FORMAL SEMANTICS FOR MUSIC NOTATION CONTROL FLOW

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF CARNEGIE MELLON UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Zeyu Jin

July 2013

Abstract

Music notation includes a specification of control flow, which governs the order in which the score is read using constructs such as repeats and endings. Music theory provides only an informal description of control flow notation and its interpretation, but interactive music systems need unambiguous models of the relationships between the static score and its performance. In this work, a framework is introduced to describe music control flow semantics using theories of formal languages and compilers. A formalization of control flow answers several critical questions: Are the control flow indications in a score valid? What do the control flow indications mean? What is the mapping from performance location to static score location? The framework can be used to describe extended control flow notation beyond conventional practice, especially nested repeats and arrangement. With the introduction of SDL, the Score Description Language, and DCL, the Dynamic Control Language, the framework can be extended to describe scores notated with word instructions and real-time controls. To demonstrate the correctness and effectiveness of this framework, a score compiler and a score model manager are implemented and evaluated using case-based tests. A software, Live Score Display, is built upon this framework and is offered as a component for interactive music display.

Acknowledgements

I would like to express my deep gratitude to Professor Dannenberg, my research supervisor, for his patient guidance, enthusiastic encouragement and useful critiques of this research work. I would also like to thank Professor Stern and Professor Randall, for their advice and assistance in during the planning and development of this research work. My grateful thanks are also extended to Mr. Xia, Mr. Øland and Mr. Liang, for their support and suggestions, and to Mr. Tang, Mr. Cheng, Miss. Peng, Mr. Liu and Mr. Ozbay for their help in system evaluation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Music notation has been evolving for centuries, creating a symbolic system to convey music information. Early music notation contained only lines and notes, which are sufficient for communicating pitches and durations. It was later that bar lines and time signatures emerged, grouping music into measures and introducing the idea of beats.[1] The notation for music control flow, like repeats and codas, came even later. Control flow helps to identify repeating structures of music and eliminates duplication in the printed score. In the Classical period, control flow notation is closely tied to music forms such as binary, ternary and sonata and is more of a musical architecture than a means of saving space.[2] Conventional practice for control flow notation is well established. The literature [6, 20] has formalized the notation in all kinds of ways and there is little conflict among definitions. However, traditional music theory has not explored the possibilities of expanded or enriched representations for control flow, and there is a gap between often-simplified theoretical ideals and actual practice, especially in modern works. In practice, we find nested repeats, exceptions and special cases indicated by textual annotations, multiple endings,

[1]Far beyond formalizing the notion of beats, music notation led to the "discovery" of time as an independent dimension that did not depend upon physical actions. In particular the musical rest is the first direct representation of "nothingness" existing over time, or of time itself. Composers developed this concept centuries before the scientific revolution, Kepler, Newton, graphs with a time axes, etc. [3]

[2]For example, "In practically all the sonatas of the earlier period the exposition is repeated, as is indicated by the repeat-sign at its end, which is also helpful for the reader in finding the end of the exposition ..." [2]

and symbols for rearrangement.

We encountered this gap between theory and practice in the implementation of music notation display software. We needed a formal (computable) way to relate notation to its performance, and we found conventional notions too limiting to express what we found in actual printed scores. To address this problem, we developed new theoretical foundations based upon models of formal language and compilation, and we applied these developments to the implementation of a flexible music display system.

Music control flow is the reading order of measures affected by control symbols including the time signature, measures, repeats, endings, DC/DS signs, section signs, etc. It can also be viewed formally as a function f that maps the performed beat k to a location of a score, $<m,b>$, a measure and beat pair. $f(k)$ describes the reading order of the score. In principle, we can rewrite the score in the order $f(1)$, $f(2)$, ... to create an equivalent score with no control flow (other than reading sequentially). We call this the "flattened score" or "performance score". Audio recordings and MIDI sequences are both in the order of the flattened representation of the corresponding score.

Existing music theory devotes little attention to control flow, and in fact, there does not seem to be even a standard term for the concept of control flow. Conventional practice focuses on meaning of individual control flow symbols; the most widely accepted way is to use words and visuals to illustrate the reading order. For example, Read uses arrows to mark the true reading order (see Figure 1.1) [20]. This approach defines both the syntax and meaning.

In this work, we focus on a system that formalizes the syntax and semantics of control flow notation. The goal is to form a definition that meets the two criteria: (1) A well-defined syntax provides productions for interpreting all valid scores; (2) A well-defined semantics ensures that all valid scores have one and only one interpretation, namely there is a unique mapping from beat to score position. Under these criteria, however, the conventional visual definition is incomplete; there are cases where we are not sure which rule to use. One such case is nested repeats where there are two ways of grouping the right repeat with the left repeat. In common practice music, we often restrict the number of levels of repeats to be

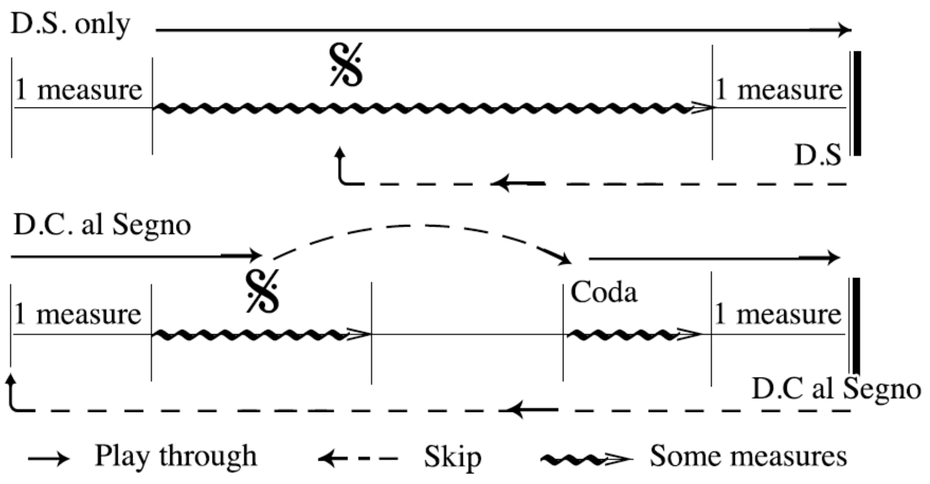


Figure 1.1: Control flow definition in Read's book

1, which excludes the nested repeat problem. In more modern and flexible music practice, there are nested structures, text annotations [9] and repetitions conditioned on actual time [23]. There lacks an unambiguous way to formalize the syntax and the meaning of such notation. While nested repeats are relatively simple to formalize, we argue that the general problem of formalizing control flow is non-obvious, interesting, and useful.

In this thesis work, I present a new framework for formalizing control flow notation based on formal grammar and compiler theories. One core feature of this framework is that it can be easily designed by humans and applied automatically by computer. As an example, we show a formal definition of control flow notation that unifies conventional music notation and some of the most frequent notations used in modern music practice. We evaluate the ambiguity and completeness of this definition within our framework. We then show the implementation of this method and its application and finally the evaluation of the framework and the application in terms of their correctness, effectiveness and performance.

Chapter 2

Background and Motivation

Similar to the concept of control flow used in programming language, music control flow defines the reading order of a score and is used to both simplify and clarify the score structure by merging repetitive patterns into loops. Also similar to programming languages, music control flow has implicitly included both branching and looping expressions [**?**]. This statement will be further evaluated in Section 4.1 and it also motivates this work to explore music notation and extend it as if it is a program language. Before going into details of the formal semantics behind the music practice, this chapter reviews music symbols and their general usage that are frequently seen in real music practice. It also draws upon the difference between tradition music notation and the more flexible and informal practices used in popular music such as Jazz to show the need of finding a practical and well-defined framework for people to formalize and explain existing and new music notation. In the last section, we review some of the existing digital music display and representation approaches that relate to and motivate the design of Live Score Display, the real-time score display system built over the formal definition of music control flow notation.

2.1 Common Music Practice

The so-called "common practice" refers to the traditional and widely-used system of music notation, consistent with most popular music education books and manuals[6, 20, 23]. The main elements include repeat, endings, and a series of Da Capo / Dal Segno notation ("DC/DS family" for abbreviation). The following is a list of symbols used in these notation patterns:

- Left repeat and right repeat: ||: and :|| (double-sided repeat :||: can be seen as one right repeat followed by a left repeat);
- First and second ending: $-1\neg$ and $-2\neg$. Sometimes, there might be a third or a fourth ending;
- DC/DS family and related symbols, including *D.C.*, *D.S.*, *D.C. al coda*, *D.S. al coda*, *D.C. al fine*, *D.S. al fine*, *Segno*, to *coda*, *coda*, and *fine*;
- Measure repeat signs.

To define the meaning and usage of these symbols, most books use visual definitions like the one in Read's book [20], which is based on redirection, and the one shown in Figure 2.1, which is based on the reading order. These definitions are well defined for the reading order given a score without erratic or ambiguous notation, but the definitions alone are not precise or complete to show what's correct and non-ambiguous for all scores. To do that, we often make the following assumptions:

- No control flow symbols are allowed within the blank staff in Rule 1, 2 and 3.
- There can only be at most one type of DC/DS symbols in the entire music.
- Left and right repeats can be used within the blank staff in Rules 4-9.

Although these assumptions are not as formal as those visual definitions and it is possible for us to find ambiguous counter examples, they do a good job excluding a large number of cases, especially the nested repeats, from the set of "correct music notation" in common

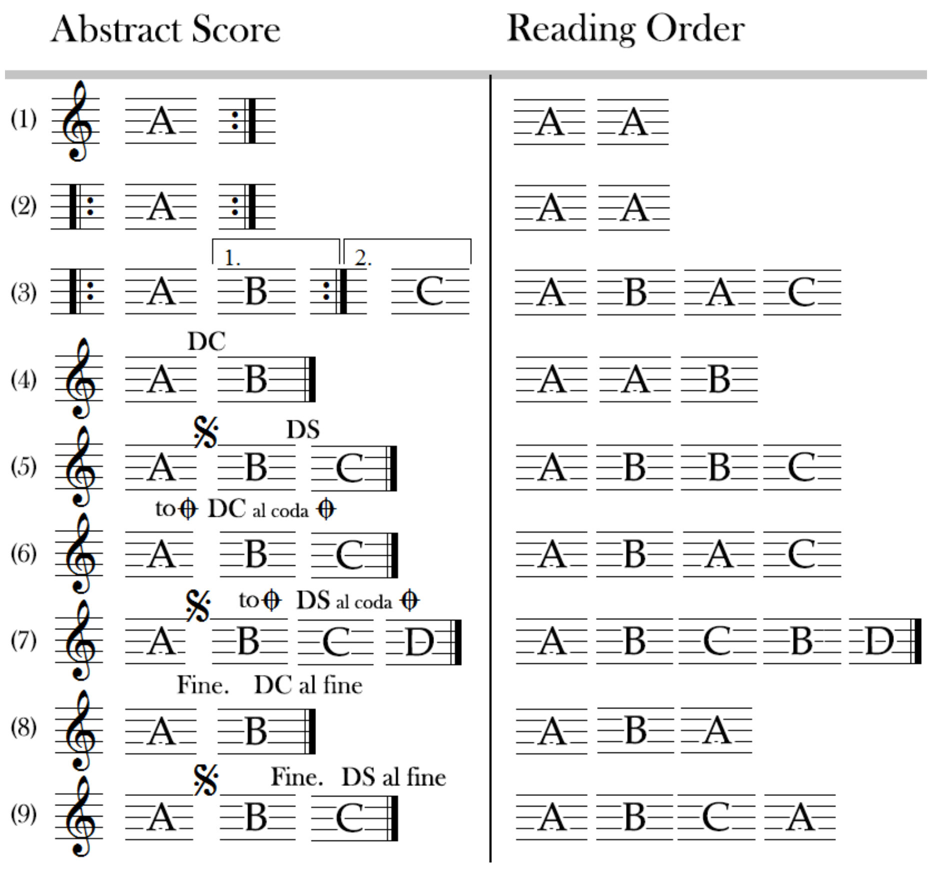


Figure 2.1: Control flow notation and the corresponding reading order in common music practice

music practice. Only when there are other patterns of notation, such as the nested repeats, that are considered with reasonably important in practice, should we be concerned about these common practice definitions. In fact, there are a number of such cases that are found in the more recent music practice and accepted by contemporary musicians.

2.2 Extended Music Practice

In modern music practice, the control flow notation is much more flexible; there are many examples undefined in the conventional notation framework. Figure 2.2 shows some of the

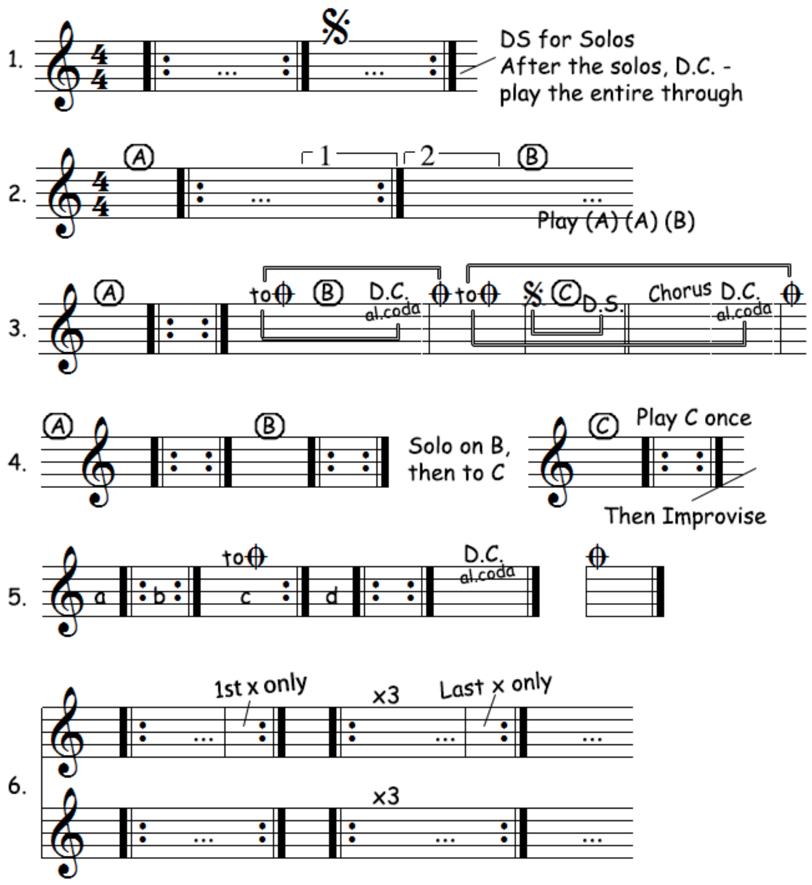


Figure 2.2: Control flow notation in modern music practice

many unconventional examples found in *The Real Book* [9]. The six examples[1] contain four typical notations in modern music practice:

Nested repeats (example 1 and 5): it is assumed in the book that repeat signs work like nested brackets; the right repeat sign is associated with the left repeat sign at the same level instead of the closest.

Multiple and nested DS/DC-coda notation: In Example 3, the notation is complicated. The relation among D.C. al coda, to coda and coda signs are connected by lines.

[1]From 1 to 6: "Follow Your Heart" by J. Mclaughlin pp.154–155; "Forest Flower" by Charles Lloyd p.158; "Good Evening Mr. & Mrs. America" by John Guerin p.176; "Group your own" by Keith Jarrett p.182; Little Niles by Raudy Weston p.267; "Mallet Man" by Gordon Boek p.282

Re-arranged sections (Example 2): re-order the section marks to create a different reading order of the score.

Word annotation (Example 1, 4 and 6): omitting or playing some measures conditioned on different rounds of repetition.

One could dismiss these particular scores as anomalies, but we find that in music, the exception proves the rule. Existing commercial software interprets Example 5 inconsistently. In Finale 2012 and Sibelius 7, the default reading order is a-b-b-c-b-b-c-d[2], but in MuseScore[3], the order is a-b-b-c-a-b-c-d. The second right repeat connects to the beginning of the music, capturing the nested pattern, but later, section b is repeated only once, which seems to be a flaw. The correct order is a-b-b-c-a-b-b-c-d, which differs from both software interpretations. Later, we will discuss the practice of "arrangements," where score sections are performed according to directions that are external to the score itself. This raises additional problems of notation and semantics. Clearly, we need a more sophisticated approach that offers syntax and meaning that can be shared by both humans and computers. In Chapter 3, we show such an approach based on formal languages and attribute grammars. In Chapter 4, we show an even more flexible method based an intermediate notation called Score Description Language that supports extensions based on textual annotations. In Chapter 7.2, we show some of the examples of how the extended music practice is mapped onto our approach.

2.3 Score Display and Representation

The score representation and interpretation is essential to score writing, printing, auto-accompaniment and conducting systems. Most works in this field focus mainly on storing and visualizing the score so as to accommodate both traditional and modern music practice, and, as a minor concern, provide room for implementing correct and user-friendly score play-back systems. Early studies such as MuseData [11] and Common Music Notation

[2]One can notate nested repeat by manually specifying which measure the right repeat goes to.
[3]MuseScore music notation software: http://musescore.org/

[22] mainly concern the encoding of conventional measure-based scores in their original notational presentation. Existing software such as Music Reader[4] and hardware such as Music Score Display Device [15] concern creating a score browsing interface for the purpose of taking the place of paper scores. However, these inventions still need humans to turn the pages and are not aware of the music structure at all. To further encode logical structure of the score and support a wide range of modern presentations, structured music description languages like MusicXML [8] offer standards for score sharing and archiving. More recently, commercial score printing software like Finale[5] and Sibelius[6], and score accompaniment software like SmartMusic[7] can generate a music performance from notation. However, the algorithms that map the score to performance are problematic when nested repeats are involved. Given that popular score editors attempt to perform scores by following control flow, it is surprising that they do not define a syntax for control flow, check scores for validity, or offer a consistent meaning for control flow notation.

More recent study with an emphasis on music control flow modeling led us to develop a score player for HCMP [7]. This work provides data structures and algorithms for finding the dynamic score (similar to the "flattened score" in this work) by unfolding the static (original) score presentation. This work also draws attention to *arrangement* in which performance order is specified as a list of section labels (e.g. "play the intro, verse 1, verse 2, chorus, chorus"), a common practice in popular music. However, the algorithms in this work do not handle nested repeats or provide a systematic way to validate the correctness of notation. Overall, previous work has made great contributions to the notational presentation of the score, while the basic questions of music control flow, such as "what notation is right" and "what is the correct performance order given this notation" are still unanswered. In this work, we provide a systematic scheme to define the "right" notation and produce the flattened score representation without ambiguity.

