MusAssist: A Domain Specific Language for Music

Ilana Shapiro
issa2018@mymail.pomona.edu

March 30, 2022

Contents

Chapter 1

Introduction

Domain specific languages, or DSLs, are programming languages tailored towards a specific application. Formally, a DSL is defined as "a computer programming language of limited expressiveness focused on a particular domain." This definition encompasses four critical features: (1) the computer programming language (PL) itself, (2) a "language nature" (i.e. a sense of fluency from the way individual expressions can be combined), (3) limited expressiveness (since the purpose of a DSL is to be used in an particular domain, it should not have the complexity of a general purpose language, or GPL), and (4) domain focus (the motivation to create the DSL in the first place). Note that domain focus is simply a consequence of the limited expressiveness of the DSL (Fowler and Parsons [2011]).

DSLs can generally be placed into three categories: external DSLs, internal DSLs, and language workbenches. An *external DSL* is a PL that is separated from the primary PL of its application. It normally uses a custom syntax, but sometimes borrows the syntax of an existing PL (an example of this is XML). The code for an external DSL is conventionally parsed by code of the host application using text parsing methods. Besides XML, common external DSLs include regular expressions, SQL, and Awk (Fowler and Parsons [2011]).

An *internal DSL* is embedded in an already existing GPL, making use of its syntax and semantics. A program written in an internal DSL is already valid code in the host GPL, but only makes use of a small subset of the GPL's powerful expressive features in order to handle a specific aspect of the domain. Thus, a "custom feel" is achieved using the GPL. Lisp is the hallmark GPL for creating internal DSLs, but Ruby is also common. Rails, one of Ruby's best-known frameworks, is frequently considered to be a collection of internal DSLs (Fowler and Parsons [2011]).

Finally, a *language workbench* is a customized IDE for building and defining DSLs (Fowler and Parsons [2011]).

Music itself has a highly structured framework, and a musical score itself can be thought of as having many of the formal features of a PL. With the increased flexibility afforded to a DSL via its limited expressiveness, it can be much more effectively tailored to the application (i.e. music) than an GPL could be. Depending on the goals of the programmer, a DSL can be used to target a particular aspect of music, whether that be notation, algorithmic composition, signal processing,

or live coding with music performance.

MusAssist is an external DSL devised as a compositional aid for music notation by incorporating concepts from music theory. It attempts to organically model a composer's flow of thought by modeling its syntax after the musical structures a composer conceives when writing. Users write out musical elements and expressions in MusAssist's simple and straightforward syntax much in the same way they would when composing. In other words, users *describe* a composition in MusAssist, and MusAssist writes out the music via these instructions. Users can compose notes (including rests) and custom chords in the octave and key of choice. They can also specify templates for chords (all triads and seventh chords), harmonic sequences (chosen from Ascending Fifths, Descending Fifths, Ascending 5-6, and Descending 5-6) of a desired length, and cadences (chosen from Perfect Auth. The musical expression described by the template will then be written by the MusAssist compiler. The compiler is written in Haskell.

The target language of the MusAssist compiler is MusicXML, an internal DSL that is an extension of XML. MusicXML is interpreted by most major notation software programs (such as MuseScore). Thus, once a user has described a composition in MusAssist, they can open the resulting MusicXML file in MuseScore or another program for further customization and editing. MusAssist does not attempt to replace existing DSLs. Rather, it fills a unique niche in that it *assists* users in music composition by providing them with a set of easy-to-use instructions that would otherwise be tedious to write out by hand in a musical score. This is why MusAssist is compiled to MusicXML rather than an uneditable PDF format. MusAssist may also be particularly helpful to music students as an educational tool where they can easily see the relationship between a musical expression and its written form, such as a harmonic sequence template and the actual chords that result from it.

In order to use MusAssist, the user need not have any understanding of computing, though they should have a solid knowledge of music theory up through chord and cadence types, as well as harmonic sequences. In order to comprehend this paper, in addition to music theory, users should have a background in basic programming languages theory and compilers.

Chapter 2

Background

DSLs for music have not been studied as extensively as other application domains, but are a fascinating area of inquiry that explores the expressive power of languages and pushes the boundaries of computational creativity. The era of music DSLs began in 2008 with Ge Wang's invention of the ChucK audio processing language. ChucK is actually a GPL broadly tailored towards music, as it spans the application domains of "methods for sound synthesis, physical modeling of real-time world artifacts and spaces (e.g., musical instruments, environmental sounds), analysis and information retrieval of sound and music, to mapping and crafting of new controllers and interfaces (both software and physical) for music, algorithmic/generative processes for automated or semi-automatic composition and accompaniment, [and] real-time music performance." With ChucK, Wang developed a language that is "expressive and easy to write and read with respect to time and parallelism," thus providing users with a "platform for precise audio synthesis/analysis and rapid experimentation in computer music." (Wang [2008]).

A multitude of programming paradigms have been used for music DSLs, including declarative programming, functional programming, object-oriented programming, synchronous programming, and subcategories of synchronous programming called strong-timed programming and mostly-strongly-timed programming. The choice of programming paradigm for a music DSL depends on the specific musical subdomain the language targets. For instance, a DSL intended to handle musical signal processing or live coding (i.e. applications that have to do with the time dimension of music) would benefit from using one of the synchronous programming paradigms.

Notably, though, the choice to make a DSL external or internal is not related to the choice of programming paradigm. In general, according to Cuadrado, Izquierdo, and Molina, internal DSLs are preferred over external DSLs when there is no significant tradeoff in performance, the runtime infrastructure of the parent language is easily reused, and the target audience is comfortable using the parent language. Otherwise, an external DSL would most likely be a better choice. These same considerations apply when designing a DSL for music. (Cuadrado et al. [2012]).

This chapter serves as a review of the existing literature on the better-known DSLs for music. The chapter is organized as follows. Section 2.1 examines the programming paradigms commonly utilized in DSLs for music. Section 2.2 is a review of some common external DSLs for music, and Section 2.3 looks at examples of existing internal DSLs for music. Finally, Sections 2.4 and

Section 2.5 consider applications of music DSLs in the fields of algorithmic composition and human-computer interaction (HCI), respectively.

2.1 Programming Paradigms Used in Music DSLs

2.1.1 Declarative Programming

In contrast to the more commonly encountered paradigm of imperative programming, declarative programming is a programming model that eliminates control flow in favor of simply stating, or declaring, what the desired action or result is. Declarative programming is commonly used by DSLs in database management, and relies on pre-existing language features to execute the desired action without relying on control flow structures such as conditional logic and loops. In other words, declarative programming emphasizes *what* the final result is, while imperative programming focuses on *how* to get there. As an analogy, if someone hails a taxi, they *declare* to the driver where they wish to go – they do not give him turn-by-turn (i.e. imperative) directions. (Bertram [2021])

Musical markup languages such as Michael Good's MusicXML and LilyPond fall under this category. MusicXML is derived from XML (itself a DSL), and seeks to solve the music interchange problem between the various musical representation formats. Good [2013] LilyPond is not XML-based. Rather, it has its own syntax, and compiles to a PostScript or PDF format that can be printed or uploaded on the Internet.

2.1.2 Functional Programming

Functional programming is a programming paradigm centered around building functions for immutable values. It emphasizes *pure functions*, or functions that never alter variables but instead produce new ones as output. Pure functions can also be thought of as functions without *side effects*, or when the function neither relies on nor modifies anything outside of its parameters. The GPL Haskell is perhaps the most famous example of a functional PL (Joury [2020]).

Du Bois and Ribeiro describe HMusic, a DSL for music programming and live coding that is embedded in Haskell (thus giving HMusic the power of functional programming). HMusic provides abstractions for patterns and tracks, defined inductively. The inspiration behind HMusic was to let artists express themselves through software. The abstractions for patterns and tracks in HMusic greatly resemble grids from sequencers, drum machines, and digital audio workstations (Bois and Ribeiro [2019]).

2.1.3 Object-Oriented Programming

Rather than functions, object-oriented programming (or OOP) is centered around the objects that the developers want to create and use. The building blocks of OOP are classes (blueprints for objects), objects (instances of classes with custom-defined data), methods that describe an object's behavior, and attributes that reflect the state of the object. OOP's main principles are encapsulation (i.e. data-hiding – all important information is hidden within an object with only the most

important data exposed), abstraction (objects only expose internal mechanisms that are useful and generalizable for other objects), inheritance (in which classes reuse code from other classes), and polymorphism (in which objects can share behaviors and assume many forms) (Gillis and Lewis [2021]).

Nishino et al describe LC, an external DSL with dynamic and strong typing for computer music. LC is an object-oriented PL and is $prototyped - based$ (as opposed to class-based) (Nishino et al. [2013]). In a prototype-based language, "each object defines its own behavior and has a shape of its own." This is in contrast to class-based languages like Java, where "each object is an instance of a specific class." In particular, prototype-based languages allow for *slots* (i.e. fields/methods) to be added to an object dynamically, after it has been created. Prototype-based languages therefore open up large amounts of flexibility and tolerance in relation to the dynamic modification of the system at runtime. LC adopts prototype-based programming at the levels of compositional algorithms and sound synthesis (specifically, the user can build and modify a unit-generator graph dynamically) (Nishino et al. [2014]).

2.1.4 Synchronous Programming

Synchronous programming is a programming paradigm in which operations take place sequentially, or linearly, as opposed to asychnronous programming, where operations occur in parallel. This means that in synchronous programming, long-running operations can be "blocking" – i.e. the program cannot proceed to the next operation until the current operation has completed and returned some outcome (DeepSource). Synchronous PLs are often aimed towards programming reactive systems (Petit and Serrano [2020]).

Petit and Serrano describe Skini, a programing methalodogy and execution environment they created for "interactive structured music," where the composer would program their scores in the HipHop.js synchronous reactive language. The scores are then executed (i.e. played) live, and involve audience interaction. The purpose of Skini is to help composers create a balance between the precise determinism of written composition and the nondeterminism of social interaction. Skini uses synchronous DSLs rather than GPLs as the "temporal constructs" of synchronous PLs (parallelism, sequence, synchronization, and preemption) can directly represent musical scores, and their relative flexibility allows composers to easily try out different ideas (Petit and Serrano [2020]).

2.1.5 Strongly-Timed Programming

Strongly-timed programming is a type of synchronous programming first introduced by Ge Wang in 2008 in his development of the ChucK audio programming language. He defines it as "well-defined separation of synchronous logical time from real-time" which helps the user to debug, specify, and reason about programs written in the language. Thus, one can create programs without having to consider external factors like "machine speed, portability and timing behavior across different systems." The powerful deterministic concurrency offered by this model allows for extremely tightly woven control and audio computation, thus giving rise to a DSL that allows the programming to transition seamlessly from digital signal processing, at the sample level, to more "gestural levels of control." (Wang [2008]).

