfloor uniform(0, 12) \rfloor)$
    $tempo_{set}(uniform(80, 140))$
    $beats_{set}(choice(2, 3, 4, 6))$
    $beatType_{set}(choice(4, 8))$
    $monotony_{set}(uniform(0.25, 0.75))$
    $diversity_{set}(uniform(0, 0.5))$
    $depth_{set}(choice(2, 3, 4))$
$\rangle$

PIECE : $\langle$
    GLOBALS
    $span_{set}(2^{depth} * measure)$
```

$chords_{set}$(PROGRESSION(LAYER))
⟨VOICES, LAYER, LABEL, CHORD, MOTIF⟩
⟩

A.2 Grammar Definition (TypeScript)

This section lists the TypeScript source code that corresponds to the grammar in Section 4.3 and generated the results listed in the top part of Table 6.1. The import statements at the beginning of the file load the library functions provided by our framework. The `'./drums'` module defines constants for a non-standard percussion instrument and is not part of the library. The typed declaration `MyEntity` extends the `Entity` type with custom attributes, as explained in Section 5.1.3. The rest of the file closely resembles the code in Section A.1.

```typescript
import {diatonic, eighth, Entity, gleitz, half, measure, measures, overlaps,
        quarter, rel, repeat, split, translate, trim, whole} from '@dipl/lib-music'
import {clone, Exp, filter, first, fork, Fun, indexed, map, noop, pipe, pipeDeep,
        range, Seq, set, setx, use, val, when} from '@dipl/lib-core'
import {choice, pr, rand, seed, uniform} from '@dipl/lib-rng'
import {frac, lte, pow, round} from '@dipl/lib-math'
import {acousticGrandPiano, contrabass, I, II, III, IV, major, minor, musyngKite,
        piccolo, V, VI, VII, VIII, violin} from '@dipl/lib-constants'
import * as drums from './drums'

export type MyEntity = Entity & {
  piece: number
  depth: number
  monotony: number
  diversity: number
  layer: number
  label: number
  labels: number[]
  chord: number
  role: 'bass' | 'lead' | 'pad' | 'drum'
  chords: MyEntity[]
}

function Cut<X extends Entity> (size: Exp<number, X>): Fun<X[], X> {
  return trim(choice(
    split<X>(setx({size}), noop),
    split<X>(noop, setx({size}))
  ))
}

function Rep<X extends Entity> (size: Exp<number, X>): Fun<X[], X> {
  return trim(indexed(repeat<X>({
    size,
    cover: true
  }), i => set({index: i})))
}

function Deg1<X extends Entity> (): Fun<X, X> {
  return setx({degree: choice(I, III, V, VIII)})
}

function Deg2<X extends Entity> (): Fun<X, X> {
  return setx({degree: choice(II, IV, VI, VII)})
}

function Pr<X extends Entity> (p: Exp<number, X>, f: Exp<Seq<X>, X>): Fun<X[], X> {
  return when(pr(p), f)
}

function Bass<X extends Entity> (): Fun<X[], X> {
  return pipe(
    Rep(choice<number, X>(half, quarter)),
    when(x => x.index > 0, Deg1()),
    setx({octave: choice(1, 2)}),
    setx({freq: diatonic})
  )
}

function Pad<X extends MyEntity> (): Fun<X[], X> {
  return pipe(
    setx({octave: choice(3, 4)}),
    Pr(0.5, Cut(choice<number, X>(half, quarter, eighth))),
    range(x => x.chord || 3, i => set({degree: i * 2})),
    Pr(0.5, Cut(choice<number, X>(half, quarter, eighth))),
    setx({freq: diatonic})
  )
}

function Lead<X extends MyEntity> (): Fun<X[], X> {
  return pipe(
    Pr(0.5, Cut(choice<number, X>(half, quarter, eighth))),
    Pr(0.5, Cut(choice<number, X>(half, quarter, eighth))),
    Deg1(),
    Pr(0.5, Rep(choice<number, X>(half, quarter, eighth))),
    Pr(0.5, Deg1()),
    Pr(0.25, Deg2()),
    setx({freq: diatonic})
  )
}

function Drum<X extends Entity> (): Fun<X[], X> {
  return pipe<X>(
    Rep(choice<number, X>(measure, half, quarter, eighth)),
    setx({gain: x => x.gain * x.span / val(whole, x)}),
    x => when(x.index % 2 === 1, setx({gain: x.gain / 2})),
    x => when(lte(x.span, quarter), translate<X>(choice(0, rel(1 / 2))))
  )
}