[4]Music Reader software: http://www.musicreader.net/
[5]Finale music notation software: http://www.finalemusic.com/
[6]Sibelius music notation software: http://www.sibelius.com/
[7]SmartMusic software: http://www.smartmusic.com/

Chapter 3

The framework for defining music control flow

In this chapter, we are going to look at the framework for formalizing music control flow notation. The main idea is to model the score as a source language and compile it to a target representation, the flattened score, as if it is a programming language. The advantage of making a compiler for music notation is that we have a formal way to define the syntax and semantics and construct a compiler from the formal definition using existed techniques. Similar to the general steps of defining a compiler, music control flow can be formalized by following three steps:

1. First, define the elemental symbols that form the score and their corresponding string representation. The process that converts visual notation into string symbols is called "symbolization", which is similar to “lexical analysis” in compiler construction. The string symbols are called tokens and they are the elemental items used in syntax and semantic analysis.

2. Second, define the static syntax for a well-formed score based on formal grammars. We usually use CFG (context free grammar) since it is both intuitive and powerful.

Based on the CFG, users can use existing algorithms (parsers) to analyze the hierarchical structure of a sequence of symbols and find syntax violations if there is no such structure. The resulting hierarchical structure is called a parse tree or an AST (abstract syntax tree, a simpler equivalent representation of the parse tree).

3. Finally, define the meaning for each CFG we made in the last step using an attribute grammar. Based on the parse tree, the attribute grammar is used to infer the attribute for each tree node, from which the mapping function $f(k)$ can be obtained. The program converting a parse tree or AST to the target language is called a translator.

The above three steps are used to compile a language, such as the control flow notation, to an existing understandable language such as the flattened score. This is a direct approach to define, validate and understand control flow notation. alternatively, suppose we already know a more powerful and descriptive language for control flow notation, which we call an intermediate language, and the algorithm that converts this intermediate language to the target language, it is also applicable to design of the formal semantics to compile the original control flow notation to this intermediate language and then to the target language using the compiler we already have. This two-step approach is an indirect approach to compiler design. A flow chart of both direct and indirect approaches is shown in Figure 3.1. Later in this thesis (Chapter 4), I will show a compact intermediate language called SDL (score description language) that has better descriptive power and is able to represent meanings conveyed by word-based notation; a way of mixing this new type of control flow notation with the conventional one will be introduced in Section 6.2.

3.1 Symbolize the Score

As is the first step for compilation, the symbolization step converts the visual score to a list of string symbols, each corresponds to a piece of notation or a block of notation. To formalize, we unify these string symbols to the form `(cmd, parameter[1], parameter[2], ...)` in which "cmd" is an unique identifier of the symbol and `parameter[i]` is the *i*-th

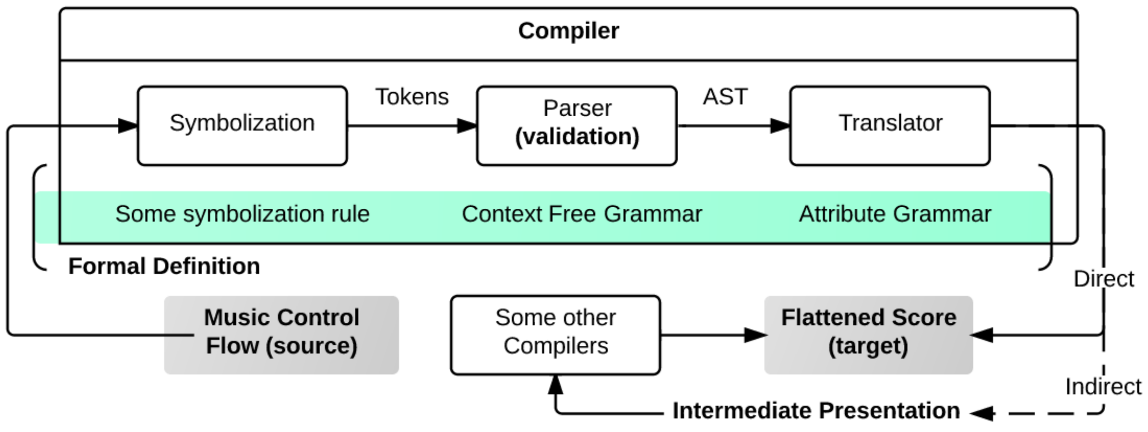


Figure 3.1: Structure of defining, validating and compiling music control flow notation

attribute. The identifier and attributes are separated by commas. The string symbols proposed and used throughout this thesis are shown in Table 3.1. To simplify the notation for the symbols with which no attribute is associated, we use the identifier `cmd` without having to write the brackets. There is a special symbol called "block" that stands for a range of the score with an attribute for starting time and another for duration attribute (both are measured in beats). To separate the problem of control flow semantics from the (much) larger and more general problem of music notation semantics, we simply label each block of notation between control flow symbols and assume the meaning and duration of a block is known. In a real implementation, we divide each block into measures and beat blocks, and associate them with the actual location (a block of pixels) in the score image. After this string representation is flattened, we are able to recreate a visual flattened score that is read in a sequential way without turning back and forth.

However, the proposed way of labeling blocks does not give us unique representations: for example, two measures of 3 beats can be written as `(b,0,3) (b,3,3)` or maybe a more verbose form like `(b,0,1) (b,1,2) (b,3,2) (b,5,1)`. This does not look like a big problem in the symbolization step, but will result in more complicated grammar design as it turns out in Section 3.3. To ensure that we get an unique list of string symbols from any given score, we can restrict that no two block symbols can be adjacent. Instead, they must be combined into a single one, which gives us `(b,0,6)` for the above example.

Table 3.1: Symbols used to label Control flow notation

Basic Symbols	**Meaning**
(b, k, l)	A block of score starting from beat k with length l
(ts, m, n)	Time signature m/n
$\|$:	Left (forward) Repeat sign
:$\|$	Right (backward) Repeat sign
[	Beginning of an ending
]	Endpoint of an ending
x	Measure repeat sign
DC	*dal Capo*
DC.Coda	*D.C. al Coda*
DC.Fine	*D.C al Fine*
DS	*Del Segno*
DS.Coda	*D.S. al Coda*
DS.Fine	*D.S. al Fine*
Segno	the *Segno* sign
Coda	the *Coda* sign
ToCoda	To Coda (or *Coda* symbol)
Fine	*Fine*
Extended Symbols	**Meaning**
(bar, id)	an alternative symbol for blocks. It means a measure. The attribute id refers to the *id*-th block in the score.
(bo, l, id)	an alternative symbol for blocks. It means a point in the score that offsets its previous measure (if $l > 0$) or its next measure (if $l < 0$) by $\|l\|$ beats.

The symbolization procedure can be summarized as: read the score from left to right, write down corresponding string symbols for each control flow symbol and identify the blocks between them and the starting beat and the duration of each block. An example is shown in Figure 3.2. From left to right, we first find a time signature, which is 4/4, and write down the first symbol `(ts,4,4)`. Then we find the next non-block notation, the left repeat sign and the `Fine` sign. The order of these two signs can be implied from their meanings. Since `Fine` means the end of the piece after repeating from `D.C. al. Fine`, it should go before the left repeat sign. The next step is to decompose the measure between this `Fine` sign and the previous time signature into blocks. Since it is required that no two adjacent blocks can be combined into one, we treat the entire measure as a single block. So we write `(b,0,4)`

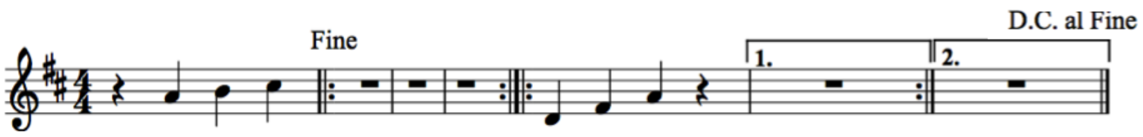


Figure 3.2: Symbolizing the score: this score is converted to the following list of strings to represent control flow: `(ts,4,4) (b, 0, 4) Fine` ||: `(b, 4, 12)` :|| ||: `(b, 16, 4)` [`(b, 20, 4)`] :|| [`(b, 24, 4)`] `DC.Fine`

followed by `Fine` and `|:`. Following this scheme, we can symbolize the score as shown in the caption of Figure 3.2.

Here we only use the basic symbols. The extended symbols, `bar` and `bo`, are the string symbols used in the real implementation: In my software Live Score Display (Section 6.3), users are required to label the barlines and the places between barlines where control flow notation occurs by putting `bar` symbol and `bo` symbols. These alternative symbols together with the time signature can be further converted into blocks either in the symbolization phase or after the flattened score is obtained, depending on the design of the syntax. The advantage of using `bar` and `bo` instead of blocks is that we can dynamically alternate the time signature without modifying the duration of all blocks. The price is a more complicated set of grammars with the risk of ambiguity (Section 3.3.1). The alternative presentation based on the extended symbols is: `(ts,4,4) (bar,1) Fine` ||: `(bar,2) (bar,3) (bar,4)` :|| ||: `(bar,5)` [`(bar,6)`] :|| [`(bar,7)`] `DC.Fine` for the example shown in figure 3.2.

3.2 Context-free Grammar

The context free grammar (CFG) [1] is used to formalize the syntax of sequences of music symbols. A context-free grammar (CFG) consists of terminal, nonterminal, productions, and a start symbol:

1. **Terminals** are the basic building blocks of the entire sequence. In our framework, the terminals are symbolized notations, which include blocks, time signature, control

flow signs and section marks. The symbols defined in Table 3.1 are all terminals.

2. **Nonterminals** are syntactic variables that denote sets of symbol sequences. For example, a score is a nonterminal since it is formed by a sequence of symbols. Defining nonterminals is a central part of this work. We use upper case letters for them.

3. The **productions** consist of:

   (a) A nonterminal in the head or left side of the production

   (b) A body or right side consisting of zero or more terminals and nonterminals, which can be substituted for the head.

   (c) A symbol $\rightarrow$ between the head and the body, meaning the left-side nonterminal can be replaced by the right-side sequences.

4. In a CFG, one nonterminal is distinguished as the **start symbol**. The start symbol is replaced according to productions. After replacement, any remaining nonterminal can be replaced according to productions, and so on, until only terminals remain. The (possibly infinite) set of possible nonterminal strings generated by the productions is called the *language* generated by the grammar.

For example, we can define S as a whole score and E as a score element which can be either a repeat loop or some blocks `b` (here we allow for consecutive blocks; soon we will come to the argument that this would lead to ambiguous grammar definition in the end). Then we can define the following: A score S is a sequence of 1 or more musical elements (E), optionally followed by a right repeat. A grammar for this language (set of scores) is:

```
Starting: S
  (G0) S -> E
  (G1) S -> E :|
  (G2) E -> E E
  (G3) E -> b
```

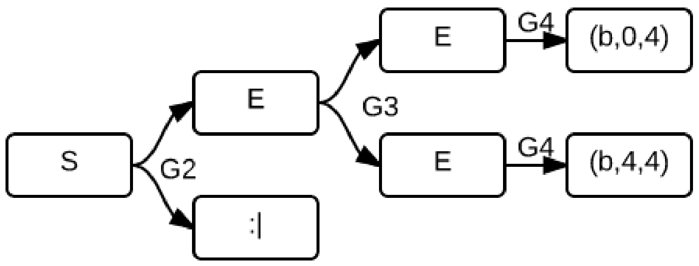


Figure 3.3: Parse tree for (b,0,4) (b,4,4) :||

Here, we consider "b" to be a terminal denoting any block, although we could add productions that expand "b" (now a nonterminal) to terminals of the form "(block, *start*, *duration*)." Based on this CFG, we are able to tell if a sequence of symbols is formed from this grammar using **derivation**. For example, `(b,0,4)(b,4,4):`|| is well-defined from the grammar because we can derive it from S using the productions:

```
S => E :|
  => E E :|                 [use E -> E E]
  => (b, 0, 4) (b, 4, 4) :|  [use E -> b (twice)]
```

We say that a sequence of symbols is a *well defined* score if it can be derived using a grammar. The result of a derivation is a tree structure where each score symbol is a leaf and where each parent node and its children corresponds to a production in the grammar. The leaves from left to right (or more precisely in preorder traversal) will generate the input sequence. The tree is called a parse tree in compilation theory. As an example, the tree for `(b,0,4)(b,0,4):`|| is shown in the Figure 3.3.

3.3 Parsing the grammar

The program that converts a sequence of symbols into a parse tree is called a parser. A parser essentially runs the grammar "backwards," reducing a string of nonterminals to the start symbol by applying productions in reverse. The parser that works for all types of CFG (like a recursive descend parser) would require running time as an exponential function of the number of production, which is inapplicable in practice. Many relaxed parsing algorithms have been developed for grammars with certain restrictions and assumptions. For example LL(1), LR(1) [12] and LALR(1) [17] are the most frequently used parsers for programming languages, but they assume leftmost parseable or 1-lookahead parseable.

In this work, LR(1) is used because it is well-balanced in generality and computational efficiency. LR(1) means scanning from left to right (L) using rightmost derivation (R) and matching the language to productions by looking ahead one terminal (1). The so-called rightmost derivation implies that the production is undetermined until the rightmost symbol is observed and matched. However, LR(1) parser fails when the grammar is so complicated that it requires more than one terminal lookahead. It also fails when the grammar has unresolved ambiguity. The first problem is hard to solve and thus we have to design the grammar such that it is acceptable to LR(1). The second problem can be dealt with using a number of techniques, which will be discussed in details in Section 3.3.1.

The idea of LR parsing is to build the automata that accept the well-formed language for the given grammar and deny the rest. The result is a parsing table in which the rows are the states of the automata, and the columns are the input terminals. The content of a cell can be (1) a pointer to the next state that the corresponding current state (row) and input symbol (column) lead to (shift operation), (2) a production identifier that gives us the matched production (reduce operation), or (3) an empty one meaning syntax error. The parsing table for the previous example is as follows:

```
State | :|  | b   | $   | S   | E
 I0   | .   | s2  | .   | .   | s1
 I1   | s3  | s2  | r0  | .   | s4
 I2   | r3  | r3  | .   | .   | .
 I3   | .   | .   | r1  | .   | .
 I4   | r2  | r2s2| .   | .   | s4
```

In the parse table, a state is denoted as `I` followed by a number starting from 0. The cell content `s2` means a shift operation which changes the current state to I2 and `r3` means to reduce the accepted symbols to a nonterminal using the 3rd production. The term `r2s2` in this case means there are two conflict operations at state 4 when the input is “b”. There are two kinds of conflicts, one is called “reduce-reduce” conflict meaning that there are two productions to match the current input while the other is called “shift-reduce” conflict (the example shown here) meaning we can either use reduce operation to match the previous symbols to the right side of a production or use shift operation to keep tracking on the other unfinished productions. The reason that a shift-reduce error occurs is that the grammar we use here, despite of how simple it is, is ambiguous.

The parser manages a stack data structure to track states (`S1`) and another stack (`S2`) to record the accepted input. Every time a shift operation is executed, the parser puts the new input symbol into its stack `S2`, changes the current state to the one indicated by the table and then pushes the current state into the state stack `S1`. When a reduce operation is executed with the number of terms in the right side of production equal to *n*, the parser pops out the first *n* symbols from the input stack and pushes the left-side symbol of the production into the stack; meanwhile it pops *n* states out of the state stack and sets the current state as the new top state in the stack. To parse a string of symbols into a parse tree, the parser initializes its state to State 0, finds operations in the table for the given input and keeps on altering the states until the input is depleted.

For example, for input `(b,0,4) (b,4,4) :|`, the parser first gets a terminal `b` at I0 and shifts to State 2. Then it gets a terminal `b` for I2 which tells us to use the reduce operation based on Production 3. Then the first block is matched with `E -> b` and a branch of the parse tree is built: `E--b`. Now we go back to the previous state I0 and deal with the new input E produced by the reduce operation. It leads us to a shift operation to state 1. The whole process is shown below:

```
NO.  Input Operation Stack S2  States S1  Production
 1     b      s2     [b        [0 2
 2     b      r3               [0         E -> b
 3     E      s1     [E        [0 1
 4     b      s2     [E b      [0 1 2
 5    :|      r3               [0 1       E -> b
 6     E      s4     [E E      [0 1 4
 7    :|      r2               [0         E -> E E
 8     E      s1     [E        [0 1
 9    :|      s3     [E :|     [0 1 3
10     $      r1     [S        [0         S -> E :|
```

Note that the string `(b,0,4) (b,4,4) :|` does not have issues of ambiguity and thus does not trigger the danger zone of the shift-reduce conflict in the parsing table.

3.3.1 Ambiguity

The problem of ambiguity arises when there are multiple parse trees for the same the sequence of symbols and the same grammar. In the previous example, if we have an input `b b b :|` then we will have ambiguous grammar since we have two ways to group those blocks: one is by the induction `E -> Eb -> bbb`; another is by the induction `E -> bE -> bbb`. In fact, the ambiguity problem has practical meanings associated with how people interpret music notations; it is similar to the question of how to group of numbers we when we are doing basic arithmetic ($+$ and $\times$). For the purpose of demonstration, we change production (G1) to E $\rightarrow$ E : $|$, which gives us another ambiguous grammar. The parse table for this new grammar is

```
State | :|   | b    | $    | S    | E
 I0   | .    | s2   | .    | .    | s1
 I1   | s3   | s2   | r0   | .    | s4
 I2   | r3   | r3   | .    | .    | .
 I3   | r1   | r1   | .    | .    | .
 I4   | r2s3| r2s2| .    | .    | s4
```

Now consider the score b : $|$ b : $|$. It can be parsed in two ways following the steps of by the parse table.

```
b :| b :| => E :| b :| => E b :| => E E :| => E E  => E => S
```

```
b :| b :| => E :| b :| => E b :| => E E :| => E :| => E => S
```

The first deduction (line 1) implies a sequence of two repeated blocks. The second deduction (line 2) implies nested repeats. Notice that while semantics are not inherent in formal grammars, we often associate semantics closely with productions and parse trees.