Nishimo et al. build upon Wang's work in strongly-timed programming. They further refine the

definition of strongly-timed programming to be a variation of synchronous programming integrating explicit control of logical synchronous time into an imperative PL in order to achieve precise timing behavior that is predicated on the "ideal synchronous hypothesis," in which that "all computation and communications are assumed to take zero time (that is, all temporal scopes are executed instantaneously)." Nishimo et al.'s DSL LCSynth, the parent language of their object-oriented DSL LC, uses strongly-timed programming address the issue of imprecise timing behavior in microsound synthesis (Nishino [2012]).

Nishino et al. go on to introduce yet another paradigm called *mostly-strongly-timed programming* that is an extension of strongly-timed programming. In mostly-strongly-timed programming, in addition to the principles of strongly-timed-programming, there is also support for explicit context switching between synchronous (i.e. non-preemptive) behavior and asynchronous (i.e. preemptive) behavior whose execution can be suspended at any arbitrary time. Nishino et al.'s object-oriented DSL LC makes use of mostly-strongly-timed programming (Nishino [2012]).

2.2 Survey of External DSLs for Music

2.2.1 LilyPond

LilyPond is an external DSL created by Han-Wen Nienhuys, Jan Nieuwenhuizen that originally began as their personal project. It features a "modular, extensible and programmable compiler" to generate music notation of excellent quality. It supports the mixing of text and music elements, and . Like MusicXML, it is a DSL aimed towards music notation. Unlike MusicXML, LilyPond does not consider the music interchange problem. Rather, it focuses on automated music printing. The compiler produces a printable PostScript or PDF file by taking in a file with a formal representation of the desired music. Its implementation uses the language Scheme (a LISP language).

The LilyPond compiler has four steps:

1. Parse into an abstract syntax tree
2. Musical elements are translated (i.e. interpreted) into graphical elements in an unformatted score
3. Format the score
4. Write the formatted score to an output file

The input to the first step is a series of text-based *musical expressions*, or fragments of music with set durations. Simple music expressions are combined to make more complex ones. The input format also supports identifiers that allow the user to re-use an expression multiple times.

In the second step, which the authors call "interpreting", a plugin architecture with plugins called *engravers* performs the conversion. Each engraver handles a single specific task, creating a modular architecture that allows for ease of maintaining and extending the program. This is the step in which context sensitive information, like key signature and current beat in the measure, is handled so that barlines and accidentals are printed correctly.

In this third step, the layout is determined. The input to this step is the unformatted score, or a collection of graphical objects. Tags called abstract graphical objects store information about

constrainment, alignment, and element spacing. Nienhuys and Nieuwenhuizen [2003].

2.2.2 PyTabs

Simic et al. present PyTabs, a DSL they designed and created for simplified musical notation. PyTabs allows the user to describe a composition comprising many sequences that can be provided in tablature or chord notation. PyTabs also provides functionality for playing a piece written in it. The authors propose a solution via PyTabs to problems in simplified music notation (specifically, the visual problem of tablature notation, and the lack of standardization of how to specify note duration in tablature notation) by standardizing these issues into a formal language. They used Python to implement PyTabs, but PyTabs is not embedded in Python; it is an external DSL (Simić et al. [2015]).

In PyTabs, the logic for parsing tablature notation is extracted into a generic parser, and then per-instrument parsing is defined later in the concrete implementation. Chord construction consists of one of the 12 semitones, a number representing octave, and a decoration (i.e. major, minor) that indicates the quality of the chord. In the future, the authors plan to add support for more instruments, as well as the ability to generated standard musical notation from PyTabs, and vice versa (Simić et al. [2015]).

2.2.3 LCSynth

Recall from Section 2.4 the DSL LSynth, created by Nishimo et al, that was inspired by Wang's strongly-timed GPL ChucK. In the background information to LSynth, Nishimo discusses the unit-generator concept, a software module from 1960 that uses "conceptually similar functions to standard electronic equipment used for electronic sound synthesis." He also talks about microsound synthesis techniques, which differs from the traditional unit-generator concept in that it does not originate in electronic sound synthesis where the signal is a function of time. Instead, microsound synthesis involves many short sound particles (microsounds) that overlap to create the total sound. The current issue with microsound synthesis is that most computer music PLs are not capable of handling the precise timing behavior required (for instance, most general purpose PLs cannot do this) (Nishino [2012]).

Nishimo et al. came up with a novel abstraction of the sound synthesis framework, as well as a new programming concept for computer music. Nishimo et al. also consider the difficulty of microsound synthesis to be an issue in the abstraction of the underlying sound synthesis software framework in the PL's design, i.e. an incompatibility between the abstractions and the user's understanding of the domain, which they call the "usability problem." (Nishino [2012]).

LCSynth integrates *counterpart entities* to the user's perception of microsound synthesis techniques. This helps remove the *structural misfits* between the representations implemented in the design of currently used music DSLs and the user's individual understanding of microsound synthesis techniques. However, LCSynth is not a stand-alone PL and works solely in the sound-synthesis domain. This is where Nishimo et al.'s DSL LC comes in. It fully integrates LCSynth into its design. (Nishino [2012]).

2.2.4 LC

Nishino et al describe LC, a strongly-timed prototype-based language of dynamic and strong typing that was originally intended to be a control language for LCSynth. LC supports lexical closure, lightweight concurrency (i.e. lexically scoped name binding in a PL with first-class functions), and live computer music. These features enhance dynamism in the language, such as through live coding and rapid prototyping, in order to assist the artistic endeavors of the user. Live coding is "a computational arts practice that involves the real-time creation of generative audio-visual software for interactive multimedia performance." Interpreted scripting languages are preferable for live coding, but DSLs specific to music are often even better (Nishino et al. [2013]).

Nishino et al. discusses the static-typing vs dynamic-typing issue for such DSLs. Dynamic typing is preferred as it is "ideally suited for prototyping systems with changing or unknown requirements" and "indispensable for dealing with truly dynamic program behaviors." Nishimo et al. also chose to make LC strongly, rather than weakly, typed, in order to avoid random bugs arising from implicit type casting. Recall from Section 2.2 that LC is also an object-oriented PL and is $prototyped-based$ (as opposed to class-based). LC is also $duck-typed$ – i.e. it features a framework for "truer polymorphic designs based on what an object can support rather than that object's inheritance hierarchy"). Furthermore, LC supports lightweight concurrency, which enables features such as quick creation/destruction of threads and less memory usage (Nishino et al. [2013]).

LC performs sound synthesis and program execution in logical synchronous time as strongly-typed programs in a single virtual machine. This allows for precise timing behavior. In addition, LC allows for a great amount of flexibility for runtime modification, which makes it suited for applications like live-coding performances on laptops (Nishino et al. [2013]).

Through LC, Nishino et al. also successfully address three current issues in music DSL design: (a) lack of support for dynamic modification of a computer music program, (b) lack of support for precise timing behavior and other time-related features, and (c) the difficulty in microsound synthesis programming. These issues will correspond to the three core features of LC: (1) prototype based programming, both for algorithmic composition and for sound synthesis, (2) the use of the "mostly-strongly-timed programming" concept, and (3) the integration of objects and functions that represent microsounds with the corresponding operations for microsound synthesis. The first two of these features were discussed in Sections 2.2 and 2.4. The third feature utilizes algorithmic scheduling of microsound objects for its microsound synthesis framework. Every sample within a microsound object can be accessed directly, and utility methods are provided in order to manipulate multiple samples simultaneously. Unlike previous work with microsound synthesis, LC's microsound synthesis does not depend on the unit-generator concept, and it also provides a generalized programming paradigm for real-time interactive computer music DSLs (Nishino et al. [2014]).

2.2.5 mimium

Matsuura and Jo describe a novel full-stack DSL called *mimium* (an acronym for **mi***nimal* − **m***usical* − *med***ium**) that combines temporal-discrete control and signal processing in a single PL. It can describe everything from low-level signal processing, all the way to discrete event processing in unified semantics. *mimium* is user-friendly; it has intuitive imperative syntax and supports stateful

functions as Unit Generators just as one would normally define and apply functions. The LLVM compiler infrastructure is used so that the runtime performance equals that of lower-level languages. *mimium* adds the least possible number of features related to sound, and it also implements a general purpose functional PL. Thus, compiler implementation is simplified, and language self-extensibility is increased (Matsuura and Jo [2021]).

Though *mimium* is an external DSL, its syntax is modeled after that of Rust due to the shorter reserved words (suitable for fields) that can perform fast prototyping like music. The basic syntax includes function definitions and calls as well as conditionals (if-else statements). *mimium* also uses the functional model. For instance, a single if statement can be used as an expression that can directly return a value (Matsuura and Jo [2021]).

The architecture of *mimium*'s compiler resembles that of a general purpose functional language. It is based on the *mincaml* compiler and is implemented in C++. Recall that the compiler uses the LLVM infrastructure. In *mimium*, the only compiled functions of the LLVM intermediate representation that depend on the runtime system are one for task registration, and one for getting the internal time. Essentially all other code is compiled on memory and subsequently executed.Thus, *mimium* can achieve similar execution speed to low-level languages like C (Matsuura and Jo [2021]).

Two essential features of *mimium* allow the description of continuous signal processing as well as discrete control processing in unified semantics. The first is the syntax for deterministic task scheduling at the sample level, as well as the implementation of the schedule. For instance, *mimium*'s @ operator can specify the time at which to execute a function. In mimium, @ is combined with the *temporal recursion design pattern* that describes repetitive event processing as a function that calls itself recursively with a time delay. @ is used to increase readability in the implementation of temporal recursion. The second feature is a description of the semantics that are utilized to define the Unit Generator for signal processing. This is achieved by hiding state variables and combining only feedback connections and limited built-in functions with states (Matsuura and Jo [2021]).