function Motif<X extends MyEntity> (): Fun<X[], X> {
  return trim((x: X) => ({
    lead: Lead<X>(),
    pad: Pad<X>(),
    bass: Bass<X>(),
    drum: Drum<X>()
  }[x.role]))
}

function Sync<X extends Entity> (p: Exp<number, X>, f: Exp<X[], X>): Fun<X[], X> {
  return use(rand, sync => pipe<X>(
    f,
    use(p, p => seed<X>(p < sync ? rand : sync))
  ))
}

function Layout<X extends MyEntity> (f: Exp<X[], X>): Fun<X[], X> {
  return pipeDeep(
    x => x.depth,
    Sync(x => x.monotony, split(clone(2))),
    f
  )
}
```

```
function Layer<X extends MyEntity> (): Fun<X[], X> {
  return pipe(
    seed(x => x.piece),
    Layout(when(pr(x => x.diversity), seed<X>(rand, x => x.layer)))
  )
}

function Label<X extends MyEntity> (): Fun<X[], X> {
  return pipe(
    x => setx({label: choice(...x.labels)}),
    x => when<X>(x.label === 0, []),
    x => seed(x.piece, x.label, x.layer)
  )
}

function Chords<X extends Entity> (): Fun<X[], X> {
  return pipe(
    setx({mode: choice(major, minor)}),
    set({chord: 3}),
    map([I, IV, V, VI], harmony => set({harmony})),
    when(x => x.harmony === V, set({chord: 4}))
  )
}

function Progression<X extends MyEntity> (f: Exp<X[], X>): Fun<X[], X> {
  return pipe<X>(
    setx({chords: Chords}),
    f,
    x => ChordAppend(choice(...x.chords))
  )
}

function ChordAppend<X extends Entity> (c: Exp<X, X>): Fun<X, X> {
  return use(c, ({harmony, key, mode, chord}) => set(
    {harmony, key, mode, chord}
  ))
}

function Chord<X extends MyEntity> (): Fun<X, X> {
  return ChordAppend<X>(
    first(filter<MyEntity, X>(x => x.chords, overlaps), noop())
  )
}
```

```
function Voices<X extends MyEntity> (): Fun<X[], X> {
  const Play = (n: number) => gleitz(musyngKite, n)
  return fork(
    setx({
      role: 'pad',
      play: Play(acousticGrandPiano),
      labels: [1, 2],
      color: 2,
      layer: rand
    }),
    setx({
      role: 'lead',
      play: Play(violin),
      labels: [1, 0, 3],
      color: 0,
      layer: 1
    }),
    setx({
      role: 'lead',
      play: Play(piccolo),
      labels: [0, 2, 3],
      color: 1,
      layer: 1
    }),
    setx({
      role: 'bass',
      play: Play(contrabass),
      labels: [1, 2],
      color: 3,
      layer: rand
    }),
    pipe(
      clone(4),
      seed<X>(rand),
      setx({
        role: 'drum',
        play: drums.font,
        freq: choice(...drums.hats, ...drums.snares, ...drums.snares),
        labels: [1, 2],
        gain: 1 / 2,
        color: 4,
        layer: rand
      })
    )
  )
}

export function Globals<X extends MyEntity> (): Fun<X, X> {
  return setx({
    piece: round(uniform(0, 10000)),
    key: round(uniform(0, 12)),
    tempo: round(uniform(80, 140)),
    beats: choice(2, 3, 4, 6),
    beatType: choice(4, 8),
    monotony: frac(round(uniform(25, 75)), 100),
    diversity: frac(round(uniform(0, 50)), 100),
    depth: choice(2, 3, 4)
  })
}

export function Init<X extends MyEntity> (f: Fun<X[], X>): Fun<X[], X> {
  return pipe(
    x => seed(x.piece),
    setx({span: measures(pow(2, x => x.depth))}),
    f
  )
}

export function Piece<X extends MyEntity> (): Fun<X[], X> {
  return Init(pipe<X>(
    setx({chords: Progression(Layer)}),
    Voices,
    Layer,
    Label,
    Chord,
    Motif
  ))
}
```

A.3 GUI Definition (TypeScript)

This section lists the TypeScript source code that defines the graphical user interface shown in Figure 5.1. We assume that the grammar definition code from Section A.2 is located in the module `'./grammar'`. We define the starting entity, the visual controls, and the grid lines for the score visualization. The `vis` function generates the interface, and the `render` function attaches it to the browser window.