When there is ambiguity, there is conflict in the parsing table. To resolve the problem, users can either **redesign the grammar** such that it's unambiguous or prioritize the productions. Taking the first approach, we can design an equivalent unambiguous grammar for the same language is as follows

```
S -> S :|
S -> E
E -> E b
E -> b
```

The parsing table is a little bigger but free from conflicts:

State	:|	b	$	S	E
I0	.	s3	.	s1	s2
I1	s4	.	.	.	.
I2	r1	s5	.	.	.
I3	r3	r3	.	.	.
I4	r0	.	.	.	.
I5	r2	r2	.	.	.

The price we pay for finding an equivalent unambiguous grammar is that the new grammar is not as simple and intuitive as the original one and the parsing table may increase in its size due to the more complicated implementation. In most modern parsers, such ambiguity problem can be solved by giving priorities to the productions: If we assume that (G2) always has the highest priority, we will end up with the second deduction (assuming nested repeat) where the first repeat loop E :| is within the loop of the second. This priority statement can also be viewed as preferring a reduce operation when conflicts like r2s3 occurs. Of course, we can also say that (G2) is always the last rule to consider, which will give us the first deduction, assuming parallel repeats. One may notice that the choice whether to set r2 with more priority or less is totally dependent on how we want to interpret the score.

A third way of eliminating ambiguity is to **design a symbolization scheme that gives us simpler source language**; in this way the complexity of syntax is shared by lexical analysis and thus reduced. It is also practical to put a preprocessing phase before syntax analysis if the symbolization step cannot be changed for some reason. In Section 3.1, the procedure that combines the extended symbols like `(bar)` and `(bo)` into blocks is a good example of the preprocessing scheme. As a result, we reduce the number of terminals by restricting that no two blocks can be adjacent. In this case the equivalent grammar for the above example becomes

```
S -> S :|
S -> b
```

The complexity is dramatically reduced and the ambiguity problem is removed. In practice, the above three techniques proposed above are all used together in the final design the grammar. The objective is to find an unambiguous grammar that minimizes the size of the parsing table and thus optimizes the performance of the LR(1) parser.

3.3.2 Error Handling

When the input list of symbols is undefined by the grammar, the parser will hit an empty cell in the parsing table and thus fail to find a valid parse tree. In this case, the parser will generate suggestions for valid input based on the non-empty columns of the failed state.

For example, the grammar for the first-and-second endings is `E -> E [ E :| E ] | E b | b`. The input `b [ b ] b ]` may trigger error messages at the first "end of ending" sign `]`. Since the parser is tracking the production for endings, which is in the form of `b [ b ] b :| [ b ]`, a message such as "expecting right repeat sign" is expected to be shown to the user. In fact, there could be acceptable symbols other than the repeat sign. In the parsing table of the grammar as shown below, the failed state is I17 when the input is `]`. There are three non-empty cells in this row. The complete error message is "`[, :|` or `b` is expected when parsing at `]`"

```
State | [    | :|   | ]    | b    | $    | E
 ...
 I17  | s22  | s21  | .    | s23  | .    | .
 ...
```

3.4 Syntax Directed Translation and Attribute Grammar

A syntax-directed definition is a context free grammar together with attributes and rules. The attributes are associated with grammar symbols and the rules are used to define the relation among symbols [16]. For any terminal or nonterminal `X`, denote `X.a` as some attribute of X. For a production like `X -> Y Z`, the rule can be `X.a = op(Y.b, Z.c)` which contains some attribute symbol, an equal sign and some expression `op` of `Y.b` and `Z.c`. Such an attribute definition is called a *synthesized* attribute since `X.a` is computed from its descendants in the parse tree; if a descendent's attribute such as `Y.b` is in the left side of the rule, then it is called an *inherited* attribute. In this framework, we will stick to synthesized attributes because they are easy to implement and to work with.

Define attribute `.code` as the flattened presentation for all the terminals and nonterminals in our formal control flow framework. Define the operation "`|`" as the concatenation operator that links two lists of symbols to be one. For example, if `E1.code=(b,0,4)(b,4,4)` and `E2.code=`‖`:(b,8,4):`‖ then `E1.code|E2.code=(b,0,4)(b,4,4)`‖`:(b,8,4):`‖. Furthermore, define attributes "`.beat`" and "`.dur`" to be the starting beat and duration for any terminal or nonterminal. For any block `b`, assume b.beat is the "starting beat" of `b` and `len` is the druration of the block. The syntax directed definition for the simple grammar (G1)–(G4) can be defined as follows:

```
Starting: S
  (G1) S -> E    {
    (1.1) S.code = E.code
    (1.2) S.beat = E.beat
    (1.3) S.dur = E.dur            }
  (G2) S -> E :| {
    (2.1) S.code = E.code | E.code
    (2.2) S.beat = E.beat
    (2.3) S.dur = E.dur + E.dur    }
  (G3) E -> E E  {
    (3.1) E.code = E1.code | E2.code
    (3.2) E.beat = E1.beat
    (3.3) E.dur = E1.dur + E2.dur  }
  (G4) E -> b {
    (4.1) E.code = (b, b.beat, b.dur)
    (4.2) E.beat = b.beat
    (4.3) E.dur = b.dur            }
```

Rule (G1) says the flattened representation of a score can be a flattened score element with the same starting beat and duration. Rule (G2) says if the score has a repeat, then the flattened score is two copies of the score element before the repeat sign with the starting beat equal to that of the score element and the duration equal to two times the duration of the score element. Rule (G3) says if a score element is formed by two sub-score elements, then the flattened element is made by concatenating the flattened representation of sub-elements, and the duration is the sum of the lengths of the sub-elements. Finally, if an element is a single block, then all the attributes of the block are copied to the element (G4).

Based on the parse tree, we can apply rules from leaves to root in order to synthesize the attributes of upper levels. For the example shown in Figure 3.3, the attribute flow is shown in Figure 3.4.

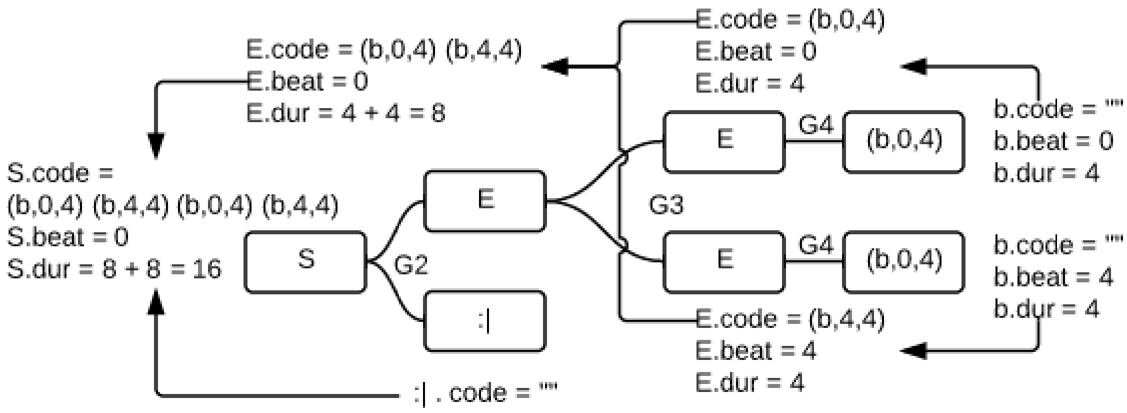


Figure 3.4: Data flow in syntax directed translation

Finally the flattened score is in `s.code` and the duration of the entire score is 16 beats. Although this is a simple example, notice that we have precisely and unambiguously specified how to interpret score control flow notation. Moreover, we have introduced a *meta-notation* based on attribute grammars that allows us to define other interpretations of score control flow. A more complete attribute grammar will be introduced in Section 3.6; and an alternative attribute grammar based on indirect compilation is shown later in Section 4.3. The complete description of the format of this *meta-notation* will be shown in Section 6.1.2 in which useful expressions like loops and conditioned branching will be introduced.

3.5 The Mapping Function

One goal of defining control flow is to obtain a mapping from beats in a performance of the score (i.e. the *flattened* score) to positions or blocks in the original score. We denote the mapping as $f(k)$, where k is the beat position in the flattened score (S.code) and $f(k)$ is the position in the score. The map can be constructed by summing beat durations in the flattened representation S.code to obtain k for each block in the original score.

First, we create a new field in each block in S.code and compute it by a simple induction rule that accumulates durations from the beginning of the score:

1. Base case: $(b,x_1,y_1) \Rightarrow (b,x_1,y_1,x_1)$
   where (b,x_1,y_1) is the first symbol in S.code.

2. ... (b,x_i,y_i,z_i) (b,x_{i+1},y_{i+1}) ... $\Rightarrow$
   ... (b,x_i,y_i,z_i) $(b,x_{i+1},y_{i+1},y_i+z_i)$ ...
   where (b,x_i,y_i) is the i-th symbol in S.code.

The i-th new symbol (b,x_i,y_i,z_i) means that beats from z_i to z_i+y_i map to beats x_i through x_i+y_i in the original score. It gives us the mapping function:

$$f(k) = k - z_i + x_i, \text{ such that } z_i \leq k < z_i + y_i \tag{3.1}$$

For Figure 3.4, we transform S.code:

```
   (b,0,4)   (b,4,4)   (b,0,4)   (b,4,4)
=> (b,0,4,0) (b,4,4,4) (b,0,4,8) (b,4,4,12)
```

which gives us the mapping

$$f(k) = \begin{cases} k & 0 \leq k < 8 \\ k-8 & 8 \leq k < 16 \end{cases} \tag{3.2}$$

The mapping function can only be obtained when the music is deterministic; if the music has to be determined on the fly, we have to do the same to the mapping function, i.e. to calculate it while the music is being played. This indeterministic issue leads to a new conceptual score presentation called *dynamic score* introduced in Chapter 5. The question of obtaining new mapping function becomes the issue of event scheduling in this context, which will be discussed in Section 5.2. In all, the formalization scheme in this chapter is not sophisticated enough to deal with all types of music control flow notation (like word-based notation) and it is also not sufficiently flexible to use in a real performance. But it is the basis of how we start developing more sophisticated approaches as shown in the next two chapters.

3.6 Formalizing nested control flow

As a demonstration of how we can use this framework to formalize music control flow for extended music practice, the following grammar supports nested repeats and unlimited endings. An even more sophisticated implementation is used in a newly implemented score display system which can model word annotation and section-based arrangement. Because space is limited, we only show DS.al.coda in the DS/DC family and limit our attention to the ".code" attribute only.

Define nonterminal S as the score, L as the left-most part of the score, E as score element, LEND as the ending within L and ND as the ending. Add two additional symbols to the original score: { at the beginning and } at the end.

The score S: to be consistent with case 1, the score needs to be decomposed into an L, the leftmost part that could have multiple right repeats, and an E followed by the ending sign $. This gives us

```
S -> L } {
  S.code = L.code;
}
S -> L E } {
  S.code = L.code | E.code;
}
S -> L Segno E ToCoda E DS.coda Coda E } {
  S.code = L.code | E0.code | E1.code | E0.code | E2.code;
}
```

The leftmost group L: If there are no non-paired right repeats in the score, L should be an E. Or this L can produce more L, right repeat sign followed by an E and also an ending structure, the leftmost multiple endings.

```
L -> L :| E {
  L.code = L1.code | L1.code | E.code;
```

```
}
L -> { E {
  L.code = E.code;
}
L -> L :| {
  L.code = L1.code | L1.code;
}
L -> LEND E {
  L.code = LEND.code | E.code;
}
L -> LEND {
  L.code = LEND.code;
}
```

The score element E can be a simple block, a repeated structure, an ending or a DS.al.Coda structure.

```
E -> b {
  E.code = (b, b.beat, b.dur);
}
E -> |: E :| {
  E.code = E1.code | E1.code;
}
E -> |: E ND [ E ] {
  For n = 1 to ND.count
    E0.code = E0.code | E1.code
            | ND.ending[n];
  E0.code = E0.code | E1.code | E2.code;
}
E -> E E {
  E0.code = E1.code | E2.code;
}
```

```
E -> E Segno E ToCoda E DS.Coda Coda E {
  E.code = E1.code | E2.code | E3.code | E2.code | E4.code;
}
```

ND and LEND: Because of the complication of this structure, we need to record each ending to the top level by creating a new attribute ND.ending with a list structure. We also need to use a loop to pair different endings with the constant part.

```
LEND -> { E ND [ E ] {
  For n = ND.count downto 1
    LEND.code = LEND.code | E0.code
              | ND.ending[n];
  LEND.code = LEND.code | E0.code | E1.code;
}
ND -> ND [ E :| {
  ND.count = ND1.count | 1;
  ND.ending[ND.count] = E.code;
}
ND -> [ E :| {
  ND.count = 1;
  ND.ending[1] = E;
}
```

The complete grammar is shown in Appendix A.2. It can be shown that this grammar set is ambiguous because of `E -> EE`, which results in "shift-reduce" errors. The problem can be solved by "prefer reduce to shift", which also indicates "left-associative" for `EE`.

In the real implementation, this grammar is not used; instead, we invent an intermediate language called SDL (Score Description Language) and use it as the target language for the score compiler. The intermediate grammar extends the vocabulary of symbols shown in Section 3.1, and is simpler and more descriptive than the basic symbols. The next chapter shows the vocabulary of this new language and algorithms that further convert the intermediate language to the flattened score.

Chapter 4

SDL: the Score Discription Language

In this chapter, I will introduce SDL, the score description language used for control flow notation, as the intermediate language mentioned in the last chapter. The design objective is to create a compact set of symbols sufficient to annotate most known control flow patterns. This small set of symbols can serve as the means for general score notation in place of conventional symbols. Using them also leads us to understand the common principle behind the control flow notations evolved in its three-hundred year history. Inspired by the assembly language, the vocabulary of SDL is formed by a command followed by parameters as shown in Table 4.1. The notation <N> is a integer; <CV> a capital letter; <P> a loop label; and <L> a jump label.

Table 4.1: Default instruction set for SDL

Symbol	Meaning
(b, $< R >$, $< R >$)	Block symbol, same as the one in Section 3.1
(&, <CV>, <N>)	A section mark; $\boxed{A2}$ is (&,A,2)
(loop, L<P>)	A loop mark labelled by L, similar to left Repeat
(rep, L<N>, T<N>)	Repeat to Label L for T times, similar to Right Repeat
(lb, B<L>)	A Jump mark labelled by B
(jmp, L<P>, T<N>, B<L>)	jump to B at the T-th repetition for loop L
(njmp, L<P>, T<N>, B<L>)	jump to B if not at the T-th repetition for loop L

The intermediate grammar can be used to annotate word-defined scores as in Figure 2.2. It is very useful to compile the original score to this intermediate form and then use the built-in translator to further compile it to a flattened score. The compiler for the intermediate grammar has sophisticated functions such as loop mapping, which helps to relate static score positions back to multiple performance positions (k in $f(k)$). For example, one might want to select the score position as it occurs in the "2nd repeat after the D.S.". This is done by labeling each symbol with a flag that marks the loop surrounding the symbol and the number of repetition of this loop.

As a brief introduction, consider the score below notated in conventional control flow notations.

```
(&,A,1) |: |: (b,0,4) :| (&,A,2) (b,4,4) :|
```

By using a compiler that transforms it to SDL (Section 4.2, we can get the corresponding score notated in SDL as follows:

```
(&,A,1) (loop,L0) (loop,L1) (b,0,4)
(rep,L1,2) (&,A,2) (b,4,4) (rep,L0,2)
```

Using the SDL compiler (Section 4.3), we can obtain the following flattened score:

```
(&,A,1) (b,0,4) (b,0,4) (&,A,2) (b,4,4)
(b,0,4) (b,0,4) (&,A,2) (b,4,4)
```

Each symbol is marked with a flag that shows the mapping from the symbol to loop. The format is `[L1,count1]-[L2,count2]-...` where count n is the number of repetitions of loop Ln. It tells us this symbol is in the *count*1-th repetition of loop $L1$, *count*2-th repetition of loop $L2$, etc. The loop Ln is nested in the L$n-1$ by default. The flags for the above example is shown below. Notice that (b,0,4) appears with four distinct flags, meaning that there are four different positions in the flattened score that refers to this particular block in the original score.

```
[] [L0,1]-[L1,1] [L0,1]-[L1,2] [L0,1] [L0,1]
[L0,2]-[L1,1] [L0,2]-[L1,2] [L0,2] [L0,2]
```

The other feature of SDL is that we can use the section marks to do arrangement, the reordering of sections. In the example above, the score is separated into subsections by section marks. Users can input a rearrangement based on the section marks and the loop mapping. The new section mark, a section label followed by a flag, is called the flattened section mark. Reordering these flattened section marks gives us direct reordering of the flattened score, which is another flattened score rendered by the new arrangement. Note that because A1 is outside the loop cycles, the first play-through of A2 includes (b,0,4). This can be changed by putting A1 inside the loop.:

```
A1-[]      (b,0,4) (b,0,4)
A2-[L0,1] (b,4,4) (b,0,4) (b,0,4)
A2-[L0,2] (b,4,4)

Arrangement: A1-[] A2-[L0,2] A1-[]
```

Then the new flattened score of this arrangement becomes:

```
A1-[]      (b,0,4) (b,0,4)
A2-[L0,2] (b,4,4)
A1-[]      (b,0,4) (b,0,4)
```

The above is a brief preview over the usage of the Score Description Language. This chapter goes into details about the formal definition used in the score compiler that converts control flow symbols to SDL, the algorithm that compiles SDL to flattened score and flags, and finally the arrangement based on flattened section marks.