2.3 Survey of Internal DSLs for Music

2.3.1 MusicXML

Michael Good's MusicXML is an Internet-friendly XML-based DSL capable of representing Western music notation and sheet music since c. 1600. It acts as an "interchange format for applications in music notation, music analysis, music information retrieval, and musical performance," thus enhancing existing specialized formats for specific use cases. Notable, MusicXML does not attempt to replace other formats tailored even more exactly to such use cases; rather its goal is to support sharing *between* these applications. Good [2013]

Good created MusicXML in an attempt to emulate for online sheet music and music software what the popular MIDI format did for electronic instruments. Good further chose to derive MusixXML from XML in order to help solve the music interchange problem: to create a standardized method to represent complex, structured data in order to support smooth interchange between "musical notation, performance, analysis, and retrieval applications." XML has the desired qualities of "straightforward usability over the Internet, ease of creating documents, and human readability"

that translate directly into the musical domain, and it has the capacity to be both more powerful and more expressive than MIDI format. Good [2001]

Good was inspired by MuseData and Humdrum, two extremely powerful academic music notation formats, for the design of MusicXML (though he went on to add additional features in order to accurately represent music from c. 1850 - present). He was particularly attracted to Humdrum's two-dimensional conception of music by part and time. Unfortunately, the hierarchical structure of native XML is not capable of supporting such a lattice structure, so Good developed an alternative by creating an automatic conversion between the two dimensions. This was achieved by using Extensible Style Sheet Transformations (XSLT) programs. Thus, automatic conversions are supported between part-wise scores (in which measures are nested within parts) and time-wise scores (in which parts are nested within measures). Good [2013]

2.3.2 HMusic

Recall the DSL HMusic, embedded in the functional PL Haskell, that was first introduced in Section 2.1. On a more technical level, HMusic is an algebra (i.e. a set and its associated functions) for creating music patterns. The set of all possible patterns is defined inductively as an ADT (algebraic data type) in Haskell. The user can write recursive Haskell functions to operate on patterns, as patterns themselves are a recursive datatype in HMusic. HMusic also defines the ADT Track, which associates an instrument to a pattern. Tracks can be the parallel composition of two existing tracks, and HMusic has support for multi-tracks consisting of tracks of varying size composed in sequence. Finally, HMusic defines a set of primitives for playing tracks and live coding. Users can play songs written in HMusic, loop tracks, and modify tracks while they are being played. Live coding is implemented through a simple UI based on the concepts of looping and function application (Bois and Ribeiro [2019]).

2.3.3 T-calculus

The T-calculus is a more mathematical approach to an external music DSL design presented by Janin in 2016. He describes a new algebraic model for music writing and programming based on separating the contents of music objects (i.e. what music they define) and the usage of music objects (i.e. how they could be combined). He approaches this from a mathematical perspective. Specifically, he models music objects with a "tiled music graph" that can be combined using the "tiled sum" operator, which is both sequential and parallel. The resulting algebraic structure is an *inverse monoid* (a monoid is a set with an associative binary operation and an identity element, and an inverse monoid is a monoid where each element in the set has a unique multiplicative inverse). To implement this, Janin developed a high level DSL called T-calculus embedded in Haskell. T-calculus is reactive, hierarchical, and modal.(Janin [2016]).

Janin discusses how every music program can itself be viewed as a music score detailing the music to be played. From the perspective of music representation formalisms, music PLs must also be abstract enough to account for the composer's creativity. Janin feels that classical western music notation can itself be seen as a music PL, but with the limitation that it only encodes music but cannot create it. In order to handle all such necessary elements of music representation as well as software engineering requirements, a unified theory of musical objects with algebraic properties

must be described. A *music algebra* defines (1) the basic music objects to be used and (2) the combinators that allow the creation of complex music objects via simple ones. Janin's goal is to define a music algebra from which he will derive a PL and a representation formalism. He does so by using *timed graphs* (directed acyclic graphs with labeled edges) that can then be combined via *tiled composition*. The vertices of timed graphs represent synchronization points, and the edge labels represent the duration of to-be-determined music objects or rests. Tiled composition involves the combining of two musical objects via the *synchronization step* (gluing the input root of the first object to the input root of the second) and the *fusion step* (removing potential ambiguity from the synchronization step). The resulting music algebra is created by adding additional edge attributes to the tiled timed graphs, which preserves the inverse monoid structure. (Janin [2016]).

2.4 Applications in Algorithmic Composition

2.4.1 Skini

Recall the DSL Skini from Section 2.3. Skini is in fact a synchronous internal DSL embedded in the GPL JavaScript that has intriguing applications in algorithmic composition. Music created in the Skini production environment is based on three principles: audience interaction determines what gets played next, the music constitutes sequences of patterns made up of elementary musical elements, and the music (though interactive) must still follow a rigid structure defined by the composer beforehand, which prioritizes artistic consistency over varied interactive performances. Skini may seem like jazz, but unlike jazz, the improvisation comes from the audience, rather than the composer (Petit and Serrano [2020]).

A Skini composition is an example of "synthesis by concatenation" of patterns. (The music is then produced by playing patterns according to audience selections). The composer will organize the patterns into *repetitive groups* and *tanks* (groups without repetition). The program implementing the score will simulate a large state machine. States correspond to group activation (i.e. the ability for the audience to choose a group) and de-activation, and transitions will model the audience interaction and passing time. For the program, the authors use HipHop.js, a synchronous reactive language that is a multi-tier extension of JavaScript, in order to simplify the programming of the complex temporal behaviors inherent to musical scores. HipHop.js executes steps known as *reactions* or *instants*. Steps execute statements in sequence or in parallel; statements communicate using *broadcast signals*, each of which has a unique *present/absent status* (Petit and Serrano [2020]).

Finally, the Skini execution environment of the HipHop.js program has two essential data structures: the "matrix of available groups of patterns" (which is controlled by the execution of the HipHop.js score, and identifies at every moment which groups and tanks are activated) and "pattern queues" (which are provided by the audience and subsequently used by the synthesizers). Skini has been used in real-life live performances, in both jazz and classical settings.(Petit and Serrano [2020]).

2.4.2 Advantages of using DSLs over Virtual Machines (VMs) for Music

It is reasonable to consider that perhaps other avenues of computation, such as Virtual Machines (VMs), would be more effective to work with music than a DSL. However, Sulyok et al. demonstrate otherwise. They look into the effect of embedding various levels of musical data in VM architectures, as well as "phenotype representations" of an algorithmic music composition system. The authors consider two distinct sets of instructions for a linear genetic programming framework: the first is Turing-complete register machine with no knowledge of the nature of its output, and the other is a DSL tailored to music composition, designed around awareness of its output. DSL instructions include transfer, branching, and conditional instructions (Sulyok et al. [2019]).

The "phenotype" is the output of the VM. It comprises a sequence of notes, where a note is defined by duration and pitch. (Linear genetic programming is a kind of genetic programming in with the programs in the population get represented as a linear sequence of instructions from a PL). The fitness metric for the genetic programming framework was derived by the extraction of musical elements from a corpus of Hungarian folk songs. These same elements were extracted from the phenotype and assessed for maximum similarity to the corpus (Sulyok et al. [2019]).

In total, the authors considered six configurations, by using two VM architectures (the is a general-purpose von Neumann machine, or the GP machine, and the other is the DSL machine), and three different pitch schemes. The authors found that the DSL machine outperformed the GP machine even from early generations. Therefore, the instruction set tailed towards music increases the chance that even a truly random genetic string would lead to a desirable output (Sulyok et al. [2019]).

2.5 Applications in HCI

2.5.1 Computational Counterpoint Marks

Martinez presents a novel approach for extending the traditional staff domain (i.e. the domain of Western classical music notation, also known as Common Western Music Notation (CWMN)) to the PL domain. The syntax of his external DSL is intended to model the interaction between people and computers in a live electronics music performance. Therefore, both humans and computers will be able to understand the notation. This allows for a unified music representation for live performance that is human-readable and does not depend on technology (Martinez [2021]).

Martinez extends CWMN to the interactive domain through the creation of abstract-verbal statements called called *Computational Counterpoint Marks* on the score phonetic-dimension. Computational Counterpoint Marks are human-readable annotations acting as expression-marks added to a score. They accurately describe the logic of an interaction between the performer and the computer. This enhances the accompanied graphic signs, leading to a unified and human-readable representation of an interactive piece's performance logic. Furthermore, novel score annotations can also be understood by a computer via the digital score. This means that the musical score itself actually becomes the source code of a piece's performance logic, which allows the computer to perform live during a concert. Finally, Martinez considers the time domain. Specifically, he proposes symbolic rather than absolute time representation that is based on traditional score nota-

tion. Martinez's model maps symbolic time to absolute time through an estimate based on updates during performance about the current symbolic time (Martinez [2021]).

2.5.2 Research through Design

The development of Nishino et al.'s original external DSL LCSynth was approached through the HCI method 'Research through Design' (RtD), in which designers and researchers develop "a product that transforms the world from its current state to a preferred state" and "the artifacts produced in this type of research become design exemplars, providing an appropriate conduit for research findings to easily transfer to the HCI research and practice communities." RtD places an emphasis on the contribution of knowledge to academia rather than the design of a commercial product (Nishino [2012]).

Nishimo et al. use RtD to develop a new DSL focusing on the problem of microsound synthesis. He came up with a novel abstraction of the sound synthesis framework, as well as a new programming concept for computer music. Recall from Section 4.3 that Nishimo et al. consider the potential incompatibility between the abstractions and the user's understanding of the domain that they call the "usability problem." They address this from a formal perspective in their paper "LCSynth: A Strongly-Timed Synthesis Language that Integrates Objects and Manipulations for Microsounds," which the reader may peruse for further reading in this area. (Nishino [2012]).

2.6 Summary

DSLs are a fascinating and effective way to bridge the conceptual gap between computing and music. Since Wang introduced ChucK in 2008, external and internal DSLs utilizing programming paradigms including functional programming, object-oriented programming, synchronous programming, strongly-timed programming, and mostly-strongly-timed programming have made advances in handling issues in microsound synthesis and signal processing, such as in the LCSynth, LC, and mimium languages. On a higher level, DSLs like PyTabs, Skini, and Computational Counterpoint Marks have addressed the areas of music notation representation and algorithmic composition, as well as intersect with the field of HCI. Finally, the novel concepts of live coding and laptop music are considered with the DSLs HMusic and even LC.

Chapter 3

Language Features

MusAssist allows users to write individual notes and rests, as well as chords consisting of custom collections of notes. Drawing upon concepts from music theory, users can also describe chords based on type, quality, and inversion (such a C augmented seventh chord in second inversion). Furthermore, users can describe any of the four primary harmonic sequences (ascending fifths, descending fifths, ascending 5-6, and descending 5-6) of any length and in any local key (i.e. no matter what the current key signature is). MusAssist additionally allows users to describe any of the five primary cadences (perfect authentic, imperfect authentic, plagal, half, and deceptive) in any local key. Finally, users have the ability to change the key signature or move to a new measure at any time. Currently, MusAssist only supports one clef (and one line): a single treble clef line. MusAssist also does not currently support custom tempo or tempo changes. The tempo is set at ♩ = 80bpm.

This chapter details the concrete syntax of MusAssist. In the following sections, parameters take the following form: `<PARAMETER>`

3.1 Rests

The syntax for a rest is as follows

`(rest <DURATION>)`

where `<DURATION>` is taken from one of the following options:

`sixteenth, eighth, dotted_eighth, quarter, dotted_quarter, half, dotted_half, whole`

An example of a MusAssist rest is `(rest dotted_quarter)`

This expression translates to the following when loaded into the MuseScore notation software:

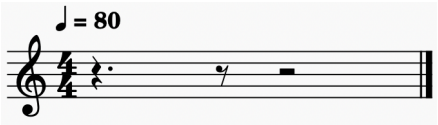

Figure 3.1: Dotted quarter rest

3.2 Notes

The syntax for a note is as follows

`(<NOTENAME><ACCIDENTAL><OCTAVE> <DURATION>)`

where `<NOTENAME>` is taken from one of the following options:

`F,C,G,D,E,A,B`

and `<ACCIDENTAL>` is taken from one of the following options:

`#, ##, b, bb`

`<ACCIDENTAL>` is an optional parameter. If the user leaves it out, the note is considered natural. Notice that double sharps are represented with `##` rather than the conventional $\times$ symbol, as this is not an easily typed out character on a keyboard.