```
import {render} from 'preact'
import {fieldset, inputs, label, number, presets, propKey, vis} from '@dipl/dom-vis'
import {EntityDefaults, measure, repeat} from '@dipl/lib-music'
import {fork, Fun, map, set} from '@dipl/lib-core'
import {Globals, Init, MyEntity, Piece} from './grammar'

const entity: MyEntity = {
  ...EntityDefaults,
  piece: Math.round(Math.random() * 10000),
  depth: 4,
  monotony: 0.5,
  diversity: 0.5,
  layer: 0,
  label: 0,
  labels: [1],
  chord: 3,
  role: 'drum',
  chords: []
}

const input = inputs<MyEntity>(
  fieldset('Controls', inputs(
    presets({random: Globals}),
    propKey(label, 'piece', number(0, 10000, 1)),
    propKey(label, 'key', number(0, 12)),
    propKey(label, 'tempo', number(80, 140)),
    propKey(label, 'beats', number(2, 6)),
    propKey(label, 'beatType', number(2, 8)),
    propKey(label, 'monotony', number(0, 1, 0.01)),
    propKey(label, 'diversity', number(0, 1, 0.01)),
    propKey(label, 'depth', number(0, 5))
  ))
)

function Lines<X extends MyEntity> (notes: X[]): Fun<X[], X> {
  return Init(
    fork(
      repeat<X>({size: measure}),
      map(notes, x => set({freq: x.freq}))
    )
  )
}