4.1 The Intuition

Before introducing the formal definition that translates conventional control flow symbols to SDL, the intermediate presentation for the score compiler, we first draw upon the intuition of how SDL works in general. Consider the forward and the backward repeat as a loop statement and implement it in PASCAL [14] and MIPS assembly language [10]:

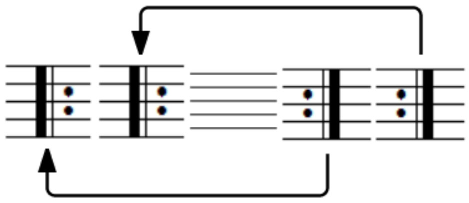


Figure 4.1: Cross-repeat notation

```
Score Notation       PASCAL Code                MIPS Assembly
|:                   for i := 1 to 2 do         L:  beq $t1, $t0, end
 some measures       begin                      ... addi $t1, $t1, 1
                     <statemenst>               g L
:|                   end                        end:
```

The analogy above shows the intuition behind the repeat notation. The forward repeat sign serves as a `for` statement in Pascal or a label in the assembly language. The right repeat sign not only tell us where to repeat but also conveys the number repetitions, 2 by default. Based on this observation, we can invent two instructions: `(loop, L)` and `(rep, L, 2)`; the first is a label corresponding to the `L` label in the assembly language above. The second is similar to the g instruction meaning repeat to L by 2 times. With these two instructions, we are free from the nested structure of |: :| and thus able to make more complicated notation such as cross repeat shown in Figure 4.1:

```
(loop, L1) ... (loop,L2) ... (rep,L1,2) ... (rep,L2,2)
```

In most modern programming language, one may not be able to notate cross repeat structure since we are forbidden from using direct jump statements like `goto` and the `for` loop statement is designed to be restricted to nested structures. Such a design ensures safer coding environment since we are much less likely to fall into endless loops. In the context of music control flow, however, we are free from this issue if we restrict the `loop` symbol and `rep` symbol to be well paired. It means that no two `rep` signs can go to the same `loop` and each `loop` precedes the corresponding `rep`. This rule makes it less likely to form a interminable loop.

Similarly to the analogy of `for` loops, the endings can also be modeled as loops nested with `if` or `switch` statements.

```
|:                          for i = 1 to 2 do begin
<measures A>                  <statements A>
[                             switch (i) do
<measures B>                     case 1: <statements B>; break;
:| [                          else
<measures C>                     case 2: <statements C>; break;
]                           end; end;
```

As an analogy to the `switch` statement, we design our `(jmp, L, n, P)` instruction meaning if the loop `L` is repeated `n` times, then we jump to the block starting from a label `P`. Also, we create another label symbol `(lb, P)` to pair with the `jmp` instruction. To simplify the design, we also introduce the `(njmp, L, n, P)` instruction which means jump when the condition is not satisfied. In this way the switch statement can be written as:

```
switch(i) begin     =>    (loop, i)
case 1:             =>    (njmp, i, 1, P1)
  <statements A>    =>      ...  A
  break;            =>    (lb,   P1)
case 2:             =>    (njmp, i, 2, P2)
  <statements B>    =>      ...  B
  break;            =>    (lb,   P2)
... // n cases      =>      ...
end                 =>    (rep, i, n)
```

Following the above scheme, one can write the loop with endings as `(loop,L) ... (njmp,L,1,P1) ... (lb,P1) (njmp,L,2,P2) ... (lb,P2) (rep,L,2)`.

The objective of using an alternative instruction for jump labels as an addition to `loop` label instructions is to ensure terminability and readability. It can be shown that the three instructions `(lb)`, `(rep)` and `(jmp)` are sufficient for replacing the existed control flow notation in common music practice. Based on the same idea, one can easily identify that "D.S. al. Segno", "DC" and forward-and-backward repeats are equivalent in SDL:

```
Segno        |:                           (loop, L)
 <A>   =     <A>     =      <A>      =    <A>
D.S.         :|             D.C.          (rep, L, 2)
```

If we view the `Fine` sign as a ToCoda sign and put the Coda sign at the end of the piece, then "DS.al.Coda", "DS.al.Fine" and the 2-endings notation are equivalent in the following sense:

```
Segno          Segno          |:         (loop, L)
<A>            <A>            <A>        <A>
ToCoda         Fine           [          (njmp, L,1,P1)
<B>            <B>            <B>        <B>
D.S.al.Coda    D.S.al.Fine    :| [       (lb,P1) (njmp,L,1,P2)
Coda           ||             <C>        <C>
<C>                           ]          (lb,P2) (rep,L,2)
```

In summary, the SDL shows the fact that the conventional notation can be generalized into forward-and-backward repeats or endings. It is a nice property that underlies the conventional notation schemes invented by our ancestors. It also indicates a new way to define the control flow symbols: we can define forward-and-backward repeats or endings and make equivalence statements about other notation patterns.

4.2 Translating to SDL

Based on the analysis in the last section, we show the formal grammar for compiling control flow notation into SDL. To begin with, define the non-terminals `S` as the entire score, `L` as the left-most part of the score (in which single backward repeats are used), `E` as a music group element, `LEND` as the endings in the leftmost part of the left-most score `L`, and `ND` as the endings in `E`. The reason to separate the leftmost part of the score is that we allow multiple single backward repeat before the first forward repeat sign is used. It is the same case for the multiple endings in the left-side of the score.

To simplify the grammar, we add a left bracket sign `{` at the beginning of the piece and a right bracket `}` at the end. Without the the beginning sign, it is hard to tell whether froma single forward look up in LR(1) whether to use L or E. With the brackets, the leftmost part `L` is a series of symbols starting with an `{`. To ensure pairing of `loop` verses `rep`, and `lb` verses `jmp`, define function createLoop() that generates a unique loop label; and the function createLabel() that generates a unique jump label.

The score level starts with a leftmost non-terminal `L` followed by a terminal or `E` that finalizes the leftmost notation and ends with a ending sign `}`.

```
S -> L } {
  S.code = L.code;
}
S -> L E } {
  S.code = L.code + E.code;
}
```

The following are the score level models for the DC/DS families. Note that the `DC/DS. al. Fine` sign are only associated with `S` since it contains `Fine` sign meaning the end of the entire piece.

```
S -> L Fine E DC.Fine } {
  C = createLoop();
  T = createLabel();
  S.code = (looplb,C) + L.code + (jmp,C,1,T) + E.code +
           (rep,C,2)  + (lb,T);
}
S -> L Segno E Fine E DS.Fine } {
  C = createLoop();
  T = createLabel();
  S.code = L.code  + (looplb,C) + E0.code + (jmp,C,1,T) +
           E1.code + (rep,C,2)  + (lb,T);
}
S -> L Segno E DS E } {
  C = createLoop();
  S.code = L.code + (looplb,C) + E0.code + (rep,C,2) + E1.code;
}
S -> L Segno E ToCoda E DS.Coda Coda E } {
  C = createLoop();
  S.code = L.code  + (looplb,C) + E0.code + (jmp,C,1,T) +
           E1.code + (rep,C,2)  + (lb,T)  + E2.code;
}
```

A **Leftmost non-terminal** `L` can be another leftmost non-terminal followed by a non-leftmost non-terminal `E` (or nothing), or leftmost endings followed by `E` (or nothing). `L` stops expanding more leftmost non-terminals when the begin sign `{` is shown. Also note that DC sign is associated with L sign since DC refers to the beginning of the piece, which should be the first symbol inside L. No doubt that DS can also be nested into `L`.

```
L -> L :| E {
  C = createLoop();
  L0.code = (looplb,C) + L1.code + (rep,C,2) + E.code;
}
L -> { E {
  L.code = E.code;
}
L -> L :| {
  C = createLoop();
  L.code = (looplb,C) + L1.code + (rep,C,2);
}
L -> LEND E {
  T = createLabel();
  L.code = LEND.code + E.code;
}
L -> LEND {
  L.code = LEND.code;
}
L -> L DC E } {
  C = createLoop();
  L.code = (looplb,C) + L1.code + (rep,C,2) + E.code;
}
L -> L ToCoda E DC.Coda Coda E } {
  C = createLoop();
  T = createLabel();
  L.code = (looplb,C) + L1.code + (jmp,C,1,T) + E.code +
           (rep,C,2)  + (lb,T)  + E1.code;
}
L -> L Segno E DS E } {
  C = createLoop();
  L.code = L1.code + (looplb,C) + E0.code + (rep,C,2) + E1.code;
}
L -> L Segno E ToCoda E DS.Coda Coda E } {
  C = createLoop();
  T = createLabel();
  L.code = L1.code + (looplb,C) + E0.code + (jmp,C,1,T) +
           E1.code + (rep,C,2)  + (lb,T)  + E2.code;
}
```

The music element level is an independent grouping of blocks and control flow notation other than L. The grammar for forward-and-backward repeats is:

```
E -> |: E :| {
  C = createLoop();
  E.code = (looplb,C) + E1.code + (rep,C,2);
}
E -> |: E ND [ E ] {
  T = createLabel();
  E0.code = (looplb,ND.loop) + E1.code + ND.code +
            (njmp,ND.loop,ND.count,T) + E2.code + (lb,T) +
            (rep,ND.loop,ND.count + 1);
}
```

Also notice that the DS.al.Coda and DS sign can be used in `E` since it is equivalent to forward-and-backward repeat. Then we have:

```
E -> Segno E DS } {
  C = createLoop();
  E0.code = (looplb,C) + E1.code + (rep,C,2);
}
E -> Segno E ToCoda E DS.Coda Coda } {
  C = createLoop();
  T = createLabel();
  E0.code = (looplb,C) + E1.code + (jmp,C,1,T) +
            E3.code + (rep,C,2)  + (lb,T);
}
```

Note that ND.loop is the loop label created in the innermost layer of the grammars of endings; ND.count is the number of endings, which is also counted from the innermost ending to the outer.

The blocks, time signatures and section marks are the minimal independent music elements and thus should be produced by `E`:

```
E -> ts { E.code = ts.code; }
E -> b  { E.code = b.code; }
E -> &  { E.code = &.code; }
```

Similar to the Leftmost notation for DC/DS families, the grammar for DC/DS residing in `E` can be obtained by simply replacing the symbol L with E. Finally, an element can be split into two successive elements:

```
E -> E E {
  E0.code = E1.code + E2.code;
}
```

As a consequence of using E->EE, this grammar is ambiguous. The conflict can be solved by the "left associative", or "prefer reduce when shift-reduce conflict is encountered".

The nonterminal for ending, ND, stands for the bracket sign and the block in between; in the string-based notation it is written as [E :|. Since the number of repetitions has to be infered from the syntax, we add a new field varible ".count" to record the number of layers in ND. Another field variable ".loop" is created to record the loop label because the njmp instruction has to use the loop label as its second variable; the rep instruction can only be created after ".count" is calculated from bottom to top so the loop label has to be created at the very bottom. The above idea is shown in the following code:

```
ND -> ND [ E :| {
  T = createLabel();
  ND.loop = ND1.loop;
  ND.count = ND1.count + 1;
  ND.code = ND1.code + (njmp,T,ND1.count,ND1.loop) + E.code + (lb,T);
}
ND -> [ E :| {
  B = createLoop();
  T = createLabel();
  ND.count = 1;
  ND.loop = B;
  ND.code = (njmp,B,1,T) + E.code + (lb,T);
}
```

4.3 The SDL compiler

In this section we are going to look at the algorithms that converts SDL to a flattened score and flags. There are two different implementation candidates: The first is also based on the idea of a score compiler which creates a parse tree and then translates the tree to tokens and flags. This method limits the grammar to be nested and the flag structure will follow a tree structure. The other method traverses the SDL as if it is being played. In this case, we allow non-nested structures like “cross repeats” as shown in Figure 4.1, but the flag structure will not necessarily following a tree structure. In the real implementation of Live Score Display, the first SDL compiler is used, but one can switch to the second one if more complicated structures are necessary.

4.3.1 Recursive parsing algorithm for SDL

In Section 3.6 and 4.3, we formalize the syntax of music control flow using a CFG, context-free grammar, and the semantics based on a syntax driven approach. We can use a CFG since the control flow symbol themselves have local dependencies; we can define `E` and `L` as an independent group of symbols in which the semantics of each symbol can be determined within this group alone. However, in SDL where the symbol `jmp` can be dependent on the outer most scope of the syntax, the context-free property is somehow ruined. For example, in the following code, the `jmp` instruction refers to the outermost `rep` loop; the conclusion is that CFG is not proper for defining the structure of SDL.

```
(loop,L1)  (b,0,4)  (loop,L2)  (b,4,4) (jmp,L1,2,R3) (b,8,4) (lb,R3)
(rep,L2,3) (b,12,4) (rep,L1,2)
```

Instead of using a syntax-driven approach, we are going to propose a recursive method to fit and parse the symbols by two functions: Function $G(symbols, flags)$ creates flags and unwraps the loops such that long range dependencies is released from every symbol and thus becomes local. The function $H(symbols, flags)$ resolves `jmp` instructions by checking the flags within the range between a paired `jmp` and `lb`. Define S as the SDL score, and V as an sequence of symbols that contains either no `rep` or `loop` instructions or well-paired `rep` or `loop`. Define U as an sequence of symbols that contains no `rep` or `loop`. Note that U is a subset of V. Also define W as a sequence of symbols that contains no `jmp` or `lb`. The recursive algorithm is as follows where each function works as a template and returns the flattened symbols corresponding to its input.

```
input: S; output: G(S)
G(U V,F)    -> { return  G(U,F) G(V,F)  }
G(V U,F)    -> { return  G(V,F) G(U,F)  }
G((loop, L) V2 (rep,L,n), F) -> {
    return H(G(V2, F-[L,1]), F-[L,1]), H(G(V2, F-[L,2]), F-[L,1]),
    ... ,  H(G(V2, F-[L,n]), F-[L,1]);
}
G(b U,F)    -> { return b(F) G(U,F) }
G(ts U,F)   -> { return ts(F) G(U,F) }
G(& U,F)    -> { return &-F G(U,F) }
G((jmp,L,k,B) U,F)  -> {
    if (F contains [L,K])
       return (jmp,B) G(U,F);
    else
       return G(U,F);
}
G((jmp,L,k,B) U,F)  -> {
    if (F contains [L,K])
       return G(U,F);
    else if (F contains L)
       return (jmp,B) G(U,F);
    else
       return G(U,F);
}
```

The return expression is similar to the `.code` field used in CFG, meaning the list of output symbols. The operation `F-[L,k]` means to extend the flag list `F` with a new node `[L,K]`. If F = [L1,1]-[L2,2] for example, then F-[L3,1] is [L1,1]-[L2,2]-[L3,1]. The operation "F contains `[L,k]`" checks if a link `[L,k]` is contained in the chain of flags within `F`. The operation `symbol(F)`, like `b(F)`, means to assign the flag F to the symbol. For section marks `&`, we extend the chain of flags by defining the operation `&-F`. The G operation also checks if a jump instruction meets condition indicated by the flags. If it does, the jump instruction is converted to an unconditioned jump instruction `(jmp, L)`, which will be handled by the *H* function shown below:

```
H ((jmp,L,k,B) U) => { return H(U); }
H ((jmp,B) U1 (lb,B) U2) => { return H(U2); }
H ((jmp,B) U) => { return (jmp,B) H(U); }
H (W) => { return W; }
```

These recursive function are called by "pattern matching": H ((jmp,L,k,B) U) means "use this function if we find a symbol of the format (jmp,L,k,B) at in the leftmost side of the input". These functions are used in order: we first match `(jmp,L,k,B) U` if it is not found, we match `(jmp,B) U1 (lb,B) U2` on and on. When we passed the last rule, an error message shall be generated.

As demonstration, the steps for parsing the instructions shown at the beginning of this section are shown below

```
Input: (loop,L1)  (b,0,4)  (loop,L2)  (b,4,4) (jmp,L1,2,R3) (b,8,4) (lb,R3)
(rep,L2,3) (b,12,4) (rep,L1,2)
```

Let's view it as

```
Input:  (loop,L1) V1 (loop,L2) V2 (rep,L2,3) V3 (rep,L1,2)
Output:
   G((loop,L1) V1 (loop,L2) V2 (rep,L2,3) V3 (rep,L1,2),[])
=> H(G(V1 (loop,L2) V2 (rep,L2,3) V3,[L1,1]),[L1,1])
   H(G(V1 (loop,L2) V2 (rep,L2,3) V3,[L1,2]),[L1,2])
```

```
=> H( H(G(V1,[L1,1]),[L1,1])
      H(G((loop,L2) V2 (rep,L2,3), [L1,1]), [L1,1])
      H(G(V3,[L1,1]),[L1,1]), [L1,1])
   H( H(G(V1,[L1,2]),[L1,2])
      H(G((loop,L2) V2 (rep,L2,3), [L1,2]), [L1,2])
      H(G(V3,[L1,2]),[L1,2]), [L1,2])
```

Since `V1 = (b,0,4)` and `V3 = (b,12,4)`, it is obvious that `H(G(V1)) = V1` and `H(G(V3)) = V3`. The above formula can be simplified as:

```
=> H( V1 H(G((loop,L2) V2 (rep,L2,3), [L1,1]), [L1,1]) V3, [L1,1])
   H( V1 H(G((loop,L2) V2 (rep,L2,3), [L1,2]), [L1,2]) V3, [L1,2])
=> H( V1  H(H(G(V2,[L1,1]-[L2,1]),[L1,1]-[L2,1]),[L1,1])
          H(H(G(V2,[L1,1]-[L2,2]),[L1,1]-[L2,2]),[L1,1])
      V3, [L1,1])
   H( V1  H(H(G(V2,[L1,2]-[L2,1]),[L1,2]-[L2,1]),[L1,2])
          H(H(G(V2,[L1,2]-[L2,2]),[L1,2]-[L2,2]),[L1,2])
      V3, [L1,2])
=> H( V1  H(H((b,4,4) (b,8,4),[L1,1]-[L2,1]),[L1,1])
          H(H((b,4,4) (b,8,4),[L1,1]-[L2,2]),[L1,1])
      V3, [L1,1])
   H( V1  H(H((b,4,4) (jmp,R3) (b,8,4) (lb,R3),[L1,2]-[L2,1]),[L1,2])
          H(H((b,4,4) (jmp,R3) (b,8,4) (lb,R3),[L1,2]-[L2,2]),[L1,2])
      V3, [L1,2])
=> H( V1  H((b,4,4) (b,8,4),[L1,1])
          H((b,4,4) (b,8,4),[L1,1])
      V3, [L1,1])
   H( V1  H((b,4,4),[L1,2])
          H((b,4,4),[L1,2])
      V3, [L1,2])
=> H( V1 (b,4,4) (b,8,4) (b,4,4) (b,8,4) V3, [L1, 1])
   H( V1 (b,4,4) (b,4,4) V3, [L1, 1])
```

```
=> (b,0,4) (b,4,4) (b,8,4) (b,4,4) (b,8,4) (b,12,4)
   (b,0,4) (b,4,4) (b,4,4) (b,12,4)
```

Because of space limits, we cannot show explicitly the flags for each step. The resulting flags are:

```
   (b,0,4) - [L1,1]
   (b,4,4) - [L1,1]-[L2,1]
   (b,8,4) - [L1,1]-[L2,1]
   (b,4,4) - [L1,1]-[L2,2]
   (b,8,4) - [L1,1]-[L2,2]
   (b,12,4)- [L1,1]
   (b,0,4) - [L1,2]
   (b,4,4) - [L1,2]-[L2,1]
   (b,4,4) - [L1,2]-[L2,2]
   (b,12,4)- [L1,2]
```

The recursive grammar definition also implies a nested structure of (loop) (rep) pairs which can be thought as a limitation or a feature (no cross repeats that might lead to infinite repetition). Another way of compiling SDL is to simulate a cursor, like the instruction pointer used in executing the machine language, that traverses the score one instruction at a time. In this case, we do not have a syntax for SDL and need to pay attention to the risk of falling into an infinite loop.