`<OCTAVE>` is taken from one of the following options:

`1,2,3,4,5,6,7,8`

as these are the octaves of a standard piano.

`<DURATION>` is taken from the same options as in Section 3.1.

Examples of MusAssist notes are `(Cbb5 eighth)`, `(D#5 eighth)`, and `(A2 quarter)`

These expressions, respectively, translate to the following when loaded into MuseScore:

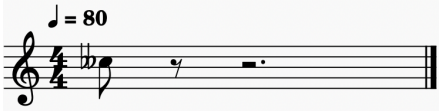

Figure 3.2: Eighth note C𝄫5

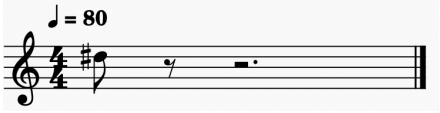

Figure 3.3: Eighth note D♯5

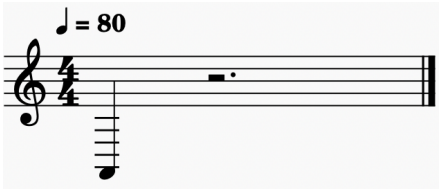

Figure 3.4: Eighth note A2

3.3 Custom Chords

The syntax for a custom chord is as follows

`([<NOTENAME><ACCIDENTAL><OCTAVE>] <DURATION>)`

where `<NOTENAME>`, `<ACCIDENTAL>`, `<OCTAVE>`, and `<DURATION>` are taken from the same options as in Section 3.2, and `<ACCIDENTAL>` is similarly optional, with natural implied when it is left out.

`[<NOTENAME><ACCIDENTAL><OCTAVE>]` indicates an arbitrarily long list supplied by the user, where each of the values are of the form `<NOTENAME><ACCIDENTAL><OCTAVE>`

Examples of MusAssist custom chords are `([Bb5, Dbb6, C5] half)` and `([C##1, E5] sixteenth)`

These expressions, respectively, translate to the following when loaded into in MuseScore:

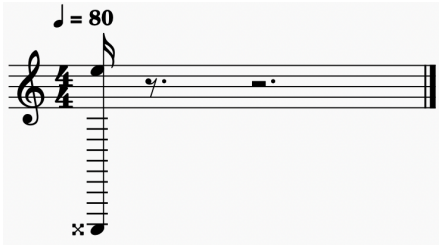

Figure 3.5: Half note chord with notes B♭6, D𝄫6, and C5

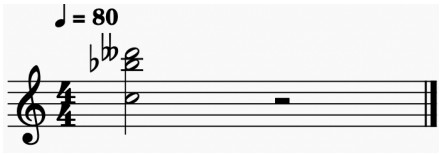

Figure 3.6: Sixteenth note chord with notes C𝄪1 and E5

3.4 Chord Templates

The syntax for a chord template is as follows

`(<NOTENAME><ACCIDENTAL><OCTAVE> <QUALITY> <CHORDTYPE> inv:<INVERSION> <DURATION>)`

where `<NOTENAME>`, `<ACCIDENTAL>`, `<OCTAVE>`, and `<DURATION>` are taken from the same options as in Section 3.2, and `<ACCIDENTAL>` is similarly optional, with natural implied when it is left out.

`<ACCIDENTAL>` is now taken from one of the following restricted options:

`#, b`

This is because chords cannot be built on a double sharp or flat, as such chords may introduce triple sharps or flats, and MusAssist does not support these elements. `<ACCIDENTAL>` does continue remain optional, with natural implied when it is left out.

`<INVERSION>` is taken from one of the following options:

`root, first, second, third`

where `third` is only an option for seventh chords.

`<QUALITY>` is taken from one of the following options:

`maj, min, aug, dim, halfdim`

(which, as one would expect, indicate major, minor, augmented, diminished, and half diminished qualities). Half diminished can only apply to a seventh chord, not to triads.

Finally, `<CHORDTYPE>` is taken from one of the following options:

`triad, seventh`

Examples of MusAssist harmonic sequences are `(C6 min triad inv:first quarter)` and `(F\#4 halfdim seventh inv:third eighth)`

These expressions, respectively, translate to the following when loaded into MuseScore:

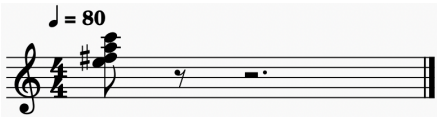

Figure 3.7: C6 minor triad, first inversion

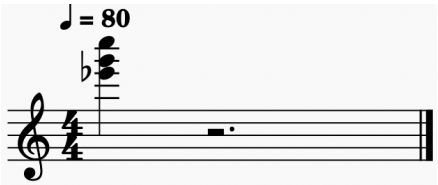

Figure 3.8: F♯4 half diminished seventh, third inversion

3.5 Harmonic Sequences

The syntax for a harmonic sequence is as follows

`(<HARMSEQTYPE> <NOTENAME><ACCIDENTAL><OCTAVE> <QUALITY> <DURATION> length:<LENGTH>)`

where `<NOTENAME>`, `<OCTAVE>`, and `<DURATION>` (but not `<ACCIDENTAL>`) are taken from the same options as in Section 3.2. `<DURATION>` and `<LENGTH>` are not to be confused: `<DURATION>` specifies the length of each individual chord in the harmonic sequence, while `<LENGTH>` specifies the number of chords in the sequence.

Unlike chord templates, here `<QUALITY>` is taken from one of the following restricted options:

`maj, min`

and `<ACCIDENTAL>` is taken from the same restricted options as in **??**. `<ACCIDENTAL>` continues remain optional, with natural implied when it is left out.

This is because together, `<NOTENAME><ACCIDENTAL><OCTAVE> <QUALITY>` determine the local key (and the start octave) of the harmonic sequence. A key can only be major or minor, and a key in MusAssist cannot be described with a double sharp or flat, as such key signatures may introduce triple sharps or flats, which recall are not supported.

`<HARMSEQTYPE>` is taken from one of the following options:

`AscFifths, DescFifths, Asc56, Desc56`

These are the four most commonly encountered sequences. The musical theoretical aspects of harmonic sequences are outside the scope of this paper, but a good reference for review can be found here: https://myweb.fsu.edu/nrogers/Handouts/Diatonic_Sequence_Handout.pdf

Finally, `<LENGTH>` is any natural number (though a sequence that is too long will exceed the octave restriction of 1 through 8).

Harmonic sequences can be written out in a variety of ways, depending on the chord inversions chosen. Furthermore, MusAssist at this time does not support multiple clefs (i.e. lines) simultaneously. This means that currently, harmonic sequences cannot be written with a bass line. However, the harmonies are preserved through the upper voices in keyboard-style voice leading. Though the upper-voice harmonization of a harmonic sequence need not follow the direction of the sequence's name (i.e. a descending fifths sequences could present as a series of ascending chords in the upper voices), MusAssist chooses a chord inversion and voice leading pattern such that each sequence does align with the direction of its name.

This is demonstrated in each of the following examples, all in C major. Each has length fifteen, to demonstrate that each of the four supported harmonic sequences consists of fourteen chords before it repeats (albeit in a different octave). Principles of smooth voice leading were used for all sequences.

The following figures were created with the following MusAssist expressions, respectively: `(AscFifths C4 maj quarter length:15)`, `(DescFifths C4 maj quarter length:15)`, `(Asc56 C4 maj quarter length:15)`, `(Desc56 C6 maj quarter length:15)` When loaded into MuseScore, they look like:

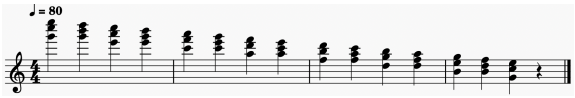

Figure 3.9: Ascending Fifths Sequence

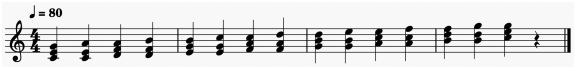

Figure 3.10: Descending Fifths Sequence

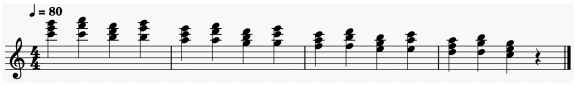

Figure 3.11: Ascending 5-6 Sequence

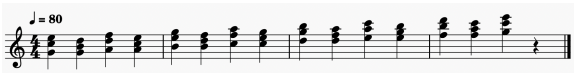

Figure 3.12: Descending 5-6 Sequence

In most music theory settings, the ascending fifths sequence would terminate after five chords (i.e. before it reaches a root position diminished triad). However, MusAssist does not stop at this chord based on the principle that users should be able to generate sequences of any length. Terminating this sequence after five chords is up to the user.

Furthermore, a minor descending fifths sequence would generally have the final appearance of scale degree seven raised, since it is leading back to the tonic. However, MusAssist does not do this based on the principle that all sequences should follow the pattern they are given, for any arbitrary length. It is up to the user to raise the final scale degree seven in a minor descending fifths sequence.

3.6 Cadences

The syntax for a cadence is as follows

`(<CADENCETYPE> <NOTENAME><ACCIDENTAL><OCTAVE> <QUALITY> <DURATION>)`

`<CADENCETYPE>` is taken from one of the following options:

`PerfAuthCadence, ImperfAuthCadence, HalfCadence, PlagalCadence, DeceptiveCadence`

`<NOTENAME>`, `<OCTAVE>`, and `<DURATION>` are taken from the same options as in Section 3.2.

`<ACCIDENTAL>` is taken from the same restricted options as in Section 5.2.1 and `<QUALITY>` is taken from the same restricted options as in Section 3.5.

Together, `<NOTENAME><ACCIDENTAL><OCTAVE> <QUALITY>` determine the local key (and the start octave) of the cadences. Recall that a key can only be major or minor, and a key cannot be in a

double sharp or flat due to the potential for triple sharps or flats to appear in the key.

The music theory of cadences is outside the scope of this paper, but good references for review can be found here: https://sites.google.com/site/musictheorycheatsheet/diatonicism/ii-functional-harmony-and-cadences and here: https://sites.google.com/site/musictheorycheatsheet/diatonicism/ii-functional-harmony-and-cadences

The five possible cadence types in MusAssist demonstrated in the following examples.