render(
  vis({
    input,
    axiom: entity,
    notes: Piece,
    lines: Lines
  }),
  document.body
)
```

List of Figures

List of Tables

Bibliography

[Ace04] Andres Garay Acevedo. Fugue composition with counterpoint melody generation using genetic algorithms. In *International Symposium on Computer Music Modeling and Retrieval*, pages 96–106. Springer, 2004.

[AD86] Harold Abelson and Andrea A DiSessa. *Turtle geometry: The computer as a medium for exploring mathematics.* The MIT Press, 1986.

[Ame89] Charles Ames. The Markov process as a compositional model: A survey and tutorial. *Leonardo*, 22(2):175–187, 4 1989.

[AW05] Moray Allan and Christopher Williams. Harmonising chorales by probabilistic inference. In *Proceedings of the 17th Conference on Advances in Neural Information Processing Systems*, pages 25–32. The MIT Press, 2005.

[BAL96] John Biles, Peter Anderson, and Laura Loggi. Neural network fitness functions for a musical IGA, 1996.

[Bel89] Bernard Bel. Pattern grammars in formal representations of musical structures, 1989.

[Bel92] Bernard Bel. Modelling improvisatory and compositional processes. *Languages of Design, Formalisms for Word, Image and Sound*, 1(1):11–26, 1992.

[BHP17] Jean-Pierre Briot, Gaëtan Hadjeres, and François-David Pachet. Deep learning techniques for music generation: A survey, 2017.

[Bil94] John Biles. GenJam : A genetic algorithm for generating jazz solos. In *Proceedings of the 1994 International Computer Music Conference*, pages 131–137. ICMA, 1994.

[BK92a] Bernard Bel and Jim Kippen. Bol processor grammars, 1992.

[BK92b] Bernard Bel and Jim Kippen. Modelling music with grammars: formal language representation in the Bol Processor, 1992.

[BMP06] Steven Brown, Michael J Martinez, and Lawrence M Parsons. Music and language side by side in the brain: a PET study of the generation of melodies and sentences. *European Journal of Neuroscience*, 23(10):2791–2803, 2006.

[BMR$^+$20] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners, 2020.

[Bod02] Rens Bod. A unified model of structural organization in language and music. *Journal of Artificial intelligence research*, 17:289–308, 2002.

[BRBM78] William Buxton, William Reeves, Ronald Baecker, and Leslie Mezei. The use of hierarchy and instance in a data structure for computer music. *Computer Music Journal*, pages 10–20, 1978.

[CAVR98] Pedro P Cruz-Alcázar and Enrique Vidal-Ruiz. Learning regular grammars to model musical style: Comparing different coding schemes. In *International Colloquium on Grammatical Inference*, pages 211–222. Springer, 1998.

[CC07] Ching Hua Chuan and Elaine Chew. A hybrid system for automatic generation of style-specific accompaniment. In *4th International Joint Workshop on Computational Creativity*, 2007.

[Che04] Marc Chemillier. Toward a formal study of jazz chord sequences generated by Steedman's grammar. *Soft Computing*, 8(9):617–622, 2004.

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[Cle98] Bradley J Clement. Learning harmonic progression using Markov models, 1998.

[Coh12] Richard Cohn. *Audacious Euphony : Chromatic harmony and the triad's second nature.* Oxford University Press, 2012.

[Con03] Darrell Conklin. Music generation from statistical models. *Journal of New Music Research*, 45, 6 2003.

[Coo59] Deryck Cooke. *The Language of Music.* Oxford University Press, 1959.

[CSŞ11] Parag Chordia, Avinash Sastry, and Sertan Şentürk. Predictive tabla modelling using variable-length Markov and hidden Markov models. *Journal of New Music Research*, 40(2):105–118, 2011.

[CUF16] Hang Chu, Raquel Urtasun, and Sanja Fidler. Song from PI : A musically plausible network for pop music generation, 2016.

[DHRVW09] W Bas De Haas, Martin Rohrmeier, Remco C Veltkamp, and Frans Wiering. Modeling harmonic similarity using a generative grammar of tonal harmony. In *Proceedings of the 10th International Conference on Music Information Retrieval (ISMIR)*, 2009.

[DHYY18] Hao-Wen Dong, Wen-Yi Hsiao, Li-Chia Yang, and Yi-Hsuan Yang. Musegan: Multi-track sequential generative adversarial networks for symbolic music generation and accompaniment. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), 4 2018.

[DS98] Jack Douthett and Peter Steinbach. Parsimonious graphs: A study in parsimony, contextual transformations, and modes of limited transposition. *Journal of Music Theory*, 42(2):241–263, 1998.

[DuB03] Roger Luke DuBois. *Applications of generative string-substitution systems in computer music.* Phd thesis, Columbia University, 2003.