4.3.2 Sequential parsing algorithm for SDL

The sequential parsing algorithm starts with a scan over the symbol list to create lookup tables for `loop` and `lb` symbols. The lookup table `Loop(k)` outputs the index of `(loop,k)` in the symbol list; and the lookup table `Lb(k)` outputs the index of `(lb,k)`. Then, we start traversing from the first symbol; an instruction pointer, denoted as `IP`, is used to track the index the symbol being parsed. When we reach a `(loop)` instruction, we put a new flag at the tail of the current flag chain, which is denoted as `F` in this work. When we reach a `(rep)` instruction, we decide whether to increase the flag counter and repeat back to the corresponding `(loop)` sign; or to remove the flag and get out of the loop if we reached the maximal number of repetition. For `(jmp)` symbols, we check if the condition matches the flag chain and if so, we omit all the following symbols until we reach the corresponding `(lb)` sign. For other symbols, we put them in the new list of flattened score if we are not in "omit" mode and assign the current flag `F` to the output symbol. The algorithm is shown in the following pseudo code; the flattened score is formed by the calling order of the output() function.

```
F  = []: Flag; IP = 0: int; // instruction pointer
omit = false;
while (IP < list.length)
    s = list[IP];
    switch (s)
      case b:  if (omit is empty) output(s); break;
      case ts: if (omit is empty) output(s); break;
      case &:  if (omit is empty) output(s); break;
      case (loop, L):
        if (F contains L) F = F-[L,1];
        break;
      case (rep, L, k):
        if (F contains L)
           if (F.L = k) F.remove(L); else F.L = F.L + 1; IP = Loop(L);
        else
           error();
        break;
      case (jmp, L, k, R):
        if (F contains [L,k]) omit = L;
        break;
      case (jmp, L, k, R):
        if (F contains [L,k])
```

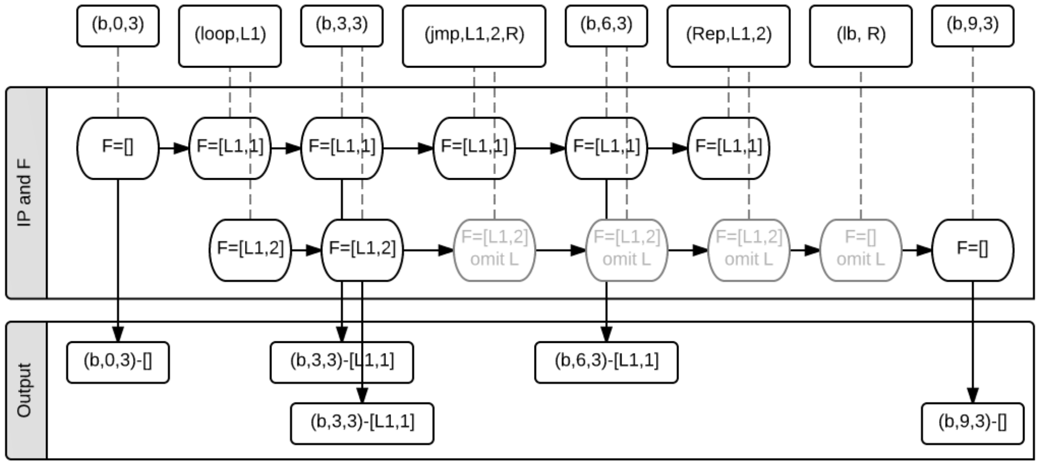


Figure 4.2: An exmaple of the execution of the sequential parsing algorithm

```
        omit = L;
      break;
    case (lb, R):
      if (omit = R) omit = empty;
      break;
  IP = IP + 1;
```

A demonstration is shown in Figure 4.2. The input symbols are: `(b,0,3) (loop,L1) (b,3,3) (jmp,L1,2,R) (b,6,3) (rep,L1,2) (lb,R) (b,9,3)`. In the container named "IP and F", the value of the flag chain and the omit variable is shown. If the block is gray, it means the symbol is ignored and no output will be generated. The container "output" shows the actual output of the flattened score. The reading order is from left to right and top to bottom.

The sequential parsing algorithm works for the cross-repeat example. Figure 4.3 shows the changing of flags at different parts of the score. The score is divided into 5 parts; the blocks below each part are the values of `F` when the instruction pointer points at it. From left to right and top to bottom, the chain of blocks serve as an automata; if the flags `F` are the same at the same part of music but at different time, the sequential scanning algorithm is not terminable. Based on this idea, we can detect infinite loops in the SDL.

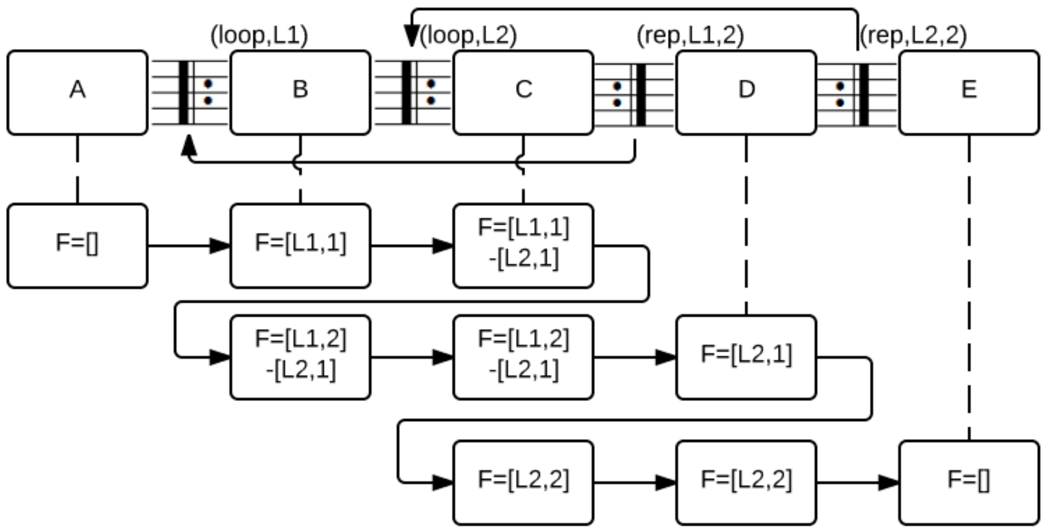


Figure 4.3: Flags in sequential parsing algorithm for non-nested structure; the cross repeats case is used for demonstration

4.4 Arrangement

In the last section, we are able to obtain the "flattened section mark", a section mark associated with a flag chain, by the operation `&-F`. In this section, we are going to examine the concept and implementation of arrangement. The term "arrangement" here is defined as reordering the score by rearranging flattened section marks. Denote a flattened section mark of section S associated with `F` as "`S-F`"; the range of the score corresponding to this section mark is the collection of symbols starting from this flattened section mark to the next mark, in the reading order of the flattened score. For example,

```
(loop,L1) &A (b,0,3) (loop,L2) &B (b,3,3) (rep,L2,2) (b,6,3)
&C (b,9,3) (rep,L1,2)
```

can be flattened as

```
A-[L1,1] (b,0,3) B-[L1,1]-[L2,1] (b,3,3) B-[L1,1]-[L2,2] (b,3,3) (b,6,3)
C-[L1,1] (b,9,3)
A-[L1,2] (b,0,3) B-[L1,2]-[L2,1] (b,3,3) B-[L1,2]-[L2,2] (b,3,3) (b,6,3)
```

```
C-[L1,2] (b,9,3)
```

which contains 6 flattened section marks. The corresponding symbols for each section mark is shown as follows:

```
A-[L1,1]        (b,0,3)
B-[L1,1]-[L2,1] (b,3,3)
B-[L1,1]-[L2,2] (b,3,3) (b,6,3)
C-[L1,1]        (b,9,3)
A-[L1,2]        (b,0,3)
B-[L1,2]-[L2,1] (b,3,3)
B-[L1,2]-[L2,2] (b,3,3) (b,6,3)
C-[L1,2]        (b,9,3)
```

Note that `B-[L1,1]-[L2,2]` contains two symbols: the first symbol `(b,3,3)` has the same flag (`[L1,1]-[L2,2]`) while the second symbol `(b,3,3)` has a different flag `[L1,1]`. Based on the definition, section `B-[L1,1]-[L2,2]` should contains these two symbols since there are no other section marks in between. It is also desirable to limit the symbols to the same flag of the section. An alternative notation for the arrangement is in the form of `F-S`, which means the range of the score labelled by the flag `[L1,1]` that starts from section A and ends at the next mark at the same level as A. In the example above, the symbols that match with `[L1,1]` are:

```
A-[L1,1] (b,0,3) (b,3,3) (b,3,3) (b,6,3) C-[L1,2] (b,9,3)
```

The section marks whose flag is not exactly `[L1,1]` are not contained in the list. `[L1,1]-A` is the range of symbols starting from the section mark A to the next mark C in the same level, which gives us `(b,0,3) (b,3,3) (b,3,3) (b,6,3)`. Note that we can skip over the B section using this notation.

After arrangement, we get a new flattened score presentation as opposed to the one determined merely by control flow notation or SDL. However, in the real implementation, we do not actually create another flattened score from the arrangement; instead, we apply the arrangement in the dynamic score, which is a conceptual presentation of the actual score being played. The dynamic score is a realtime mapping function from the performance time, in *beat*s, to the location of the static score. We can rearrange the sections in performance time and put dynamic control flow symbols like “repeat until cued by the conductor”. The algorithm used in dynamic score is similar on the sequential parsing algorithm shown in Section 4.3.2; and this is the only level of score representation where human interaction is involved.

Chapter 5

The Dynamic Control Flow

In this chapter, we are going to look a conceptual score presentation called the dynamic score, corresponding to the music that is really played at performance time. The term "conceptual" has to two levels of meaning: first, the dynamic score does not have a permanent presentation as static scores do; we do not have a dynamic score unless we start reading the score and performing with it. Second, the dynamic score is undetermined until the performance is over, as opposed to the flattened score where the mapping function is deterministic. If we assume that the performed music is exactly what shown on the score, studying this "conceptual model" would seem senseless since the flattened score presentation is already sufficient to represent the music. But in real performance, there are three scenarios that render the dynamic score necessary, at least, for the purpose of computer-based score following and displaying:

- Undetermined action that is determined by some kind of cues in the performance time. For example "start playing certain measures when the conductor gives a cue".
- Unexpected rearrangement: the conductor alters the arrangement of the music by saying "let's do Section B again" or gives a cue for the same meaning.
- Unexpected jumps: like "go to the second repetition of section B", which is often seen in rehearsal.

The first case has to do with the score notation that relates to the dynamic determination of the score: we call the symbols used such notation as dynamic control flow symbols. In the first section of this chapter, a set of dynamic control symbols called DCL are introduced. And in the second section, we are going to look at a cueing scheme that works for both DCL and the two other cases above.

5.1 DCL: the Dynamic Control Language

Similar to SDL, the dynamic control language, DCL, is a set of instructions of the form `(<ID>,<parameter-1>,<parameter-1>,...)` that perform conditioned branching and jumps to the reading order of the flattened score. The main difference is that these symbols are controlled by “cue”, a signal sent from the conductor or otherwhere that conveys some meaning specified by the dynamic control flow symbol that is about to be played. For example, we define `(dloop,L)` and `(drep,L,inf)` to bracket an infinite loop that terminates until a cue called “break” is received. The actual carrier of the cue can be agreed before the performance; it could be a gesture, a word or even a flash of red light. In all, the design of DCL involves not only the default action but also the action after a certain cue is received. The proposed list of DCL symbols are shown in Table 5.1. One can easily adding more instructions to the proposed DCL symbols to fulfill more complicated controls.

Now the remaining question is how these symbols are actually notated. There are two options: first, we can put dynamic control symbols on the static score, and treat them as non-control flow symbols at the compilation step; we do not treat them as dynamic control flow symbols until the flattened score is obtained; then we use the flags to reassign values to the dynamic labels such that all the dynamic repeat symbols and their corresponding labels are well-paired. Another option is to leave the job to the stage when the flattened score is obtained. We can notate on the flattened score if it is accessible somehow by means of a computer-based display. Using the flattened score has the advantage of better flexibility since we can look inside the unwrapped presentation of the score to add controls for different repetitions of the same part. It also releases the effort of computing and reassigning

Table 5.1: The proposed vocabulary for Dynamic Control Language

Dynamic Symbol	Cue	action
(dloop, L)	default	a label `L` which a dynamic repeat instruction jumps to
(drep, L)	(cue)	repeat to L only when this cue is received
(drep, L, k<N>)	default (break)	repeat to L by k times do not repeat when this cue is received
(drep, L, inf)	default (break)	repeat to L without termination do not repeat when this cue is received
(dlb, R)	default	a label `R` which a jmp instruction jumps to
(djmp, R)	(cue)	jump to R this cue is received
(dnjmp, R)	default (cue)	jump to R not jump to R this cue is received
(dbr, R1,R2...,Rn)	default (i)	jump to the last branch or R1 (if no last branch) branch to Ri $\in$ {R1, R2, ..., or Rn} when a numbered cue of value *i* is recieved
(drbr, R1,R2,...,Rn)	branch	jump to R1, R2, ..., or Rn by the same probability

new dynamic labels to ensure well-paired loops and jumps.

In the real implementation, the arrangement step and the dynamic score are fused together: in the application shown in Section 6.2.2, we allow the users to put dynamic control flow symbols between the flattened sections. Figure 5.1 shows how users notate a branching event among three parallel sections `B1`, `B2` and `B3`, where each section is an infinite loop. First, we put an infinite loop over the course of branches, namely creating a `(dloop,L)`-`(djmp,L,inf)` bracket. Then, we use `(dbr, R1, R2, R3)` instruction to create a conditioned branch, pointing to three dynamic labels, R1, R2 and R3, for cue 1, cue 2 and cue 3 respectively. Finally, we circle each parallel section with its corresponding label `(lb, Ri)` and a jump `(jmp, B)` that leads to the end of the branch (before the infinite repeat instruction). For the purpose of demonstration, the figure is drawn into parallel branches while the sequential presentation of these section marks and the dynamic events is not so clear in its meaning (shown below); it demands a graphical user interface to assist users to notate the structure clearly and correctly. The UI design is shown in section 6.3.2.

```
A-[] (dloop,L) (dbr,R1,R2,R3)
(dlb,R1) B1-[] (djmp,R) (dlb,R2) B2-[] (djmp,R) (dlb,R3) B3-[]
```

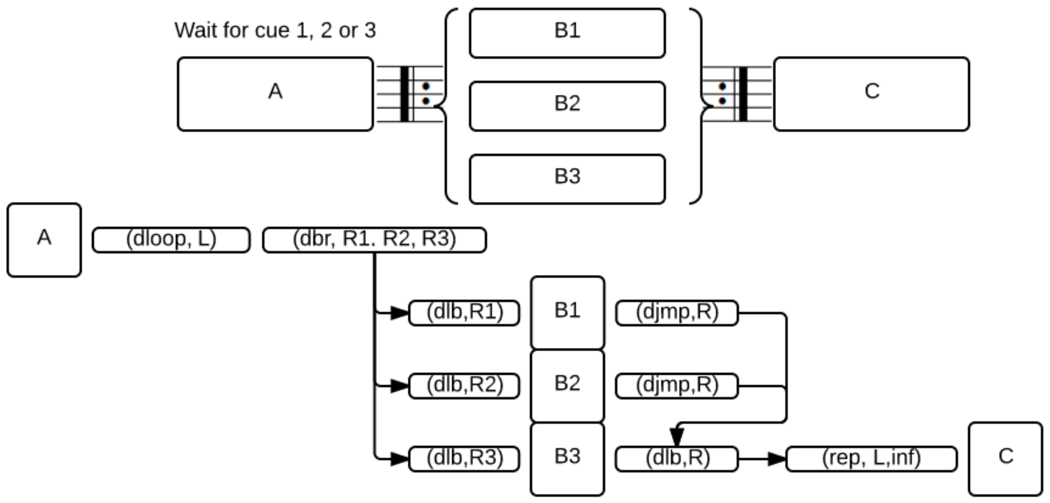


Figure 5.1: Arrangement fused with dynamic control flow notation

```
(dlb,R ) (rep,L,inf) C-[]
```

5.2 Scheduling Dynamic Events

The scheduling problem can be viewed as determine the next symbol and its the play time based on the current symbol and cues. To formalize this process, suppose at performance time `t`, we are going to schedule the play time of the next symbol `s(next)`. We have already received a cue `c` and are going to process the current symbol `s(curr)`. The symbol succeeding `s(curr)` in the flattened score is denoted as `s(curr+1)`. We also define a function schedule(s,t) that schedules the symbol `s` at a future time `t`. The scheduling process for blocks and infinite jumps is demonstrated in the following pseudo code:

```
if (s(curr) is (b,time,cur)) then
   set s(next) to s(curr + 1);
   schedule(s(next), t + cur);
else if (s(curr) is (drep,L,inf)) then
   if (c is "break") then
      set s(next) to s(curr + 1);
   else
      set s(next) to the location of (dloop,L).
   schedule(s(next), t);
else
   show error: unknown symbol.
```

For blocks, we schedule the next symbol to the future time that offsets current time by the amount of its duration. For other symbols defined in Table tab:dcl, we would act instantly. The mapping from the performance time to the score location can be obtained by mapping the current flattened symbol `s(curr)` to the actual symbol on the score. The details implementing these mapping functions will be introduced in Chapter 6.2.

Based on the combined arrangement and dynamic score notation, we can handle unexpected arrangement and jumps easily through the following scheme: at performance time, we keep a cursor that points at the section or the dynamic event being played. The real-time rearrangement (case 2) can be viewed as a change in the future events and thus we simply apply the change to the arrangement without further modification. For unexpected jumps, we use the flags in the flattened score to find the jump point and apply the change by overriding the pointer `s(next)` with the location of the jump point. The algorithm that finds the jump point will be introduced in Section 6.2.