`(PerfAuthCadence Eb5 min sixteenth)`, `(ImperfAuthCadence F#5 min quarter)`, `(DeceptiveCadence B4 maj eighth)`, `(HalfCadence G#5 min whole)`, `(PlagalCadence C5 maj half)`

Again, the cadences are written in the upper voices only, in keyboard voice leading style, due to MusAssist's current lack of support for multiple clefs. Principles of smooth voice leading were applied throughout.

These MusAssist expressions, respectively, translate to the following when loaded into the MuseScore notation software. Notice the doubled root in the perfect authentic cadence in an attempt to simulate the 4-5-1 bass line, as well as to preserve the 2-1 downward step in the uppermost voice required for this cadence.

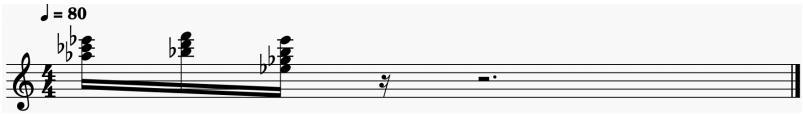

Figure 3.13: Perfect Authentic Cadence in E♭minor

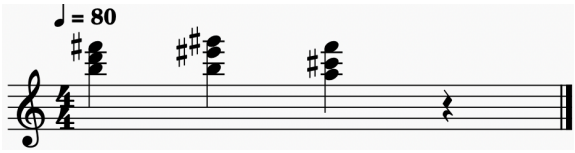

Figure 3.14: Imperfect Authentic Cadence in F# minor

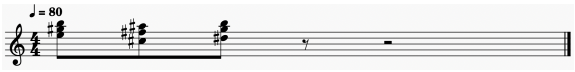

Figure 3.15: Deceptive Cadence in B major

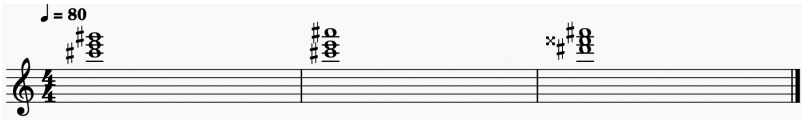

Figure 3.16: Half Cadence in G# minor

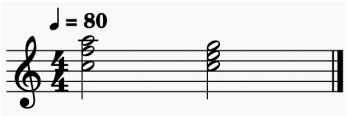

Figure 3.17: Plagal Cadence in C major

3.7 Labeled Expressions

A series of any of the above MusAssist expressions can be stored in a label (i.e. a string of the user's choice). The label must start with a letter, and can contain letters, numbers, underscores, and single quotes. This label is thus syntactic sugar for the musical expressions it refers to. When the label is referenced later in the program, the compiler will translate the expression it refers to.

For example,

```
label123_' = (D4 whole) (Ab4 quarter) (rest whole) ([Bbb5, Db5, C5] half)
label123_'
```

will translate to the following when loaded into MuseScore:

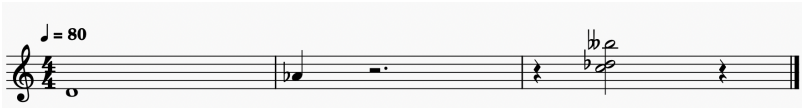

Figure 3.18: Music Translated from Labeled Expression

3.8 Key Signatures

The syntax for a key signature change is as follows

```
SET_KEY <NOTENAME><ACCIDENTAL><QUALITY>
```

where `<NOTENAME>` is taken from the same options as in Section 3.2, `<ACCIDENTAL>` is taken from the same restricted options as in Section 5.2.1 and `<QUALITY>` is taken from the same restricted options as in Section 3.5. Again, accidentals and qualities are restricted because keys must be major or minor, and cannot be built on a double sharp or flat due to the problem of such keys introducing triple sharps or flats.

A key signature also cannot have double sharps or flats, which could happen even if the key signature is not built on a double sharp or flat note. This means that the following key signatures, though they are built on seemingly valid notes, are actually invalid:

```
D#maj, E#maj, G#maj, A#maj, B#maj, Fbmaj
E#min, B#min, Cbmin, Dbmin, Fbmin, Gbmin
```

Notably, keys cannot be changed within a measure. If the users attempts to do this, a new measure will automatically get started. Also, a MusAssist key signature change must occur by

itself on its own line. Finally, if the user sets a key signature to what the current key signature already is, nothing will happen.

For example,

```
SET_KEY Dmin
SET_KEY Amaj
SET_KEY Amaj
SET_KEY Cmaj
```

will translate to the following when loaded into MuseScore:

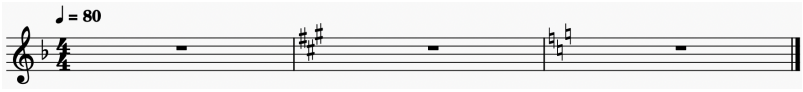

Figure 3.19: Key Signature Changes

3.9 New Measures

A new measure can be started at any point in the program with the command `NEW_MEASURE`

For example,

```
(D4 quarter)
NEW_MEASURE
(E4 eighth)
```

will translate to the following when loaded into MuseScore:

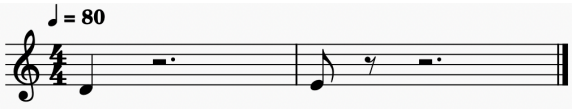

Figure 3.20: Starting a New Measure

A new measure MusAssist instruction must occur by itself on its own line.

3.10 Comments

Comments are designated with `//`. Any text after the comment indicator is ignored by the complier. For instance:

```
// this is a comment
```

MusAssist also supports multiline comments. Any text enclosed between the multiline comment symbols `//*` and `*\\` is ignored by the compiler. For instance:

```
//* This
is
a
comment *//
```

Chapter 4

Lexing and Parsing

Parsing of MusAssist concrete syntax was implemented with parser combinators using parsers from Parsec, an industrial strength parser library. Parsec's helper module Token was used for lexing. To combine parsers, Haskell's Applicative module was used, and the following custom monadic sequencing operators[1]

```
infixl 1 >>=:
left >>=: f = f <$> left

infixl 1 >>:
left >>: v = left >> return v
```

were defined to generate the results of the parse.

The following sections discuss the parsing of the concrete syntax from Chapter 3 into abstract syntax. The abstract syntax was carefully designed to best model musical structures organically.

4.1 Note Names

Recall from Section 3.2 that the set of all possible values for the concrete syntax `<NOTENAME>` is:

`F,C,G,D,E,A,B`

These are parsed, respectively, into the abstract syntax `NoteName` (a custom Haskell algebraic data type, or ADT, defined below:

```
data NoteName =
    F
    | C
    | G
    | D
    | E
```

[1] credit: Ben Wiedermann

```
    | A
    | B
```

The reason that an ADT was created for note names, rather than a simple type alias for String was not used for note names, was so that (1) correctness could be enforced for the possible note names and (2) so that custom orderings, as well as the ability to take successors and predecessors, can be defined for the note names. As will be discussed in Chapter 5, two orderings will be needed for note names, which are accomplished through deriving `Ord` and implementing a custom instance of the `Enum` typeclass. For now, simply notice that the `NoteName` ADT lists the notes in the order of sharps.

4.2 Accidentals

Recall from Section 3.2 that the set of all possible values for the concrete syntax `<ACCIDENTAL>` is:

`#, ##, b, bb`

These are parsed into the abstract syntax `Accidental` (a custom Haskell ADT) defined below. Recall that `<ACCIDENTAL>` is always an optional parameter in the concrete syntax. The mappings from concrete to abstract syntax are as one would expect semantically, and when the user leaves out the `<ACCIDENTAL>` parameter after `<NOTENAME>`, the result of the parse is `Natural`

```
data Accidental =
     DoubleFlat
     | Flat
     | Natural
     | Sharp
     | DoubleSharp
```

Like notes, accidentals were not represented in the abstract syntax with a String alias in order to (1) enforce correctness for the limited possible values and (2) enforce a custom ordering. As will be discussed in Section 5.2.1, one ordering (as well as the ability to take successors and predecessors) will be needed for accidentals, which is accomplished through deriving `Enum`.

4.3 Octaves

Recall from Section 3.2 that the set of all possible values for the concrete syntax `<ACCIDENTAL>` is:

`1,2,3,4,5,6,7,8`

These are parsed into the abstract syntax `Octave`, a Haskell type alias for Int. `Octave` is defined as follows:

```
type Octave = Int
```

An Int type alias, rather than a custom ADT, was chosen for octaves as the constraint of 1-8 is simple to enforce in a later stage, and no custom ordering is needed beyond what is already provided by Int.

4.4 Inversions

Recall from Section 5.2.1 that the set of all possible values for the concrete syntax `<INVERSION>` is:

`root, first, second, third`

They are parsed, respectively, into the abstract syntax `Inversion`, a custom Haskell ADT. `Inversion` is defined as follows:

```
data Inversion =
  Root
  | First
  | Second
  | Third
```

An ADT was better than a String alias to represent inversions in order to best enforce correctness of the data given their limited possible values.

4.5 Length

Recall from Section 3.5 that the concrete syntax `<LENGTH>` can be any natural number. This thus parses to the abstract syntax `Length`, a Haskell type alias for Int:

```
type Length = Int
```

4.6 Labels

Recall from Section 3.7 that a the concrete syntax of a MusAssist label is simply a string of the user's choice that must start with a letter, and can contain letters, numbers, underscores, and single quotes. Labels thus parse to the abstract syntax `Label`, a Haskell type alias for String:

```
type Label = String
```

4.7 Durations

Recall from Section 3.1 that the set of all possible values for the concrete syntax `<DURATION>` is:

`sixteenth, eighth, dotted_eighth, quarter, dotted_quarter, half, dotted_half, whole`

They are parsed, respectively, into the abstract syntax `Duration`, a custom Haskell ADT. `Duration` is defined as follows:

```
data Duration =
    Sixteenth
    | Eighth
    | DottedEighth
    | Quarter
```

```
    | DottedQuarter
    | DottedHalf
    | Half
    | Whole
```

Durations were not represented in the abstract syntax with a String alias in order to (1) enforce correctness for the limited possible values and (2) enforce a custom ordering. `Duration` derives `Ord`, which enforces for the ordering of durations from smallest to largest, as seen in the data deffinition. Of note, durations are not represented as floats as MusAssist does not support arbitrarily complex durations, and ordered custom data types are easier to work with than floats.

4.8 Qualities

Recall from Section 5.2.1 that the set of all possible values for the concrete syntax `<QUALITY>` is:

`maj, min, aug, dim, halfdim`

They are parsed, respectively, into the abstract syntax `Quality`, a custom Haskell ADT. `Quality` is defined as follows:

```
data Quality =
    Major
    | Minor
    | Augmented
    | Diminished
    | HalfDiminished
```

An ADT was better than a String alias to represent qualities in order to best enforce correctness of the data given their limited possible values.