[EBPM13] Arne Eigenfeldt, Oliver Bown, Philippe Pasquier, and Aengus Martin. Towards a taxonomy of musical metacreation: Reflections on the first musical metacreation weekend. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 9(1), 2013.

[Ecm20] Ecma International. *ECMAScript 2020 Language Specification*, 262 edition, 2020.

[Eib16] Lukas Eibensteiner. Procedural music generation with grammars. Bachelor's Thesis, TU Wien, 2016.

[ES02] Douglas Eck and Juergen Schmidhuber. Finding temporal structure in music: Blues improvisation with LSTM recurrent networks. In *Proceedings of the 12th IEEE Workshop on Neural Networks for Signal Processing*, pages 747–756, 2002.

[Eul74] Leonhard Euler. De harmoniae veris principiis per speculum musicum repraesentatis. *Novi Commentarii academiae scientiarum Petropolitanae*, 18:330–353, 1774.

[Fra04] Judy A Franklin. Recurrent neural networks and pitch representations for music tasks. In *Proceedings of the 17th International Florida Artificial Intelligence Research Society Conference*, pages 33–37, 2004.

[Fra05] Judy A Franklin. Jazz melody generation from recurrent network learning of several human melodies. In *Proceedings of the 18th International Florida Artificial Intelligence Research Society Conference*, pages 57–62, 2005.

[Fra06] Judy A Franklin. Recurrent neural networks for music computation. *Informs Journal on Computing*, 18(3):321–338, 2006.

[FS01] Mary Farbood and Bernd Schöner. Analysis and synthesis of Palestrina -style counterpoint using Markov chains. In *Proceedings of the 2001 International Computer Music Conference*, pages 471–474, 2001.

[FV13] Jose D Fernández and Francisco Vico. AI methods in algorithmic composition: A comprehensive survey. *Journal of Artificial Intelligence Research*, 48:513–582, 2013.

[GC07] Édouard Gilbert and Darrell Conklin. A probabilistic context-free grammar for melodic reduction. In *Proceedings of the International Workshop on Artificial Intelligence and Music, 20th International Joint Conference on Artificial Intelligence (IJCAI), Hyderabad, India*, pages 83–94, 2007.

[Gog06] Michael Gogins. Score generation in voice-leading and chord spaces. In *Proceedings of the 2006 International Computer Music Conference*, 2006.

[GS15] Mathieu Giraud and Slawek Staworko. Modeling musical structure with parametric grammars. In *Mathematics and Computation in Music*, pages 85–96. Springer, 2015.

[GTK10] Jon Gillick, Kevin Tang, and Robert M Keller. Machine learning of jazz grammars. *Computer Music Journal*, 34(3):56–66, 2010.

[GW13] Mark Granroth-Wilding. *Harmonic analysis of music using combinatory categorial grammar.* Phd thesis, The University of Edinburgh, 2013.

[Han60] Howard Hanson. *Harmonic materials of modern music: Resources of the tempered scale.* Appleton Century Crofts, 1960.

[HCC17] Dorien Herremans, Ching-Hua Chuan, and Elaine Chew. A functional taxonomy of music generation systems. *ACM Computing Surveys*, 50(5), 2017.

[HFM92] Hermann Hild, Johannes Feulner, and Wolfram Menzel. HARMONET: A neural net for harmonizing chorales in the style of J. S. Bach. In *Advances in neural information processing systems*, 1992.

[HG91] Andrew Horner and David E Goldberg. Genetic algorithms and computer-assisted music composition. In *Proceedings of the 4th International Conference on Genetic Algorithms*, 1991.

[HHR$^+$19] Cheng-Zhi Anna Huang, Curtis Hawthorne, Adam Roberts, Monica Dinculescu, James Wexler, Leon Hong, and Jacob Howcroft. The Bach Doodle: Approachable music composition with machine learning at scale. In *Proceedings of the 18th International Society for Music Information Retrieval Conference*, 2019.

[HHT06] Masatoshi Hamanaka, Keiji Hirata, and Satoshi Tojo. Implementing "a generative theory of tonal music". *Journal of New Music Research*, 35(4):249–277, 2006.

[HHT07] Masatoshi Hamanaka, Keiji Hirata, and Satoshi Tojo. FATTA: Full automatic time-span tree analyzer. In *Proceedings of the International Computer Music Conference*, volume 1, pages 153–156, 2007.

[HJ14] Paul Hudak and David Janin. Tiled polymorphic temporal media. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design*, pages 49–60, 2014.

[HMGW96] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation: An algebra of music. *Journal of Functional Programming*, 6(3):465–483, 1996.

[Hol80] Steven R Holtzman. A generative grammar definition language for music. *Journal of New Music Research*, 9(1):1–48, 1980.

[Hol81] SR Holtzman. Using generative grammars for music composition. *Computer Music Journal*, 5(1):51–64, 1981.

[Hor94] Damon Horowitz. Generating rhythms with genetic algorithms. In *AAAI-94: Proceedings of the 12th National Conference on Artificial Intelligence*, volume 94, page 1459. American Association for Artificial Intelligence, 1994.