Chapter 6

Implementation and Application

The score compiler is used in a music score display application that, in turn, can be used as part of a human-computer interface for score following, human computer music performance, music education, multimedia databases, etc., where the correct reading order is essential. In this section, we show the actual implementation of the score compiler and the score model manager built on it. As a demonstration, we built a system called Live Score Display (LSD for short) where performers import score images, annotate control flow symbols, re-arrange the score and play the score in real time. With the flattened score representation, the user can browse to any repetition of a repeat, even in heavily nested repeats. The work flow of LSD is show in Figure 6.1:

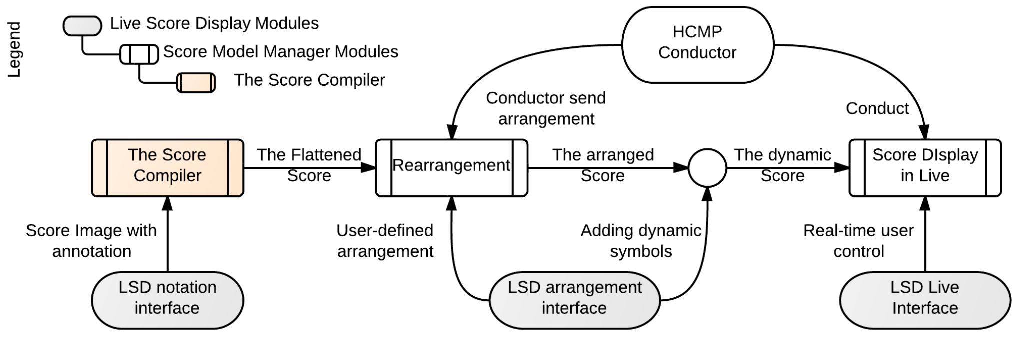


Figure 6.1: Flow chart of Live Score Display

The lowest-level component is the score compiler which takes in the symbolized score and outputs the flattened score or error messages. It is integrated into the Score Model Manager API which controls the logic of score model processing including: (1) converting an annotated score to string symbols; (2) feeding the score compiler with string symbols and producing the flattened score; (3) processing arrangement symbols and outputting the arranged score; (4) fusing the dynamic control symbols together with the flattened score and outputting a data structure prepared for the performance; and (5) managing dynamic control signals and processing the mapping from a real-time beat to the actual location on the score. The LSD application is built upon the score model manager by adding three interfaces that serve as the connection between users and the computer. The three interfaces are: (1) a notation interface where users annotate the score images with control flow symbols; (2) an arrangement interface where users define rearrangement based on section marks, and put dynamic control symbols between sections; (3) the live interface where users play the score with a band using an HCMP controller. In this section, we are going to look at the implementation of each of the levels. Java-styled pseudo code is used to demonstrate the data structures and algorithms.

6.1 The Score Compiler API

We implemented a Java-based API called MCFC (music control flow compiler) that creates a score compiler from a user-defined lexical definition and grammars at start-up and translates string-format score symbols to a flattened score based on the given grammar. Lex and yacc [13] are frequently used for generating code frameworks that create compilers. MCFC however generates the entire compiler directly from the CFG and attribute grammar, allowing users to switch among different grammars without closing the application. This feature is especially useful when dealing with notation from different domains.

MCFC does not assume any model of the score and works in the string representation. TO embed this API, any application must first convert the score to a list of symbols as demonstrated in Section 3.1. In the Score Model Manager, we developed a scheme that converts the human-annotated score structure into symbols, as shown in Section 6.2.1. To make the compiler understand the semantics of these symbols, the user provides the lexical definition file that declares the symbol. This file uses regular expressions similar to Flex [18] software.

The score compiler API offers a simple way to define, validate and compile control flow symbols based on string operations. The work flow of this API is as follows:

- Input symbol description file (.lex) and grammar file (.g). The API validates the syntax and semantics of these files, builds the parsing table, and generates compilation scheme.

- Input the string symbols to compile. If the symbols fail the lexical check, "fail to recognizing symbols" message will be returned. If the symbols fail the syntax check, the score compiler will return the syntax error messages described in Section 3.3.2.

- If the symbols are successfully compiled, a new list of symbols, the compiled score in the target language, will be returned.

Note: the compiled score is not necessarily the flattened score if the grammar is not designed to do so. In LSD, the intermediate presentation in Chapter 4 is used as the target language. The score compiler API does not contain the method that converts the intermediate presentation to the flattened score. This is a function supported by the score model manager API as described in Section 6.2.

6.1.1 Symbol Definition

In MCFC, lexical definition is by regular expressions similar to those used in lex [13] but in a simpler form. Each row in this file defines a symbol's own ID, stringID and parameter formats indicated by regular expression.

```
<ID> (stringID, <Regular expression>, <Regular expression>, ... )
```

For example, to define the string semantic of a block sign, a left repeat, a right repeat and a time signature we can write the content of file as follows:

```
10000 (b, [0-9]+.?[0-9]*, [0-9]+.?[0-9]*})
10011 |:
10012 :|
10004 (ts, [0-9]+, [0-9]+)
```

While [0-9]+.?[0-9]* means an integer or a real number, we provide a pre-defined symbol <N> to mean an integer, <CL> for a capital letter, and <R> for a real number. In this way, we can write the block sign and time signature in a simplified form

```
10000 (b, <R>, <R>)
10004 (ts, <N>, <N>)
```

The symbol analyzer in the MCFC can validate the input string `s` and convert it into a token `t` which is identified by its ID. One can access its *i*-th parameters by `t.parameter[i]`, which is a string even when specified as "`<R>`". When defining the attribute grammar, one can use parameters as numbers by converting them with the string-to-number function.

Later in the Score Model Manager, we will see that the code range 10000 to 10009 is reserved for the default control flow instructions. When using with the Score Model Manager, where the Score Compiler is a component, avoid the reserved code unless overriding them with new symbol definitions. To override predefined symbols, use the same `ID` with different regular expressions. For example, a simplified block notation with only the duration field can be written as follows to override the default definition:

```
10000 (b, <R>)
```

6.1.2 Grammar Definition

The grammar definition is a separate file with extension ".g." The content of the grammar definition is essentially identical to the attribute grammar shown in Section 3.4 except that we use b.attr1 and b.attr2 to assess the first and second parameter instead of b.beat and b.dur. For each nonterminal *x*, the program assumes *x*.code to be the corresponding string for this symbol, and thus instead of `E -> b { E.code = (b, b.attr1, b.attr2) }` , we write it simply as

```
E -> b { E.code = b.code }
```

To create a new field for any symbols just initialize it before it is called. It is the same case for arrays. For example, `E.counter = 1` initializes the counter field to 1 and `E.ending[1] = E2.code` initializes an the no.1 element of array E.ending to the content of E2.code. A "plus" expression is used to add field values. For example, the expression `E.counter = E1.counter + E2.counter` sets the left symbol `E`'s counter field to be the sum of the counters of the two right side symbols. The currently implemented types of user defined field variables are either "code" or integers.

To use loops in the attribute grammar, we can write the following:

```
For <variable> = <starting number> to <ending number>
  <statement>;
  ...
End
```

The `<starting number>` and the `<ending number>` can be a field variable associated with a right-side symbol.

6.1.3 Compilation

After inputing both symbol definition and grammar, the compiler will check if the grammar fits into LR(1). If the given grammar is ambiguous or cannot be parsed by LR(1), the compiler will generate error messages or warnings. For "reduce-shift" errors, the compiler warns about the problem and indicates where the conflicts happens, tut the compiler will continue on with the assumption that "reduce is preferred." When a "reduce-reduce" error occurs, the compiler halts. There is an option to turn on "priority orders" of the grammars so that a "reduce-reduce" error is compromised by using the production with the highest priority.

After the grammars are checked and the parsing table is successfully created, the compiler can start processing strings and compiling it. When an error occurs, the output string is an error message with "ERROR" at the start. There are two types of errors: (1) lexical error, where some input strings are not recognized as tokens; (2) syntax error where the input string does not fit in the proposed CFG grammar. Suggestions are provided to the user for further revision. For example, the wrong string symbols

```
|: (b,1,2) [ |: (b,3,2) :| [ (b,5,2) ] :|
```

will trigger the error message:

```
Syntax error at ([) :
  suggested symbols are(is): (:|)
```

If the input is well-formed,

```
|: (b,1,2) [ |: (b,3,2) :| :| [ (b,5,2) ] :|
```

then no error message is generated but the compiled strings are:

```
 (b,1,2) (b,3,2) (b,3,2) (b,1,2)
 (b,5,2) (b,1,2) (b,3,2) (b,3,2)
 (b,1,2) (b,5,2)
```

6.2 Score Models

In this section, we will look at the score models in term of their data structure and computation. Based on these ideas, we implemented the Score Model Manager, a Java-based API that controls the data flow among different levels of score presentation as well as their backtrace mappings. Isolating different levels of score models will help separate the question of control flow definition in Chapter 3 from its real implementation; we can plug in alternative algorithms between each pair of levels without worrying about disrupting the data flow. To begin with, we define the score models and their logical order:

1. The original score image: the data structure storing the score images in pixels and page numbers.

2. The Notated Score: the machine-readable form of the original score. The location and contents of the notation, including the systems, barlines, music notation and markings, are stored in hierarchical data structures.

3. The Symbolized Score: the score representation parsed from the notated score using the symbolization techniques shown in Section 3.1. Each string symbol is associated with one or more objects from the notated score to connect the visuals to the string symbols.

4. The intermediate form of the score (SDL): the score representation parsed from the symbolized score using the compiler shown in Section 4.2.

5. The Flattened Score: the score representation parsed from the SDL using the recursive SDL compiler 4.3.1. The string symbols are associated with flags; and all flattened symbols obtained.

6. The Dynamic Score: this is the stage where human users can input the arrangement and notate DCL symbols. The output of this stage is the prepared dynamic score for the purpose of performance. The LSD interface will use the Score Model Manager to handle real-time events as well as scheduling the score model and display at the appropriate time.

Although there are six levels (seven when adding the dynamic score scheduler implemented in LSD) of score models that are used in the real implementation, two of them require input from outside sources, and thus are the only stages visible to the users: (1) the score image with the recognized control flow notation overlaid (2) the flattened score shown in the arrangement phase. These two levels correspond to the notation interface and arrangement interface in LSD repectively.

6.2.1 Notated Score

Section 3.1 shows the general method that converts a score from its visual form to a list of strings that represent the notated symbols. However, it does not address how a computer could parse the score image to a machine readable version of the score (the notated score) before beginning the symbolization phase. This step, from a score image to the notated score, can be done by manual annotation or optical score recognition [19]. To use manual annotation, we need to design a routine for the annotation process such that the resulting data structure can be easily computed for the purpose of symbolization. For optical score recognition, we need a generic data structure to describe the digital score such that the symbolization step can be formalized and simplified. This section concerns the manual annotation routine, the data structure and the algorithms converting the proposed data structure to string symbols, and the essential input of the score compiler.

As described in Section 3.1, we want to extract from the score the control flow symbols, the blocks between them and the default reading order without considering the meaning of the control flow symbols. The score is structured into pages and systems. First we identify the order of pages and the location of systems; next, we break the score into pieces of systems and form a long strip by concatenating all these systems. Finally, we need to know the location of control flow symbols and the number of beats between them to infer the blocks. We can use the extended symbol set, including `bar` and `bo` symbols for the purpose of annotation, and calculate the size of blocks from the time signature, barlines and beat offsets. As a result, the proposed scheme for manual annotation contains these three steps.

1. Annotate the systems by specifying the starting point and ending point on the vertical dimension of the image.

2. For each system, annotate the barlines and time points where control flow symbols occur. The time point is defined as the place offsetting its previous or next barline by some certain number of beats.

3. At each barline and time point, label the control flow symbols if any.

A natural design of the data structure for the notated score is shown in the following pseudo code:

```
class Score      {Page pages[]; };
class Page       {System systems[]; Score score; };
class System     {double begin_y, end_y; TimePoint tps[]; Page page;};
class TimePoint  {double loc_x; System system};

class BarLine    inherits TimePoint {
       ControlFlowSymbol symbols[];
}
class BeatOffset inherits TimePoint {
       double offset;
       ControlFlowSymbol symbols[];
}
class ControlFlowSymbol {
       TimePoint tp;
       ... // symbol ID, parameters, etc
}
```

The field `tp` in class `ControlFlowSymbol` is used for backtrack as does the `system` field in class `TimePoint` and the `page` field in class `System`. To infer the string symbols from this hierarchical structure we scan by the natural order of pages, systems and time points; and at each time point, convert its `ControlFlowSymbol` to string symbols. We also need to create a new string symbol for the this time point (either `bar` or `(bo, offset)`. The detailed algorithm is described in the following pseudo code:

```
StringSymbol[] list;
for (page in pages) for (system in page.systems)
    for (tp in system.tps)
      for (symbol in symbols) list.expand(toStringSymbol(symbol));
      list.expand(
        "(bar)" if tp is BarLine:
        "(bo,"+tp.offset+")" if tp is BeatOffset
      );
```

The method `toStringSymbol` converts a ControlFlowSymbol to its string presentation, and associate the string with its original control flow symbol. The following algorithm creates block symbols by merging `bar` and `bo` symbols. The function `create_block` creates a block and associates this block with its corresponding bars and bos in the array `list_of_timepoints`:

```
double beat = 0, beatAligned = 0, start = 0, time_sig = 4;
let list_of_timepoints be array of TimePoint;
for (symbol in list)
  if (symbol is (ts))
    time_sig = symbol.beat_per_measure;
  if (symbol is (bar))
    beatAligned = beatAligned + time_sig; beat = beatAligned;
    list.remove(symbol); list_of_timepoints.append(symbol);
  if (symbol is (bo))
    if (symbol.offset > 0) beat = beatAligned + symbol.offset;
    else beat = beatAligned + time_sig + symbol.offset;
    list.remove(symbol); list_of_timepoints.append(symbol);
  else
    if (start != beat)
      list.addAt(symbol, create_block("b", start, beat-start,
        list_of_timepoints));
    start = beat;
    list_of_timepoints.clear();
```

There are two purposes of associating a score image to string symbol: First, we simplify the score control flow problem to a reading order problem of a set of strings, which better fits into the framework of score formalization as described in Chapter 3 and is computationally effective. Second, after the mapping function of strings is obtained, we are able to further infer the mapping from virtual time (beats) to the actual location of the score images. In this way, we are able to playback the score by traversing the strings, which is efficient and effective in computation.

6.2.2 Static Score

The static score is defined as a collection of string symbols, each of which has a symbol ID, a set of parameters and a pointer pointing to its corresponding control flow symbol in the notated score.

```
struct StaticScore { Symbol symbols[]; };
struct Symbol {ID id; int parameters[]; ControlFlowSymbol pointer;}
```

The static score is the starting point for score flattening. It also stores the function mapping string symbols to the location of a score. The location of a string symbol `s` is

```
Page = s.tp.system.page
System = s.tp.system
Time Point = s.tp
```

One can look up the corresponding location by this index `(Page, System, TimePoint)` or directly by the range: `(s.tp.x, s.tp.system.begin_y)-(s.tp.x, s.tp.system.end_y)` on page `s.tp.system.page`.

6.2.3 The Intermediate Score and The Flattened Score

The static score is then fed to the score compiler. The compiled score is obtained when the static score is free from syntax errors. Following the algorithms in Section 4.2, we will be able to compile the static score. The algorithm shown in Section 4.3.1 is implemented in this Score Model Manager API. So we are able to further compile it to the flattened score with the flags and flattened labels. There is an option in the API for the developers to choose SDL as the intermediate presentation or not. If the developer designs the score compiler specifying that the target language is the flattened score, then he/she can turn off option and plug in their own algorithm to find the flags and section marks.

The flattened score is essentially the re-ordered version of the static score with some additional information necessary for computing mapping queries including "find the third measure of the second repetition of the first repeat loop". In implementation, the flattened score contains the following:

1. A list of "flattened symbols", each of which contains a pointer to the corresponding string symbol in the static score and a "flag" as described in Section 4.4.

2. A set of "flags", each flag has its own list of pointers to the corresponding "flattened symbols" and its own identifier made of a list of "loop label ID" and "repeat count" pairs

3. A set of section IDs; each associated with a `(section)` instruction

4. A set of loop labels, each associated with a `(loop)` instruction

The formal description of the flattened score is as follows:

```
class FlattenedScore {
  flat_symbols[]: array of pointers to Symbol;
  flags[]: array of Flag;
  sections[]: array of Section;
  loopLabels[]: array of pointers to Symbol;
};
class Flag {
 parent: a pointer to Flag;
  loopID: int; //the index of loopLabels[] in FlattenedScore
  count: int;
};
class Section {
  sectionID: string; //e.g. "A1"
  flag: a pointer to Flag;
  sectionSymbol: a pointer to Symbol;
};
```

The primary use of the FlattenedScore is to map queries to the actual location of the score. There are three types of queries: (1) flattened measure numbers or beat numbers; (2) queries concerning repeat numbers like “2nd measure after the third repetition of Loop 4”, “`2m offsets [Loop4,3]`” for short; (3) queries concerning section numbers like “`A1-[Loop1,2]`”. Given a flattened score, the remaining task is to find the mapping from beats to Symbols. This can be easily obtained by summing up the duration of blocks and updating the third parameter, the flattened beats as described in Section 3.5. We define this mapping function as `b2s(beat:float)` and furthermore we can map a beat value `k` to its actual location by the following:

```
class FlattenedScore
  ...
  function b2loc(b): (Page, System, TimePoint)
    Symbol symbol = b2s(b);
    return (symbol.tp.system.page, symbol.tp.system, symbol.tp);
  ...
```

We can use the field variables `loopLabels[]` and `flags[]` to find the flattened symbol and then the score location in the same way. For example, for the third type query which contains a section mark and a flag, the pseudo code is as follows:

```
class FlattenedScore
  ...
  function section2symbol(sectionID, flag): (Page, System, TimePoint)
    sec = find_section(new Section(section, flag));
    if (sec == null) return "cannot find section";
    Symbol symbol = sec.sectionSymbol;
    return (symbol.tp.system.page, symbol.tp.system, symbol.tp);
  ...
  function find_section(section: Section): Section
    for (s in sections)
      if (s.sectionID = section.sectionID || s.flag.equals(section.flag))
        return s;
    return null;
  ...

struct Flag
  ...
  function equals(flag: Flag): bool
    if (parent == null || parent.equals(flag.parent))
      return loopID == flag.loopID & count = flag.count;
    else
      return false;
  ...
```

The additional function in `Flag` recursively matches each level of loopID and the number of repetition indicating when two symbols are in the same level of repetition.