4.9 Chord Types

Recall from Section 5.2.1 that the set of all possible values for the concrete syntax `<CHORDTYPE>` is:

`triad, seventh`

They are parsed, respectively, into the abstract syntax `ChordType`, a custom Haskell ADT. `ChordType` is defined as follows:

```
data ChordType =
     Triad
     | Seventh
```

An ADT was better than a String alias to represent qualities in order to best enforce correctness of the data given their limited possible values.

4.10 Cadence Types

Recall from Section 5.2.1 that the set of all possible values for the concrete syntax `<CADENCETYPE>` is:

`PerfAuthCadence, ImperfAuthCadence, HalfCadence, PlagalCadence, DeceptiveCadence`

They are parsed, respectively, into the abstract syntax `CadenceType`, a custom Haskell ADT. `CadenceType` is defined as follows:

```
data CadenceType =
  PerfAuth
  | ImperfAuth
  | Plagal
  | HalfCad
  | Deceptive
```

An ADT was better than a String alias to represent cadence types in order to best enforce correctness of the data given their limited possible values.

4.11 Harmonic Sequence Types

Recall from Section 3.5 that the set of all possible values for the concrete syntax `<HARMSEQTYPE>` is:

`AscFifths, DescFifths, Asc56, Desc56`

They are parsed, respectively, into the abstract syntax `HarmonicSequenceType`, a custom Haskell ADT. `HarmonicSequenceType` is defined as follows:

```
data HarmonicSequenceType =
  AscFifths
  | DescFifths
  | Asc56
  | Desc56
```

An ADT was better than a String alias to represent harmonic sequence types in order to best enforce correctness of the data given their limited possible values.

4.12 Tones

A tone (i.e. a "sounding," or non-rest, note) is comprised of a note name, accidental, and octave, and is represented with `Tone`, a custom Haskell ADT:

```
data Tone = Tone NoteName Accidental Octave
```

The parsers for note name, accidental, and octave are thus combined to create a parser for tones:

```
parseTone :: Parsec String () Tone
parseTone = Tone <$> parseNoteName <*> parseAccidental <*>
            (natural >>=: \octave -> fromIntegral octave)
```

where `natural` is a helper lexing function for natural numbers from Parsec's `Token` module.

4.13 Intermediate Expressions

An intermediate expression describes one of two entities: a musical template that will get expanded in Chapter 5, or a "final expression" that will not get expanded. Intermediate expressions are represented with `IntermediateExpr`, a custom Haskell ADT:

```
data IntermediateExpr =
  Note Tone Duration
  | ChordTemplate Tone Quality ChordType Inversion Duration
  | Cadence CadenceType Tone Quality Duration
  | HarmonicSequence HarmonicSequenceType Tone Quality Duration Length
  | Label Label
  | FinalExpr Expr
```

Parsers for the musical building blocks described in the previous sections of this chapter are combined, similar to in Section 4.12, to create parsers for each `IntermediateExpr`

As the ADT demonstrates, the templates to expand are notes, chord templates, cadence, harmonic sequences, and labels. In Chapter 5, they will be each expanded to a value of type `Expr`, a custom Haskell ADT describing the building blocks of all musical expressions:

```
data Expr =
  Rest Duration
  | Chord [Tone] Duration
  | LabeledExpr [Expr]
```

Any template described in `IntermediateExpr` will get expanded to the `Expr`s it contains. Another way of thinking about this is that `IntermediateExpr`s get "lowered" to `Expr`s.

An `IntermediateExpr` of type `FinalExpr` simply unwrapped to the `Expr` The concrete syntax that parses to `FinalExpr` is that for rests and custom chords.

4.14 Intermediate Instructions

An intermediate instruction describes the four possible instructions in a MusAssist program: changing the key signature, creating a new measure, writing a series of musical expressions, or assigning a label to a series of expressions to be reused later. Intermediate instructions are represented with `IntermediateInstr`, a custom Haskell ADT:

```
data IntermediateInstr =
  IRKeySignature NoteName Accidental Quality
  | IRNewMeasure
```

```
  | IRWrite [IntermediateExpr]
  | IRAssign Label [IntermediateExpr]
```

As will be discussed in Chapter 5, each `IntermediateInstr` is expanded in the intermediate representations phase to a value in the ADT `Instr`:

```
data Instr =
  KeySignature Int Int
  | NewMeasure
  | Write [Expr]
  | Assign Label [Expr]
```

Like `Expr`, `Instr`. is not included in the parse result. `IRKeySignature` is a template describing a key signature that gets expanded (or "lowered") from its description to the number of sharps and flats held in `KeySignature` Every other `IntermediateInstr` simply maps 1-1 to its corresponding `Instr`, with `IntermediateExpr` parameters expanded to `Expr` as necessary. Again, this will be explained in great detail in Chapter 5.

4.15 Final Remarks

When writing the parser, it was important to order parser alternatives correctly due to the issue of overlapping prefixes of identifiers in the concrete syntax. For instance, consider the parser for intermediate expressions (i.e. that maps to the ADT `IntermediateExpr`)

```
parseExpr :: Parsec String () IntermediateExpr
parseExpr =
  try parseLabel
  <|> parens
    (try parseChordTemplate
      <|> try parseNote
      <|> try parseCadence
      <|> parseFinalExpr
      <|> parseHarmSeq)
    <?> "Expected expression"
```

We must call `parseChordTemplate` before `parseNote`, because `parseChordTemplate` will attempt to parse the string "halfdim" (i.e. a quality) and parseNote will attempt to parse the string "half" (i.e. a duration). Since "half" is a prefix of "halfdim", this means that if we call parseNote before parseChordTemplate on the string "halfdim," the parse will succeed on the prefix "half", which will incorrectly be parsed as a duration, and then the parse will fail on "dim." Furthermore, as seen in `parseExpr`, it is necessary to use Parsec's "try" keyword on all parsers that succeed on strings with overlapping prefixes, even when they are in the right order, so that when the parse fails on the first parser alternative(s) after the overlapping prefix, no input will get consumed before trying the next one.

Chapter 5

Intermediate Representations

This chapter discusses the expansions (or "lowerings") of the musical templates described in the ADTs `IntermediateInstr` and `IntermediateExpr`

5.1 Intermediate Instructions

The `IntermediateInstr`s `IRNewMeasure, IRWrite [IntermediateExpr],` and `IRAssign Label [IntermediateExpr]` all have straightforward conversions to `Instr`.

`IRNewMeasure` simply gets translated to `NewMeasure`, `IRWrite [IntermediateExpr]` gets translated to `Write [Expr]`, and `IRAssign Label [IntermediateExpr]` gets translated to `Assign Label [Expr]` (The expansion of the `IntermediateExpr`s in the lists is discussed in Section 5.2).

The only complex expansion of an `IntermediateInstr` in the conversion of a key signature description (i.e. note name, accidental, and quality) to the number of sharps or flats in the key.

5.1.1 Key Signatures

The logic for the number of sharps or flats in a key signature is divided based on the two possible key signature qualities: major and minor.

Major

In order to determine the number of sharps or flats for a major key, the possible valid keys (i.e. keys with 0-7 sharps or flats) were listed and divided into keys with 0 or more sharps (Figure 5.5), and keys with 0 or more flats (Figure 5.7). (If an invalid key is given, an error is thrown).

Key (major)	C	G	D	A	E	B	F#	C#
Last Sharp		F#	C#	G#	D#	A#	E#	B#
Num Sharps	0	1	2	3	4	5	6	7

Figure 5.1: Major Keys with 0-7 Sharps

Key (major)	C	F	Bb	Eb	Ab	Db	Gb	Cb
Last Flat		Bb	Eb	Ab	Db	Gb	Cb	Fb
Num Flats	0	1	2	3	4	5	6	7

Figure 5.2: Major Keys with 0-7 Flats

Let a *flat key signature* have at least one flat, and let a *sharp key signature* have at least one sharp.

In Figure 5.5, we see that the number of sharps in a sharp major key signature corresponds to the *index* of the note name of last sharp appearing in the key signature, in the order of sharps.

Similarly, in Figure 5.7, we see that the number of flats in a flat major key signature corresponds to $7-(i-1)$, where i is the the index of the key signature's note name in the order of sharps. The "seven-minus" calculation comes from the fact that the order of flats (BEADGCF) is simply the reverse of the order of sharps (FCGDAEB), and there are seven possible sharps or flats because there are seven possible note names.

In order to support this logic, a list of `NoteNames` in the order of sharps called `globalOrderOfSharps` was created.

The question now becomes how to determine the last sharp in a sharp major key signature. The first step is to determine which key signatures are flat or sharp, given only the key signature's description of note name and accidental. Examining Figures 5.5 and 5.7 again, we notice the following patterns:

1. Any key signature with a flat in its name (i.e. A♭ major) is a flat key signature. However, the only valid key signatures with flats in their names (i.e. those that do not have double sharps or flats and thus appear in Figures 5.5 and 5.7) are those whose note name n satisfies $n \geq$ C in the order of sharps.
2. Any key signature with a sharp in its name (i.e. C♯ major) is a sharp key signature. However, the only valid key signatures with sharps in their names (i.e. those that do not have double sharps or flats and thus appear in Figures 5.5 and 5.7) are those whose note name n satisfies $n \leq$ C in the order of sharps.
3. C major and F major are special cases. C major has zero sharps or flats, and F major has one flat.
4. Any other key signature (which must have a natural in its name, i.e. G major) is a sharp key signature.

If we are in a sharp key signature, the rule from music theory is that the last sharp is half

a step down from the tonic (i.e. key) note name. However, to determine the name of the note below a given note, we must define a second ordering of the ADT `NoteName` Recall that `NoteName`, but uses the order of sharps for this. Thus, to order note names in the order they occur in a scale (i.e. CDEFGAB), we create a custom instance of the `ENUM` typeclass for `NoteName` (see: here). Notably, making `NoteName` an instance of `ENUM` does not define a second set of comparison definitions that defy `NoteName`'s derivation of `Ord` (i.e. F is still less than C), but it gives us access to the functions `succ` and `pred` that will return the previous or next `NoteName` based on the scale ordering CDEFGAB. In `NoteName`'s custom instance of `Enum`, `succ` and `pred` are both defined circularly. Thus, `succ B` is C, and `pred C` is B.

Thus, if we are in a sharp key signature, to determine the name of the last sharp in the key, we can simply call `pred` on the tonic (i.e. key) note name. To get the index of this resulting `NoteName` in the order of sharps, we simply look up its index in the list `globalOrderOfSharps` (accounting of course for the zero-indexing of lists).