[Hör98] Dominik Hörnel. MELONET I: Neural nets for inventing baroque-style chorale variations. In *Proceedings of the 10th International Conference on Neural Information Processing Systems*, pages 887–893, 1998.

[Hör04] Dominik Hörnel. Chordnet: Learning and producing voice leading with neural networks and dynamic programming. *Journal of New Music Research*, 33(4):387–397, 2004.

[HPE10] Andrew Hawryshkewich, Philippe Pasquier, and Arne Eigenfeldt. Beatback: A real-time interactive percussion system for rhythmic practise and exploration. In *Proceedings of the 2010 Conference on New Interfaces for Musical Expression*, 2010.

[HPN17] Gaëtan Hadjeres, François Pachet, and Frank Nielsen. Deepbach: A steerable model for Bach chorales generation. In *Proceedings of the 34th International Conference on Machine Learning*, pages 1362–1371, 2017.

[HQ18] Paul Hudak and Donya Quick. *The Haskell School of Music: From signals to Symphonies.* Cambridge University Press, 2018.

[HQSWC15] Paul Hudak, Donya Quick, Mark Santolucito, and Daniel Winograd-Cort. Real-time interactive music in Haskell. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design*, pages 15–16, 2015.

[Hud04] Paul Hudak. An algebraic theory of polymorphic temporal media. In *International Symposium on Practical Aspects of Declarative Languages*, pages 1–15. Springer, 2004.

[HWSC15] Dorien Herremans, Stéphanie Weisser, Kenneth Sörensen, and Darrell Conklin. Generating structured music for bagana using quality metrics based on Markov models. *Expert Systems with Applications*, 42(21):7424–7435, 2015.

[IMAW15] Martin Ilčík, Przemyslaw Musialski, Thomas Auzinger, and Michael Wimmer. Layer-based procedural design of façades. *Computer Graphics Forum*, 34(2):205–216, 2015.

[JCS16] Diego Jesus, António Coelho, and António Augusto Sousa. Layered shape grammars for procedural modelling of buildings. *The Visual Computer*, 32:933–943, 2016.

[KB89] Jim Kippen and Bernard Bel. The identification and modelling of a percussion language, and the emergence of musical concepts in a machine-learning experimental set-up. *Computers and the Humanities*, 23:199–214, 1989.

[Kei78] Allan Keiler. Bernstein's "the unanswered question" and the problem of musical competence. *The Musical Quarterly*, 64(2):195–222, 1978.

[KM07] Robert M Keller and David R Morrison. A grammatical approach to automatic improvisation. In *Fourth Sound and Music Conference*, 2007.

[KMDH13] Hendrik Vincent Koops, JoséPedro Magalhães, and W Bas De Haas. A functional approach to automatic melody harmonisation. In *Proceedings of the 1st ACM SIGPLAN workshop on Functional art, music, modeling & design*, pages 47–58, 2013.

[Knu90] Donald E Knuth. The genesis of attribute grammars. In *Attribute Grammars and Their Applications*, pages 1–12. Springer, 1990.

[KPFKV12] Maximos A Kaliakatsos-Papakostas, Andreas Floros, Nikolaos Kanellopoulos, and Michael N Vrahatis. Genetic evolution of L and FL-systems for the production of rhythmic sequences. In *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pages 461–468, 2012.

[KU08] Phillip B Kirlin and Paul E Utgoff. A framework for automated Schenkerian analysis. In *Proceedings of the 9th International Conference on Music Information Retrieval*, 2008.

[LA85] Gareth Loy and Curtis Abbott. Programming languages for computer music synthesis, performance, and composition. *ACM Computing Surveys*, 17(2):235–265, 1985.

[Lan89] Peter Langston. Six techniques for algorithmic music composition. In *Proceedings of the International Computer Music Conference*, volume 60, 1989.

[Ler04] Fred Lerdahl. *Tonal pitch space*. Oxford University Press, 2004.

[LG73] David Lidov and Jim Gabura. A melody writing algorithm using a formal language model. *Computer Studies in the Humanities*, 4(3-4):138–148, 1973.

[Lia16] Feynman Liang. Bachbot: Automatic composition in the style of Bach chorales. Master's thesis, University of Cambridge, 2016.

[Lin68] Aristid 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.

[LJ$^+$83] Fred Lerdahl, Ray Jackendoff, et al. *A generative theory of tonal music*, volume 1996. The MIT Press, 1983.

[LRB09] Bruno F Lourenço, Jos é C L Ralha, and Márcio CP Brandao. L-systems, scores, and evolutionary techniques. In *Proceedings of the 6th Sound and Music Computing Conference*, pages 113–118, 2009.

[LS70] Björn Lindblom and Johan Sundberg. Towards a generative theory of melody. *STL-QPSR*, 10(4):53–86, 1970.

[Man06] Stelios Manousakis. Musical L-systems. Master's thesis, Koninklijk Conservatorium, 2006.