6.2.4 Arrangement and the Dynamic Score

In LSD, the users are provided with (1) the flattened sections, (2) a preview of the arranged score and (3) the interface to write section marks with dynamic control symbols. These three elements are essential for users to make a correct arrangement and thus should be supported by the Score Model Manager.

Section 4.4 describes the arrangement step as re-ordering the section marks, a static section mark followed by a flag chain. As we defined, a section starting from section mark `S` is the block of music between this section mark and the next section mark (or the end). The algorithm rearranging the flattened score by a sequence of section marks is as follows:

```
struct FlattenedScore
  ...
  function rearrange(marks: array of Section): array of Symbol
    rearranged : array of Symbol;
    for (mark in marks)
      sec = find_section(mark);
      ind = flat_symbols.indexOf(sec.sectionSymbol);
      for i = ind : flat_symbols.length
        if (flat_symbols[i] is a section mark) break;
        rearranged.append(flat_symbols[i]);
    end
    return rearranged;
  ...
```

The resulting score is the “arranged score”. This is not the score used in performance; it is used only for the purpose for the user preview. Note that the section marks in the flattened score can be duplicated or omitted. We need to create new section marks for this presentation. For example, we can add the number of duplication at the end of the section name. The arranged score in principle is a modified flattened score and thus its data structure follows strictly the structure of Flattened Score.

As described in Section 5.2, we do not build the rearranged score before the performance but provide sufficient information to do it on the fly. First we divide the flattened score into sections according to the section marks set in the user arrangement; then we keep one pointer, which points at the section or dynamic control flow symbol we are at, and another pointer for the current flattened symbol of the corresponding section being played. We also need a preprocessing step that collects all the loop labels (d_looplabels[]) and jump labels (d_jmplabels[]); and another preprocessing step that rename the duplicated sections and writes them as d_sections[].

```
class DynamicScore {
    dcl[]: an array of dynamic control symbols and section marks;
    flatScore[]: an array of FlattenedScore corresponding to each
                 element in dcl[]; if an element in dcl[] is not
                 a section mark, then the corresponding Flattened
                 Score is empty.

    d_sections[]:   array of Section;
    d_looplabels[]: array of pointers to the dynamic loop labels;
    d_jmplabels[]:  array of pointers to the dynamic jump labels;

    int dcl_pointer;  // a pointer for the dynamic symbol
                      // or the section being played.
    int flat_pointer; // a pointer for the flattened symbol being played.
}
```

The dynamic score also offers several functions for real-time mapping and scheduling in performance. (1) the function next() returns the next symbol to play together with a time point offsetting its previous symbol. (2) the function goto(beat, n), moves the current pointer to the symbol indicated by the beat number "beat" in the flattened score; the parameter n is used to label the number of duplications of this symbol since one may duplicate the flattened symbol several times in the arrangement. (3) goto(looplabel, beat), goto(section, flag, beat) moves the current pointer to the given beat offsetting a certain repeat starting point or a section mark by a certain beat number.

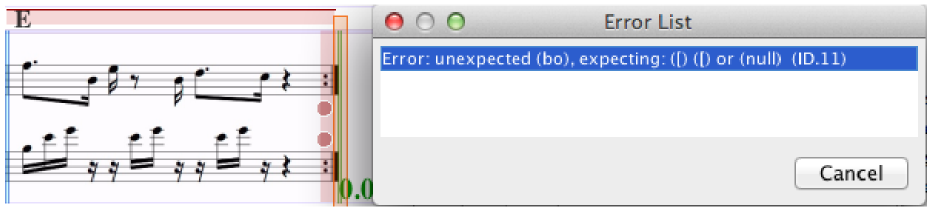
6.3 Live Score Display

The application LSD, Live Score Display, is built upon the Score Model Manager API and MCFC to create a real time score display. It is also an API designed to allow developers to integrate any of the GUI components into their own applications. In the top level view, LSD is composed of three main interfaces: (1) the score annotation interface where users can notate the score images with systems, barlines, voices, control flow symbols and SDL; (2) the arrangement interface where users can view the flattened score, and arrange the dynamic score by putting DCL symbols and section marks together; and (3) the display interface, where users can display the score along with the performance controlled by an HCMP conductor. In this section, we are going to look at the design for each interface; the hierarchy of components; and how the interfaces communicate with the score model manager.

6.3.1 Score Annotation Interface

Ideally, scores would all be machine readable with consistent control flow notation. In reality, we often work with scanned score images, and optical music recognition [19] would at best require extensive manual editing to produce usable data. Our solution is to import scanned or photographed score images and manually annotate the elements that are critical for control flow and display.

The score annotation interface responds to the notated score level in the score model manager. It is provided to import score images and annotate the layout of the score and the control flow symbols. A screenshot is shown in Figure B.1. There are four main panels: the leftmost contains the score structure tools, which are used to draw the system boundary, barlines and places between the barlines where control flow symbols occur. These places and barlines are called time points. Next to the structure toolbar is the symbol toolbar, which is used to place symbols on time points. In the middle is the notation panel where users add annotations. The rightmost panel is the navigator where a small-size score is shown along with the string-format symbols that are used by the score compiler.



```
Error: unexpected (bo), expecting: ([) ([) or (null)  (ID.11)
```

Figure 6.2: Error Message for syntax error: the leftside of this figure shows the measure in the annotation panel where the problem occurs

To compile the score and enter the arrangement mode, the user presses the "GoLive" menu on the menubar. If there are notation errors, a prompt window will display an error message and suggestion (Figure 6.2). To switch between different grammars, the user presses "switch grammar file" or "switch lex file" in the "GoLive" menu.

The annotation interface is composed of four independent parts; each can be used as building blocks for score annotation applications:

The notation panel is a four-layer canvas, each overlaying another. From the bottom to top, the layers are: (1) the score image layer, the most inactive layer for static score image display and scrolling; (2) the symbol layer, which is the main layer for showing systems, blocks and control flow symbols; (3) the note layer, where user can put text and drawings over the score image and the symbols; and (4) the interaction layer where real time animation such as the cursors, selection highlights and tooltips are shown.

The Navigation Panel is a small-sized map shown in the upper right corner of the LSD that shows the currently focused area in the main notation panel. The navigator is also used in the arrangement interface. It highlights the corresponding range in the score when users click a section mark.

The Score Model List is a list under the navigation panel in the LSD, which shows the time points and the control flow symbols each time point contains. This is a very useful component since users may put multiple symbols at one time point but have to adjust the order of these symbols. Users can also use this list to point back to the corresponding location in the score (based on the method shown in 6.2.1), which helps to speed up navigation.

The LSD Toolbar is the graphic square-button toolbar used throughout the LSD. This component allow users to bind a variable to the bottoms of the toolbar and call user-defined functions whenever the variable is rewritten. It also helps to implement complicated selection rules including multiple group selections. In the annotation interface, the score structure toolbar and the control flow symbol toolbar are both implemented using this base component.

To allow all these components to work in harmony, there is a separate component "Annotation Controller" that manages the communication among all components. If users need to integrate some of these components in their own application, they can use the Annotation Controller and access the components needed.

6.3.2 Arrangement Interface

When the user clicks "Go Live" in the annotation interface, the program calls upon the score model manager which goes through the static score level, the intermediate score level, and then outputs the flattened score or error messages. When the score is compiled successfully, the program enters “Arrangement mode” as shown in Figure B.2. The arrangement editor features a navigator which highlights the selected section in the “section selector” panel. In the bottom is the section panel where users can insert or delete more sections. The sections are shown in small square widgets with the name and corresponding loop marks (see Section 4.3). A user can also add dynamic controls between sections, such as “repeat until cue”.

The Arrangement Interface consists of three components:

The mutiple navigation panel : a big scrollable panel consisting of multiple navigators. It is used to highlight the score for the selected section.

The Arrangement Panel : the strip area at the bottom of the arrangement panel, which is used to add or delete section marks or dynamic control symbols.

The Section Toolset : the small square component inside the arrangement panel. Each has a button at the top showing the flattened section name, a text panel in the middle showing the flag chain, and two small insert buttons at its sides. Clicking these buttons will display the menu for insertion or deletion.

Similar to the Annotation Interface, there is an Arrangement Controller that manages the default logics for all these components that developers can directly use it for their own application. The annotation interface itself is also independent. It can load any annotated score file and enter the annotation mode immediately. This feature is useful for performances when users do not want to go through the annotation interface.

6.3.3 Performance Interface

After the score is arranged, the user can enter Live mode and set up an HCMP conductor [4, 5] to play with a band. The HCMP conductor synchronizes sequencers, audio, video, and other media to live performers by broadcasting score position and tempo changes via OSC signals [24]. The LSD registers with the conductor as a “player” object and displays the score’s current and next system to the performer in a manner similar to the lyric display in a Karaoke system. When the control flow jumps, an arrow is shown to point out the direction and unplayed measures are shaded (Figure B.3).

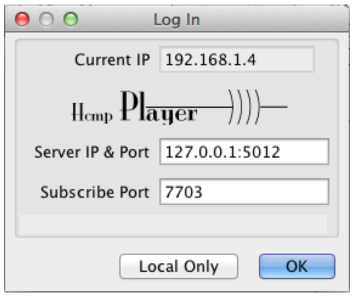
The performance interface is connected with the dynamic score level in the Score Model Manager. It is composed of three visual components and an HCMP controller:

Display Panel : is the panel in the middle showing two lines: one is the system being played and the other is the next system. For each system, the corresponding score location is marked in the small navigation images to the left. If two systems share the same page, only one navigation image will be shown. There are two types of jump in this panel, one is called “play to” which will preserve the currently played system but change the other system we want to jump to. The other is called "go to" which will change the both systems. The former is used in the middle of the performance where the next system is altered because of a new cue. The latter is used when the performance is paused and the conductor wants to play a different part of the score when the performance restarts.

Cue Panel : is the strip toolbar in the upper part of the interface. It shows the available cues in the current music. Clicking it will send out a cue to the HCMP controller and the playing order will be altered. At performance time, we are limited to use these cues; and the display panel will respond to these cues using the “play to” protocol. If the performance is paused, we can use both these cues (only section cues will work) and the flattened score list to go to any place in the score using the “go to” protocol.

Flattened Symbol List : is the component on the right side of the interface. It shows the flattened score after the arrangement (the so-called “Arranged Score” in Section 6.2.4. This list is used for navigation and it can be effective only when a performance is paused. One can select a symbol in the list as the restarting point of the performance.

HCMP Controller : At the bottom of the interface, there is a button called "connect to HCMP". It sets up an HCMP controller that listens to an online HCMP conductor and controls the display shown with it. The setup interface is shown in Figure 6.3.



Current IP 192.168.1.4
Server IP & Port 127.0.0.1:5012
Subscribe Port 7703

Figure 6.3: HCMP player setup interface

With the introduction of the LSD and the concept of arrangement, we need to add a new phase of communication between an HCMP conductor and players to adjust the section marks and the arrangement. When the score is ready, it sends a list of flattened section marks to the conductor; the conductor can send out new arrangement to all players when needed. A demonstration of this new protocol is shown in Figure 6.4.

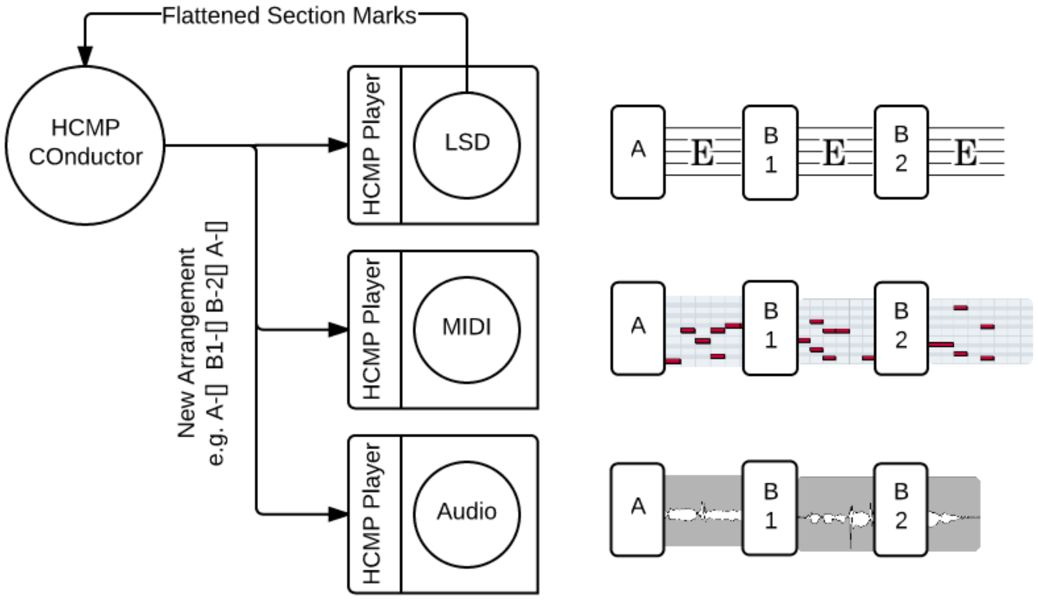


Figure 6.4: Arrangement Protocol for the HCMP architecture

The protocol also requires all HCMP players be able to annotate section marks and rearrange the play according to the arrangement symbols. The synchronization between LSD and all other HCMP players should be done offline before the performance.

Chapter 7

Evaluation and Results

This chapter focuses on evaluating the proposed framework for formal music control flow definition, the proposed grammar and Live Score Display based on the following criteria:

Correctness: including the correctness of the framework, the definition for the well-formed nested notation and the implementation. The evaluation is based on test cases that cover a wide variety of notation patterns, including all the non-nested notation in common practice, a number of of two-level nested notations in extended music practice, several more deeply nested cases and those with syntax error or symbol errors. To evaluate the correctness for cases that have errors, we manually inspect whether error messages can point to the cause of the problem and provide proper suggestions about how to correct it. By showing the test cases and the fact that all tests passed, we claim the correctness of our approach.

Effectiveness a measurement of how applicable is our approach (especially the grammar and SDL) towards the modern practice of music notation. We use real cases, including those found in the Real Book, to show that the four notation patterns in Section 2.2 can all be notated using our approach. For each case, we also show the design strategy to address the effectiveness of our approach.

Extensibility : a measurement of how well we can accommodate other control flow syntax, symbols, instructions and dynamic control based on the existing framework. To address the extensibility, we first show the method with an example, and then discuss its usability and limitation.

Performance : consisting of the performance quality of the Live Score Display system based on a real-time demonstration and user experience assessment based on users' comments and ratings. We collect feedbacks from users to show the strong points and drawbacks.

7.1 Correctness

To evaluate the correctness of (1) the framework, (2) the formal grammar for the well-formed nested notation and (3) the implementation of the Score Compiler and the Score Model Manager, we use three sets of cases to test three different problems at the Score Model's level. We assume that if the application is correct then every implemented component, the definition, and the framework are likely to be correct. The three test sets are: (T1) symbolization test set that evaluates correctness of transforming control flow notation to a static score; (T2) score flattening test set which evaluates the correctness of the score compiler; the cases ranges from the simplest non-nested notation to the highly nested ones; (T3) SDL test set that evaluates the correctness of translating SDL to a flattened score. Similar to T2, test cases of different complexity are used. We expect our system to generate zero errors between the actual output (the flattened score) and the desired output.

Test set T1 contains 32 notated score files in an XML format (`.mxml`). The 24 examples are expected to pass both the lexical and syntax check. Another 4 examples contain syntax errors and thus can pass the symbolization but not the compilation. The final 4 cases cannot pass either the lexical check or the syntax check since it contains undefined symbols. These test cases also vary in the number of pages, the number of systems for a page and the way the score is notated (such as the insertion order of systems and time points). Special cases including scores containing empty pages, empty systems and overlapping systems are also

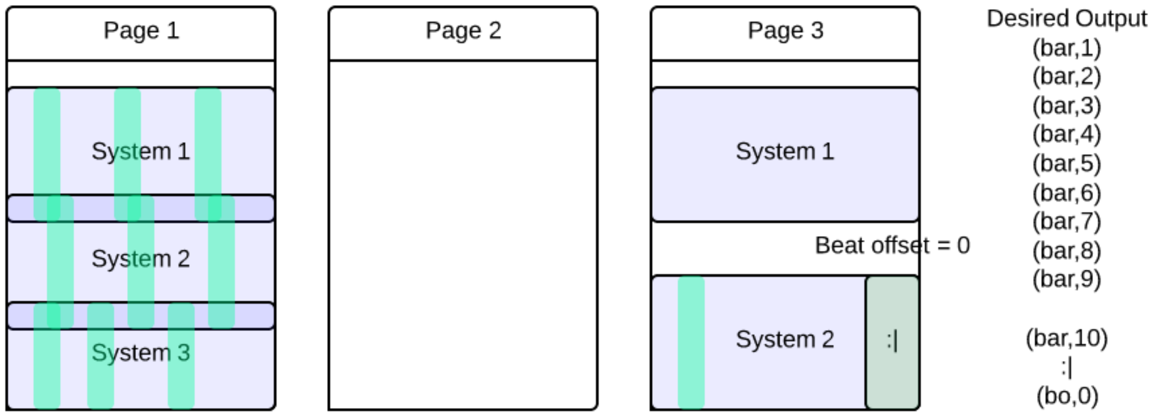


Figure 7.1: Example test case for Set T1: No.8. The visualized score structure and the desired output are shown.

used. Since we only care the correctness of the symbolization, we merely compare the static score output with desired one and whether it can pass the compilation. Table A.1 shows a manifest of the test examples. An example test case (T1.8) is shown in Figure 7.1; note that the vertical thick lines are time points and the last time point is a beat offset symbol with a right repeat. The desired output symbols are listed on the right side of the figure.