To summarize, the logic of determining the number of sharps or flats in a major key is:

1. Any key signature with a flat in its name, and whose note name n satisfies $n \geq$ C in the order of sharps, is a valid flat key signature. We look up the index of n in `globalOrderOfSharps` to get i, and the key thus has $7 - (i - 1)$ flats.
2. Any key signature with a sharp in its name, and whose note name n satisfies $n \leq$ C in the order of sharps, is a valid sharp key signature. `pred n` gives us n', the note name of the last sharp in the key signature. The index of n' in `globalOrderOfSharps` tells us how many sharps the key signature has (accounting for the zero-indexing of `globalOrderOfSharps`)
3. Any key signature with a natural in its name is a key signature. C major (zero sharps or flats) and F major (one flat) are special cases. All other such key signatures are sharp key signatures, and the number of sharps is determined in the same way as just discussed.

Minor

Rather than go through the logic of determining the number of sharps or flats in a minor key signature, we convert each minor key signature to its relative major, and apply the logic from the previous section.

Consider the following mapping of valid minor key signatures (i.e. those without double sharps or flats) to their relative major key signatures. Recall that to get the relative major of a minor key, we go up a minor third in from the tonic of the minor key.

Key (minor)	A	E	B	F#	C#	G#	D#	A#
Relative major	C	D	G	A	E	B	F#	C#
Num Sharps	0	1	2	3	4	5	6	7

Figure 5.3: Relative Major Name Accidentals of Minor Keys with Sharp in Name

Key (minor)	A	D	G	C	F	Bb	Eb	Ab
Relative major	C	F	Bb	Eb	Ab	Db	Gb	Cb
Num Flats	0	1	2	3	4	5	6	7

Figure 5.4: Relative Major Name Accidentals of Minor Keys with Sharp in Name

This allows us to notice the following patterns. Each of the following diagrams shows the accidental in the name of the relative major key signature of each of the minor keys in the list. Notice that the minor keys are ordered in the order of sharps.

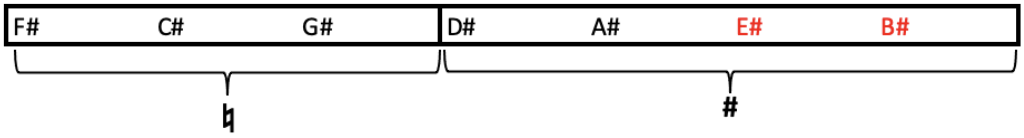

Figure 5.5: Relative Major Accidentals for Minor Keys with Sharp in Name

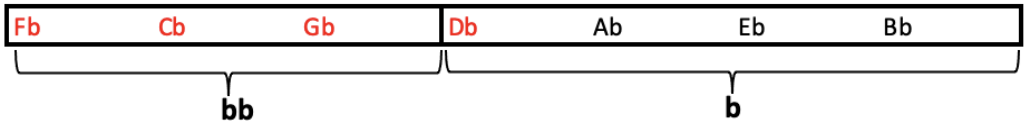

Figure 5.6: Relative Major Accidentals for Minor Keys with Flat in Name

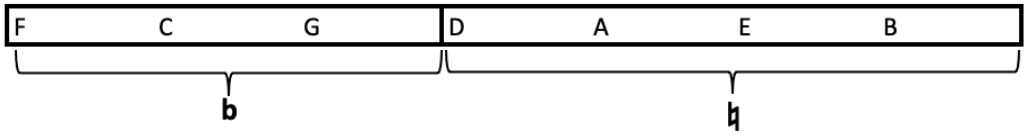

Figure 5.7: Relative Major Accidentals for Minor Keys with Natural in Name

Minor keys in red are invalid (i.e. contain double sharps or flats). However, these will be identified as invalid later, when they have subsequently been processed as major keys after being converted to the relative major. Thus, we can simply determine the accidental of the relative major key by ignoring invalid keys for now. Given a minor key with note name n and accidental a, the accidental of its relative major can be determined by D as a "pivot."

1. If $a = \flat$, then if $n \geq$ D in the order of sharps, the accidental of its relative major is $\sharp$. Otherwise, it is $\natural$.
2. If $a = \flat$, then if $n \geq$ D in the order of sharps, the accidental of its relative major is $\flat$. Otherwise, it is $\flat\flat$.
3. If $a = \flat$, then if $n \geq$ D in the order of sharps, the accidental of its relative major is $\natural$. Otherwise, it is $\flat$.

Since we go up a minor third from n, the tonic of the minor key, to get the note name of the relative major, we simply call `succ` twice on n to get the note name of the relative major.

5.2 Intermediate Expressions

The `IntermediateExpr`s `FinalExpr Expr`, `Note Tone Duration`, `Label Label` have relatively simple expansions.

For `FinalExpr`s, the `Expr` parameter is simply returned (think of this as "unwrapping" `FinalExpr Expr`).

A `Note` will get expanded to a single-tone `Chord`

In terms of `Label`s, when the `IntermediateInstr IRAssign Label [IntermediateExpr]` gets translated to the the `IntermediateInstr Assign Label [Expr]`, the list of expanded `Expr`s is stored in a symbol table mapped to the label name. Then, when expanding `Label`, the label name is simply queried in the symbol table, and the list of expanded expressions for that label is returned to replace the label reference. This desugaring step must occur in the intermediate representations stage of the compiler rather than the code generation stage for reasons that will be discussed in Chapter 6.

All the other templates described in `IntermediateExpr` have more complex expansions. On a high level, a `ChordTemplate` will get expanded to a multi-note `Chord`, and `Cadence` and a `HarmonicSequence` each will get expanded to a list of multi-note `Chord`s.

5.2.1 Chord Templates

Recall that a chord template contains information about its root note, quality, type, inversion, and duration. It will get expanded to the list of notes (i.e. tones) that it describes.

Expansion of chord templates begins with validity checks: that (1) the chord root is not a double sharp or flat (MusAssist does not support this), and (2) that the chord is not a half diminished triad (impossible according to music theory). After this, each of the notes of the chord is determined.

To do this, a helper function called `generateToneWithinScale` was created. This function, given the tonic tone and quality (either major or minor), will generate the tone within that diatonic scale given the desired interval from the tonic (must be between 1 and 6, i.e. within one octave of the tonic).

In `generateToneWithinScale`, determining the note name of the desired tone within the scale based on interval is straightforward: simply apply `succ` to the tonic note n times, where n is the interval from the tonic. Recall that `succ` is defined in the custom Enum instance of the NoteName ADT and is based on the order of notes in a scale, not the order of sharps like the `Ord` derivation is.

Notably, `generateToneWithinScale` also takes in the parameters `specialOctaveCases` and `octFunc` The default octave of the tone is the same as the tonic, but `generateToneWithinScale` will apply the custom function (`succ` or `pred` in the implementation) to the default octave if the computed note name is in `specialOctaveCases`. This allows the octave to be customized as needed. The logic of `specialOctaveCases` and `octFunc` is implemented in the chord template expansion

function, and will be revisited when we return to this function.

The most difficult part of `generateToneWithinScale` is determining the accidental of the tone. To work out the logic behind this, the following figures consider all single-accidental key signature names (even invalid ones that contain double sharps or flats, in order to establish the pattern). The diagrams group the key signature names under the accidental of the note that is the specified interval from the tonic. For instance, in both A♭ major and minor (since all major or minor scales have a major second), the major second interval from the tonic (A♭) is B♭. The accidental of B♭ is ♭, so in Figure 5.8, A♭ falls under the ♭ column.

	Major Seconds		
♭	♮	♯	𝄪
F♭	E♭	E	E♯
C♭	B♭	B	B♯
G♭	F	F♯	
D♭	C	C♯	
A♭	G	G♯	
	D	D♯	
	A	A♯	

Figure 5.8: Accidental of Major Second from Tonic per Key

This allows us to ascertain that in any key, the note that is a major second above the tonic has the same accidental as the tonic, except for keys that have the note name E and B. (Namely, these keys are E and B major or minor, E♭and B♭major or minor, and E♯and B ♯major or minor). In these cases, the accidental is "lifted" once (i.e. ♭becomes ♮, ♮becomes ♯, and ♯becomes 𝄪). This is where the ADT `Accidental`'s derivation of `Enum` comes in. In order to lift or lower an accidental, we can simply apply `succ` or `pred`. Thus, if the note name of the scale key is E or B, we apply `succ` to the accidental in the key name (i.e. the accidental of the tonic) to get the accidental of the note a major second above the tonic.

We can proceed in a similar fashion for all other imperfect intervals. In the case of thirds, it turns out that the pattern emerges for minor thirds, rather than major thirds.

	Minor Thirds		
𝄫	♭	♮	♯
F♭	D♭	D	D♯
C♭	A♭	A	A♯
G♭	E♭	E	E♯
	B♭	B	B♯
	F	F♯	
	C	C♯	
	G	G♯	

Figure 5.9: Accidental of Minor Third from Tonic per Key

We see that for any key, the note that is a minor third above the tonic has the same accidental

as the tonic, except for keys that have the note name F, C, or G. In these cases, the accidental is "lowered" once (i.e. ♭becomes 𝄫, ♮becomes ♭, and ♯becomes ♮). Just like with major seconds, now, if the note name of the scale key is F, C, or G, we apply `pred` to the accidental in the key name (i.e. the accidental of the tonic) to get the accidental of the note a minor third above the tonic.

Subsequently, the case of sixths, it turns out that the pattern emerges for major sixths, rather than minor thirds.

	Major Sixths		
♭	♮	♯	𝄪
F♭	A♭	A	A♯
C♭	E♭	E	E♯
G♭	B♭	B	B♯
D♭	F	F♯	
	C	C♯	
	G	G♯	
	D	D♯	

Figure 5.10: Accidental of Major Sixth from Tonic per Key

We see that for any key, the note that is a major sixth above the tonic has the same accidental as the tonic, except for keys that have the note name A, E, or B. In these cases, the accidental is "lifted" once, like with major seconds. Thus, if the note name of the scale key is A, E, or B, we apply `succ` to the accidental in the key name (i.e. the accidental of the tonic) to get the accidental of the note a major sixth above the tonic.

Finally, the case of sevenths, it turns out that the pattern emerges for minor sevenths, rather than major sevenths.

	Minor Sevenths		
𝄫	♭	♮	♯
F♭	G♭	G	G♯
C♭	D♭	D	D♯
	A♭	A	A♯
	E♭	E	E♯
	B♭	B	B♯
	F	F♯	
	C	C♯	

Figure 5.11: Accidental of Minor Seventh from Tonic per Key

We see that for any key, the note that is a major sixth above the tonic has the same accidental as the tonic, except for keys that have the note name F or C. In these cases, the accidental is "lowered" once, like with minor thirds. Thus, if the note name of the scale key is F or C, we apply `pred` to the accidental in the key name (i.e. the accidental of the tonic) to get the accidental of the note a minor seventh above the tonic.