[Mar10] Alan Marsden. Schenkerian analysis by computer: A proof of concept. *Journal of New Music Research*, 39(3):269–289, 2010.

[McC96] Jon McCormack. Grammar based music composition. *Complex systems*, 96:321–336, 1996.

[McI94] Ryan A McIntyre. Bach in a box: The evolution of four part baroque harmony using the genetic algorithm. In *Proceedings of the 1st IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 852–857. IEEE, 1994.

[MdH11] José Pedro Magalhães and W Bas de Haas. Functional modelling of musical harmony: an experience report. *ACM SIGPLAN Notices*, 46(9):156–162, 2011.

[Mel19] Orestis Melkonian. Music as language: putting probabilistic temporal graph grammars to good use. In *Proceedings of the 7th ACM SIGPLAN International Workshop*, pages 1–10, 2019.

[Mic16] Microsoft Corporation. *TypeScript Language Specification*, 1 2016.

[Mic21] Microsoft Corporation. *TypeScript Handbook*, 3 2021.

[Moo72] James Anderson Moorer. Music and computer composition. *Communications of the ACM*, 15(2):104–113, 1972.

[Mor07] Nigel Morgan. Transformation and mapping of L-systems data in the composition of a large-scale instrumental work. In *Proceedings of the European Conference on Artificial Life*, 2007.

[Moz94] Michael C Mozer. Neural network music composition by prediction: Exploring the benefits of psychoacoustic constraints and multi-scale processing. *Connection Science*, 6(2):247–280, 1994.

[MS94] Stephanie Mason and Michael Saffle. L-systems, melodies and musical structure. *Leonardo Music Journal*, 4:31–38, 1994.

[MWH$^+$06] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. *ACM Transactions on Graphics*, 25(3):614–623, 2006.

[Nel96] Gary Lee Nelson. Real time transformation of musical material with fractal algorithms. *Computers & Mathematics with Applications*, 32(1):109–116, 1996.

[Nie09] Gerhard Nierhaus. *Algorithmic composition: paradigms of automated music generation*. Springer, 2009.

[ODZ$^+$16] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio, 2016.

[PATW99] Somnuk Phon-Amnuaisuk, Andrew Tuson, and Geraint Wiggins. Evolving musical harmonisation. In *Artificial Neural Nets and Genetic Algorithms: Proceedings Of The International Conference In Portoroz, Slovenia*, pages 229–234. Springer, 1999.

[PDBB97] John Polito, Jason M Daida, and Tommaso F Bersano-Begey. Musica ex machina: Composing 16th-century counterpoint with genetic programming and symbiosis. In *International Conference on Evolutionary Programming*, pages 113–123. Springer, 1997.

[Pes12] Pedro Pestana. Lindenmayer systems and the harmony of fractals. *Chaotic Modeing and Simulation*, 1(1):91–99, 2012.

[Pet12] Simon Petitjean. Describing music with metagrammars. In *International Workshop on Constraint Solving and Language Processing*, pages 152–165. Springer, 2012.

[PL90] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer, 1990.

[PMW02] Marcus Pearce, David Meredith, and Geraint Wiggins. Motivations and methodologies for automation of the compositional process. *Musicae Scientiae*, 6(2):119–147, 2002.

[Pru86a] Przemyslaw Prusinkiewicz. Graphical applications of L-systems. In *Proceedings of Graphics Interface '86 / VisionInterface '86*, pages 247–253, 1986.

[Pru86b] Przemyslaw Prusinkiewicz. Score generation with L-systems. In *Proceedings of the International Computer Music Conference*, 1986.

[PW98] George Papadopoulos and Geraint Wiggins. A genetic algorithm for the generation of jazz melodies. In *Proceedings of STEP 98*, 1998.

[PW99] George Papadopoulos and Geraint Wiggins. AI methods for algorithmic composition: A survey, a critical view and future prospects. In *Proceedings of the AISB symposium on musical creativity*, volume 124, pages 110–117, 1999.

[PW01] Marcus Pearce and Geraint Wiggins. Towards a framework for the evaluation of machine compositions. In *Proceedings of the AISB'01 Symposium on Artificial Intelligence and Creativity in the Arts and Science*, 2001.

[QH13a] Donya Quick and Paul Hudak. Grammar-based automated music composition in Haskell. In *Proceedings of the 1st ACM SIGPLAN workshop on Functional art, music, modeling & design*, pages 59–70. Association for Computing Machinery, 2013.

[QH13b] Donya Quick and Paul Hudak. A temporal generative graph grammar for harmonic and metrical structure. In *Proceedings of the International Computer Music Conference*, 2013.

[Qui14] Donya Quick. *Kulitta: A Framework for Automated Music Composition.* Yale University, 2014.

[Qui15] Donya Quick. Composing with kulitta. In *Proceedings of the International Computer Music Conference*, 2015.

[Roa82] Curtis Roads. A conversation with james a. moorer. *Computer Music Journal*, 6(4):10–21, 1982.