We finally consider the perfect intervals: fourths and fifths. By thinking about these intervals for a moment, it becomes clear that perfect intervals always have the same accidental as their tonic, except in those rare cases when we get a tritone. There is only one case that this occurs for each of these intervals: for perfect fourths, keys with F in the key name, and for perfect fifths, keys with B in the note name. The logic for this can be seen by visualizing a piano. B is a tritone above F because B is only a half step (rather than a whole step) below C, and F is a tritone above B because F is only a half step (rather than a whole step) above E.

All of these conclusions allow us to summarize our findings in the following map (defined here):

```
globalStepsFromTonicToAccInfoMap :: Map Int ([MusAST.NoteName],
                                    MusAST.Accidental -> MusAST.Accidental,
                                    Maybe MusAST.Quality)
globalStepsFromTonicToAccInfoMap = Map.fromList
    [(0, ([], (succ . pred), Nothing)),                                 -- root
     (1, (drop 5 globalOrderOfSharps, succ, Nothing)),                  -- seconds
     (2, (take 3 globalOrderOfSharps, pred, Just MusAST.Minor)),        -- thirds
     (3, ([MusAST.F], pred, Nothing)),                                  -- fourths
     (4, ([MusAST.B], succ, Nothing)),                                  -- fifths
     (5, (drop 4 globalOrderOfSharps, succ, Just MusAST.Major)),        -- sixths
     (6, (take 2 globalOrderOfSharps, pred, Just MusAST.Minor))]        -- sevenths
```

`globalStepsFromTonicToAccInfoMap` maps the interval from tonic (beginning with 0, or the tonic itself) to information about the note at that interval from tonic. In the tuple on the right side of the map, the first element is the list of "special case" notes from that interval from the tonic (i.e. notes that do not have the same accidental as the tonic). The second element is the function to apply to the tonic accidental in order to get the correct accidental for these special case notes. The third and final element is the key quality that the accidental pattern is valid for (i.e. major, in the case of sixths, or minor, in the case of thirds and sevenths). Notice that although, based on the pattern discovered in Figure 5.8, we would think initially that seconds should only be valid for major, this is not the case because both major and minor keys have their supertonic a major second above the tonic. Thus, the valid key quality is `Nothing`, since this does not apply here. Similarly, key quality does not affect perfect intervals, nor the tonic itself.

We can thus use `globalStepsFromTonicToAccInfoMap` in `generateToneWithinScale` by querying based on the interval from the tonic. We then adjust the accidental by applying the queried accidental function if the note name is in the special cases list, and then we further adjust based on the key quality the pattern is valid for. For instance, if we are in a major scale and the interval value is 2 (meaning we want a major third), we need to apply `succ` to the accidental from `globalStepsFromTonicToAccInfoMap` (note that we must apply this alteration AFTER we have already applied the queried accidental function if needed). Similarly, if we are in a minor scale and the interval value is 5 (meaning we want a minor sixth), we need to apply `pred` to the accidental from `globalStepsFromTonicToAccInfoMap` once the queried accidental function has been applied as needed for special case notes.

Thus, `generateToneWithinScale` returns the tone within a diatonic scale a specified interval from the tonic, with the option to adjust the octave.

We now return to the function for expanding chord templates. We simply proceed linearly to generate each note in the chord using `generateToneWithinScale` based on the interval from the

tonic. Thirds have an interval of 2 from the tonic, fifths have an interval of 4 from the tonic, and sevenths have an interval of 6 from the tonic.

The scale is set to major for major and augmented chords, and to minor for all other chord qualities (minor, half diminished, diminished). This means that for augmented chords, the chordal fifth will need to be raised half a step (i.e. we need to apply `succ` to the accidental returned in `generateToneWithinScale`), and the chordal seventh will need to be lowered half a step. For diminished chords, both the chordal fifth and seventh will need to be lowered half a step, and for half diminished chords, the chordal fifth will need to get lowered half a step.

The special octave cases depend on the interval. For instance, for fifths, if the chordal root is F, G, A, or B, the chordal fifth's octave number is one above the root, since octave numbers change on C. We can use the convenient function `enumFromTo` from our custom instance of the `Enum` class for the ADT `NoteName`, and we get that the special octave cases for fifths are `(enumFromTo MusASTF` MusAST.B)., and the octave function to get applied in `generateToneWithinScale` for these cases is `succ`. The same logic applies to all other intervals, by simply working out which chordal roots have the specified interval above them in a different octave. This allows us to make all chords initially in root position. The octave function for `generateToneWithinScale` is always `succ` in chord template expansion, but as we will see later in the expansion of other templates, this is not always the case.

The final consideration to handle in chord template expansions are inversions. If the chord type is a triad, the chord template expansion stops after generating the notes for the triad. Otherwise, it continues on to generate the chordal seventh. In either case, inversions are handled similarly. Recall that the chord (whether triad or seventh) starts out in root position. Let n be the inversion value (0 for root, 1 for first inversion, 2 for second, and 3 or third). By incrementing the octave of the first n tones of the chord in root position, we get the correct inversion.

At this point, we have all the information we need for chord template expansion. The computed and adjusted note name, accidental, and octave of for each tone in the chord are zipped together to create a list of tones, and this is passed into the data constructor `Chord` from the ADT `Expr`. The given duration for the chord template is passed in as the second parameter for `Chord`, and we are done.

5.3 Harmonic Sequences

5.4 Cadences

Chapter 6

Code Generation

Chapter 7

Sample Programs

Chapter 8

Future Work and Conclusion

8.1 Future Work

Ideally, in the future MusAssist would support more complex musical states and elements including custom time signature and mid-composition time signature changes (similar to the behavior currently implemented for key signatures), clef changes within a part, multiple-clef parts (i.e. piano), custom parts (i.e. instruments), and multiple parts. Custom and changeable time signature would allow for the users to experiment with metric modulation, something that is currently impossible with the fixed common time setup. Clef changes within a part, both manual and automatic when a note extends too many ledger lines beyond a clef, would allow the score to be more nicely formatted and readable for the user. Support for two-clef piano would allow the MusAssist compiler to successfully modify how it generates cadences and harmonic sequences to include the essential baseline, in addition to the harmonization already implemented. The latter two goals (custom parts and multiple parts) are somewhat outside MusAssist's goal as a music compositional aid, as this extends beyond the realm of music theory. However, users may enjoy this increased flexibility when composing.

8.2 Conclusion

MusAssist is an external DSL whose Haskell-based compiler translates it to MusicXML that can be loaded into a major music notation software for further editing. Its syntax is simple and models the flow of thought a composer would have when writing music by hand. MusAssist fills a niche in the realm of musical DSLs by serving as a music compositional aid that is not intended to allow to the user to write a fully expressive musical piece, but rather to more easily create musical expressions that would be tedious to write by hand. The additional features described in the previous section would make the language align even more robustly with this goal. MuseScore can be further valuable as an education tool to music theory students, allowing them to visualize musical structures from the definitions that they describe in MusAssist.

Clearly, DSLs are a powerful mechanism to push the boundaries of computational creativity in

the field of music. Unfortunately, DSLs for music have not been studied extensively and remain an extremely comparatively small area of research, though scholars such as Wang continue to lead research in institutions such as Stanford's CCRMA (Center for Computer Research in Music and Acoustics). By continuing to examine the creative expressive power of DSLs in music, we can continue to increase our understanding of the creative capabilities and extent of customization possible for a programming language.

Bibliography

A. Bertram. What is declarative programming?, Oct 2021. URL https://www.techtarget.com/searchitoperations/definition/declarative-programming.

A. R. D. Bois and R. Ribeiro. Hmusic: A domain specific language for music programming and live coding: Semantic scholar, 2019. URL https://www.nime.org/proceedings/2019/nime2019_paper074.pdf.

J. S. Cuadrado, J. L. Izquierdo, and J. G. Molina. Comparison between internal and external dsls via rubytl and gra2mol. *Computational Linguistics*, page 816–838, Nov 2012. doi: 10.4018/978-1-4666-6042-7.ch040.

DeepSource. Synchronous programming. URL https://deepsource.io/glossary/synchronous-programming/.

M. Fowler and R. Parsons. *Domain-specific languages*. Addison-Wesley, 2011.

A. S. Gillis and S. Lewis. What is object-oriented programming (oop)?, Jul 2021. URL https://searchapparchitecture.techtarget.com/definition/object-oriented-programming-OOP.

M. Good. *MusicXML: An Internet-Friendly Format for Sheet Music*. Dec 2001. URL https://michaelgood.info/publications/music/musicxml-an-internet-friendly-format-for-sheet-music/.

M. Good. Musicxml: Introduction, Apr 2013. URL https://www.musicxml.com/publications/makemusic-recordare/notation-and-analysis/introduction/.

D. Janin. A robust algebraic framework for high-level music writing, 2016. URL https://hal.archives-ouvertes.fr/hal-01246584v2/document.

A. Joury. Why developers are falling in love with functional programming, Aug 2020. URL https://towardsdatascience.com/why-developers-are-falling-in-love-with-functional-programming-13514df4048e.

J. C. Martinez. Extending music notation as a programming language for interactive music. *ACM International Conference on Interactive Media Experiences*, Jun 2021. doi: 10.1145/3452918.3458807.

T. Matsuura and K. Jo. Mimium: A self-extensible programming language for sound and music. *Proceedings of the 9th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design*, 2021. doi: 10.1145/3471872.3472969.

H.-W. Nienhuys and J. Nieuwenhuizen. *LilyPond, A System for Automated Music Engraving*. May 2003. URL https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.6160&rep=rep1&type=pdf.

H. Nishino. Developing a new computer music programming language in the 'research through design' context. *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity - SPLASH '12*, 2012. doi: 10.1145/2384716.2384736.

H. Nishino, N. Osaka, and R. Nakatsu. Lc : A strongly-timed prototype-based programming language for computer music, 2013. URL https://quod.lib.umich.edu/i/icmc/bbp2372.2013.017/1.

H. Nishino, N. Osaka, and R. Nakatsu. Lc: A new computer music programming language with three core features, 2014. URL https://quod.lib.umich.edu/i/icmc/bbp2372.2014.237/1.

B. Petit and M. Serrano. Interactive music and synchronous reactive programming. *The Art, Science, and Engineering of Programming*, 5(1), 2020. doi: 10.22152/programming-journal.org/2021/5/2.

M. Simić, Z. Bal, I. Dejanovic, and R. Vaderna. Pytabs: A DSL for simplified music notation. *ResearchGate*, 2015. URL https://www.researchgate.net/publication/312607043_PyTabs_A_DSL_for_simplified_music_notation.

C. Sulyok, C. Harte, and Z. Bodó. On the impact of domain-specific knowledge in evolutionary music composition. *Proceedings of the Genetic and Evolutionary Computation Conference*, 2019. doi: 10.1145/3321707.3321710.

G. Wang. The chuck audio programming language: "a strongly-timed and on-the-fly environ/mentality", 2008. URL https://www.cs.princeton.edu/~gewang/thesis.html.