[Roh07] Martin Rohrmeier. A generative grammar approach to diatonic harmonic structure. In *Proceedings of the 4th sound and music computing conference*, pages 97–100, 2007.

[Roh11] Martin Rohrmeier. Towards a generative syntax of tonal harmony. *Journal of Mathematics and Music*, 5(1):35–53, 2011.

[RW79] Curtis Roads and Paul Wieneke. Grammars as representations for music. *Computer Music Journal*, pages 48–55, 1979.

[Sch35] Heinrich Schenker. *Der Freie Satz.* Universal Edition, 1935.

[Sip97] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publishing, 1997.

[SJM14] Kirill A Sidorov, Andrew Jones, and A David Marshall. Music analysis as a smallest grammar problem. In *Proceedings of the 15th Conference of the International Society for Music Information Retrieval (ISMIR 2014)*, pages 301–306, 2014.

[SM15] Michael Schwarz and Pascal Müller. Advanced procedural modeling of architecture. *ACM Transactions on Graphics*, 34(4):1–12, 2015.

[SMB08] Ian Simon, Dan Morris, and Sumit Basu. MySong: automatic accompaniment generation for vocal melodies. In *Proceedings of the 2008 Conference on Human Factors in Computing Systems*, pages 725–734, 2008.

[Smo80] Stephen W Smoliar. A computer aid for Schenkerian analysis. *Computer Music Journa*, 4(2):41–59, 1980.

[Ste84] Mark J Steedman. A generative grammar for jazz chord sequences. *Music Perception*, 2(1):52–77, 1984.

[Ste96] Mark Steedman. The blues and the abstract truth: Music and mental models. *Mental models in cognitive science*, pages 305–318, 1996.

[Sti80] George Stiny. Introduction to shape and shape grammars. *Environment and Planning B: Planning and Design*, 7(3):343–351, 1980.

[Sti82] George Stiny. Spatial relations and grammars. *Environment and Planning B: Planning and Design*, 9(1):113–114, 1982.

[TBWD01] Michael W Towsey, Andrew R Brown, Susan K Wright, and Joachim Diederich. Towards melodic extension using genetic algorithms. *Educational Technology & Society*, 4(2):54–65, 2001.

[TD08] Axel Tidemann and Yiannis Demiris. A drum machine that learns to groove. In *Annual Conference on Artificial Intelligence*, pages 144–151. Springer, 2008.

[TF12] Tsubasa Tanaka and Kiyoshi Furukawa. Automatic melodic grammar generation for polyphonic music using a classifier system. In *Proceedings of the 9th Sound and Music Computing Conference*, 2012.

[TI00] Nao Tokui and Hitoshi Iba. Music composition with interactive evolutionary computation. In *Proceedings of the 3rd International Conference on Generative Art*, 2000.

[Tod89] Peter M Todd. A connectionist approach to algorithmic composition. *Computer Music Journal*, 13(4):27–43, 1989.

[Wil09] Adam James Wilson. A symbolic sonification of L-systems. In *Proceedings of the International Computer Music Conference*, 2009.

[Win68] Terry Winograd. Linguistics and the computer analysis of tonal harmony. *Journal of Music Theory*, 12(1):2–49, 1968.

[WS05] Peter Worth and Susan Stepney. Growing music: musical interpretations of L-systems. In *Workshops on Applications of Evolutionary Computation*, pages 545–550. Springer, 2005.

[WWSR03] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. *ACM Transactions on Graphics*, 22(3):669–677, 2003.

[YCY17] Li-Chia Yang, Szu-Yu Chou, and Yi-Hsuan Yang. MidiNet: A convolutional generative adversarial network for symbolic-domain music generation, 2017.

[YG07] Liangrong Yi and Judy Goldsmith. Automatic generation of four-part harmony. In *Proceedings of the 5th UAI Bayesian Modeling Applications Workshop*, 2007.

[YL20] Li-Chia Yang and Alexander Lerch. On the evaluation of generative models in music. *Neural Computing and Applications*, 32:4773–4784, 2020.

[You17] Halley Young. A categorial grammar for music and its use in automatic melody generation. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design*, pages 1–9. Association for Computing Machinery, 2017.

[Yus09] Jason Yust. The geometry of melodic, harmonic, and metrical hierarchy. In *International Conference on Mathematics and Computation in Music*, pages 180–192, 2009.

[Zas05] Neal Zaslaw. *Essays in Honor of László Somfai on His 70th Birthday. Studies in the Sources and the Interpretation of Music*, chapter Mozart's Modular Minuet Machine, pages 219–235. Scarecrow Press, 01 2005.

[ZF14] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *Proceedings of the 13th European Conference on Computer Vision*, pages 818–833. Springer, 2014.

[ZXJ$^+$13] Hao Zhang, Kai Xu, Wei Jiang, Jinjie Lin, Daniel Cohen-Or, and Baoquan Chen. Layered analysis of irregular facades via symmetry maximization. *ACM Transactions on Graphics*, 32(4), 2013.