Test set T2 validates the score compiler, the recursive algorithm that translates SDL to flattened score, and the design of well-formed nested notation as a whole. The test cases consist of the most typical one-level, two-level and multilevel notation cases as well as problematic ones generated by altering one or two symbols from the correct notation case. To simplify, we use string symbols instead of the Notated Scores. Below shows the categories and several test cases for demonstration:

- (T2.1)-(T2.24) Non-nested basic notation
  - (T2.1)-(T2.4): bar & beat-offset only

```
CATEGORY: bar and bo
      NO: T2.4
   INPUT: (bo,-1) bar bar (bo,-2) bar (bo,2)
 DESIRED: (bo,1,-1) (bar,2) (bar,3) (bo,4,-2) (bar,5) (bo,4,2)
```

  - (T2.5)-(T2.6): Left-right Repeat and two successive left-right repeats.
  - (T2.7)-(T2.10): First and Second Endings; two successive First and Second Endings; three endings.

```
CATEGORY: 1st-and-2nd Endings
      NO: T2.3
   INPUT: bar |: bar [ bar :| [ bar ] bar
 DESIRED: (bar,0) (loop,0) (bar,1) (bar,2)
          (loop,0) (bar,1) (bar,3) (bar,4)
```

  - (T2.11)-(T2.24): Single and two successive DC/DS family notation (note: DC/DS al. Fine can only be used once in a score)

```
CATEGORY: Segno-ToCoda-DSCoda-Coda
      NO: T2.16
   INPUT: bar Segno bar ToCoda bar DSCoda Coda bar
 DESIRED: (bar,0) (loop,0) (bar,1) (bar,2)
          (loop,0) (bar,1) (bar,3)
```

- (T2.25)-(T2.44) two-level nested notation
  - (T2.25)-(T2.28): Nested left-right repeat

```
CATEGORY: Nested Repeats
      NO: T2.25
   INPUT: bar |: bar |: bar :| bar :| bar
 DESIRED: (bar,0) (loop,0) (bar,1) (loop,1) (bar,2) (loop,1)
          (bar,2) (bar,3) (loop,0) (bar,1) (loop,1) (bar,2)
          (loop,1) (bar,2) (bar,3) (bar,4)
```

- (T2.29)-(T2.32): multiple endings nested with left-right repeats

```
CATEGORY: LR repeat nested in 3-endings
      NO: T2.30
   INPUT: bar |: bar [ |: bar :| :| [ bar :| [ bar ]
 DESIRED: (bar,0) (loop,0) (bar,1) (loop,1) (bar,2) (loop,1)
          (bar,2) (loop,0) (bar,1) (bar,3) (loop,0) (bar,1)
          (bar,4)
```

- (T2.33)-(T2.44): Multiple Endings mixed with DS/DC notation

- (T2.45)-(T2.50): Highly nested structure

```
CATEGORY: LR repeat nested in 3-endings nested in DS-Fine
      NO: T2.30
   INPUT: bar Segno |: bar [ bar :| [ bar :| [ |: bar :| ]
          Fine bar DSFine
 DESIRED: (bar,0) (loop,0) (loop,1) (bar,1) (bar,2) (loop,1)
          (bar,1) (bar,3) (loop,1) (bar,1) (loop,2) (bar,4)
          (loop,2) (bar,4) (bar,5) (loop,0) (loop,1) (bar,1)
          (bar,2) (loop,1) (bar,1) (bar,3) (loop,1) (bar,1)
          (loop,2) (bar,4) (loop,2) (bar,4)
```

- (T2.51)-(T2.68) Syntax error examples:
  - (T2.51)-(T2.52) Missing right repeat sign.
  - (T2.53)-(T2.56) Missing `[` or `]` sign in multi-endings
  - (T2.57)-(T2.58) Missing right repeat sign after L

```
CATEGORY: Missing :| in E
      NO: T2.58
```

```
  INPUT: bar Segno bar :| bar DS bar
DESIRED: AT :| expected: [Fine] [toCoda] [DS]
```

- (T2.59)-(T2.64) Missing a component in DS/DC notation.
- (T2.65)-(T2.68) A component in DS/DC notation is nested by mistake inside a repeat of another DS/DC.

```
CATEGORY: DSCoda nested in |: --- :|
      NO: T2.67
   INPUT: bar Segno bar ToCoda |: bar DSCoda Coda bar :|
 DESIRED: AT DS.Coda expected: [:|] [[]
```

Test set 3 is used to validate the algorithm converting user defined SDL to a Flattened Score. There are three sub-categories in this set: (1) nested `loop-rep` cycles without `jmp`; (2) nested `loop-rep` loops with `jmp` referring to the same level; (3) nested `loop-rep` loops with `jmp` referring to the outer level; (4) nested `loop-rep` cycles with `jmp` referring to a loop label undefined in its scope; (5) unpaired `loop-rep` and `jmp-lb`. Below are some examples:

```
CATEGORY: (T3.4) 2-level LOOP-REP
   INPUT: (loop,0) (bar,1) (loop,3) (bar,2) (rep,3,3) (bar,3) (rep,0,1)
 DESIRED: (bar,1) (bar,2) (bar,2) (bar,2) (bar,2) (bar,3)
          (bar,1) (bar,2) (bar,2) (bar,2) (bar,2) (bar,3)

CATEGORY: (T3.9) 2-level LOOP-REP + double JMP
   INPUT: (loop,0) (bar,1) (loop,3) (jmp,1,3,1) (jmp,1,3,3) (bar,2)
          (lb,1) (rep,3,3) (bar,3) (rep,0,1)
 DESIRED: (bar,1) (bar,2) (bar,2) (bar,3) (bar,1) (bar,2)
          (bar,2) (bar,3)
```

```
CATEGORY: (T3.15) 2-level LOOP-REP + outer scope JMP
   INPUT: (loop,0) (bar,1) (loop,3) (jmp,1,3,1) (bar,2) (njmp,2,3,3)
          (bar,20) (lb,1) (rep,3,3) (lb,2) (bar,3) (rep,0,1)
 DESIRED: (bar,1) (bar,2) (bar,3) (bar,1) (bar,2) (bar,3)

CATEGORY: (T3.19) JMP to wrong level
   INPUT: (bar,1) (rep,0,0,0) (bar,2) (lb,0) (bar,3)
          (bar,20) (lb,1) (rep,3,3) (lb,2) (bar,3) (rep,0,1)
 DESIRED: (bar,1) (bar,2) (bar,3)
```

All three test sets are written in the program as unit test modules; the results show that the score model manager passes all three tests and manual check of error messages. Based on these results, we claim that the grammar design, the compiler and the related algorithms are correct. A brief summary generated from the test module is shown below:

```
T1: 32/32 correct;
T2: 50/50 correct; 18/18 syntax error samples passed manual check
T3: 18/18 correct; 4/4 error samples passed manual check
```

7.2 Effectiveness

In this part, we address the effectiveness of our approach by showing how users can annotate the score in the extended practice (Section 2.2). We claim that with the predefined grammar for well-formed nested control flow notation and SDL, users can annotate and flatten scores with (1) Nested structures and Multiple DC/DS notation, (2) word annotation, and (3) arrangement.

The nested repeats and multiple DC/DS structures can be annotated directly by placing the corresponding labels for the control flow symbols shown on the score. Figure 7.2 shows the way the nested multiple DS/DC example in Section 2.2 is notated.

Another type of nested notation is rather a “cross-nested” notation than a well-formed

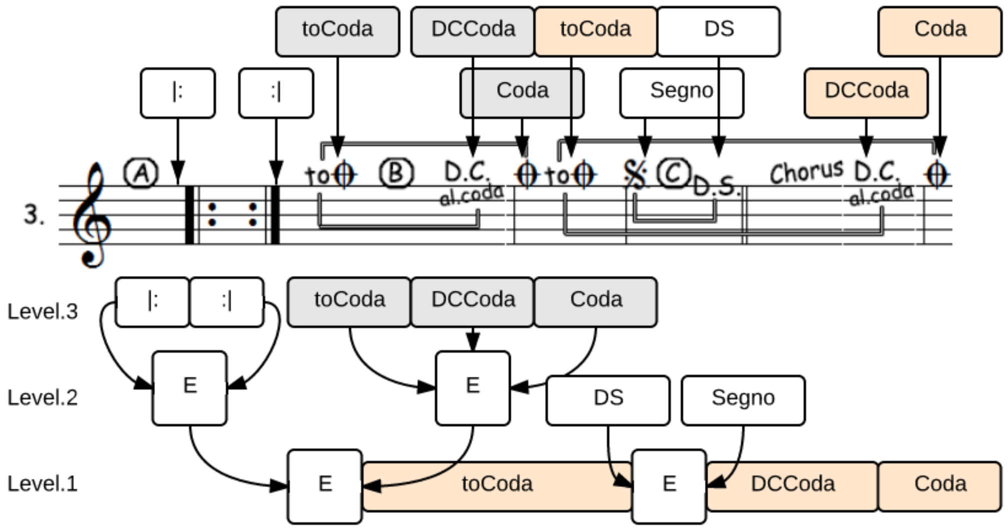


Figure 7.2: Notation for Example 3 in Figure 3.2: notation is done by labeling all the control flow symbols on the score directly (above the stuff). Below is shows the nested structure of these symbols.

nested notation where a component of DC/DS notation is nested inside a repeat. Such notation is regarded as a syntax error in the test case T2.67 as shown in Section 7.1. For example, in the third volume of the Real Book, the piece *Armando's Rhumba*[1] is structured as:

```
(b) |: (b) Segno (b) ToCoda (b) :| (b) DSCoda (b) Coda (b)
```

We can annotate this score with the help of SDL. Regard `DS.al.Coda` as a repeat sign `(rep,L1,2)` pointing back to the beginning of the piece `(loop,L1)`; and regard left repeat sign as `(loop,L2)` and the right repeat sign as `(rep,L2,2)`. We can use a jump sign conditioned on the outer loop `L1` to help us get in and out of the inner repeat loop. The notation is shown in Figure 7.3

Word Annotation can be notated with the help of SDL. Figure 7.4 and 7.5 shows the way that word annotation in Section 2.2 is notated in SDL and mixed SDL.

The arrangement can be done using the flattened section marks. Figure 7.6 shows the

[1]Armando Rhumba by Chick Corea, Real Book III pp.10-11

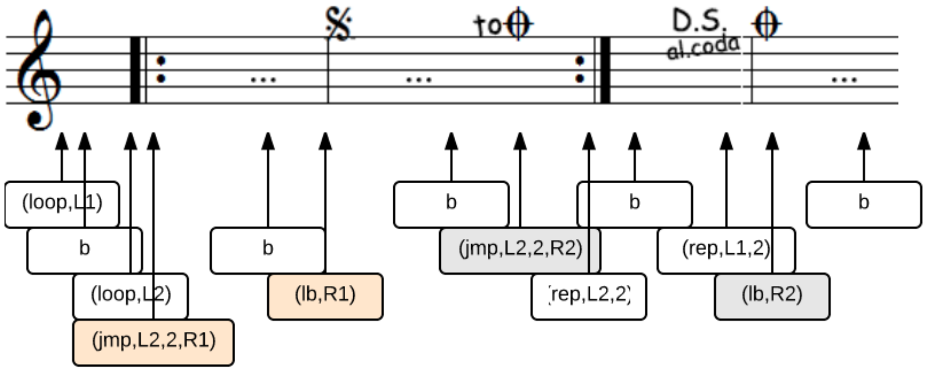


Figure 7.3: SDL Notation for the "cross-nested" example "Armando's Rhumba". Two `jmp` instructions are used to get in and out of the inner repeat.

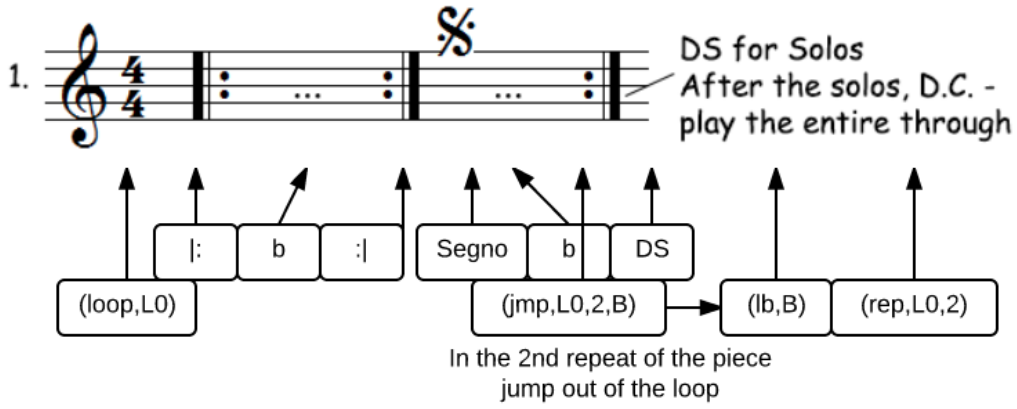


Figure 7.4: Mixed SDL Notation for Example 1 in Figure 3.2: We can understand the word notation as: in the first repetition of the whole piece, we play the block between Segno and DS as usual; and in the second repetition, we only play this section once by jumping out of the loop. In this way, we can define a new loop `L0` around the piece, notate Segno and DS signs as usual and mix it with SDL by putting a jump sign around the `DS` for the condition `[L0,2]`

arrangement signs for Example 2 and 4 in the extended control flow notation.

We successfully show that the three typical notation patterns found in modern practice can be modeled using the control flow symbols or SDL defined in the grammar of well-formed nested control flow notation.

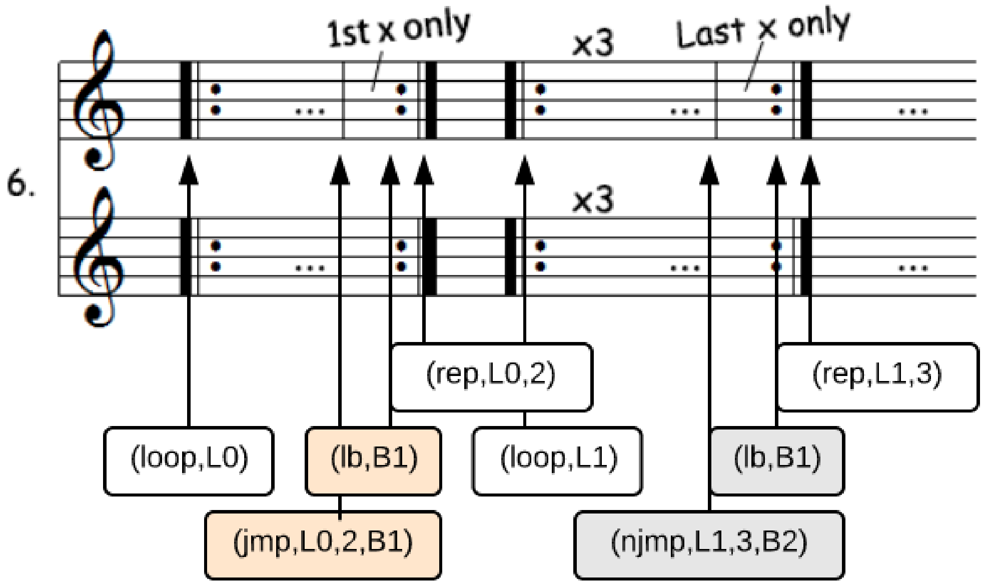


Figure 7.5: Mixed SDL Notation for Example 6 in Figure 3.2: we use `jmp` sign or `njmp` instructions to selectively play the word-notated block

7.3 Extensibility

In this section, we address the extensibility of the framework, the SDL and DCL by showing some examples.

Extend to common practice: the grammar can be rewritten in order to limit the scope of notation to the common practice. The corresponding grammar is shown in Appendix A.3. The same test set T2 is used to validate the correctness of this new grammar; the result shows that all the scores for non-nested notation can be parsed; for the two-level nested notation, only those whose repeats are nested in a DC/DS can be parsed. The result is consistent with the definition in the common practice music in Section 2.1.

Extend Control Flow Symbols: new Control Flow symbols can be defined by adding new entries in the symbol description file (`.lex`) and adding new rules for these symbols in the grammar file (.g). For example, one may want to add "repeat 3 times" symbol `:|3` by adding a new entry:

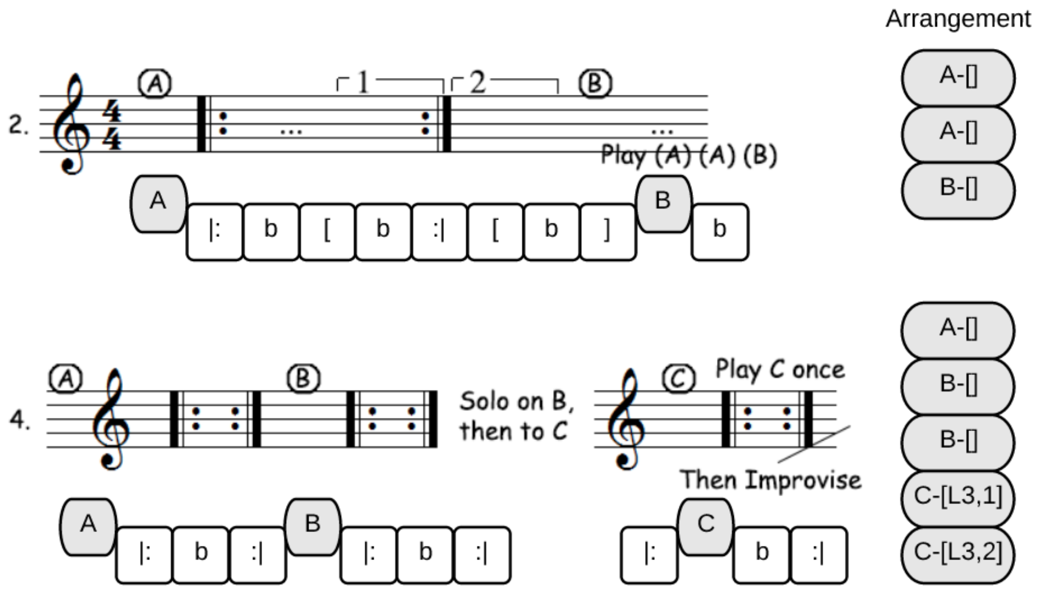


Figure 7.6: Notation for Example 2 and 4 in Figure 3.2 using Section marks and arrangement signs. Note that to divide repetitions of section C in Example 4 into a solo part and an improvisation part, we put the section mark inside the repeat loop. The flattened score will generate two flattened section marks, `C-[L3,1]` and `C-[L3,2]` which can be used for solo and improvisation respectively.

```
10888 :|3
```

and a new rule:

```
E -> |: E :|3 {
  E0.code = E1.code |  E1.code |  E1.code;
}
```

Or the following if one is working with SDL:

```
E -> |: E :|3 {
  L = createLoop();
  E0.code = (loop,L) + E1.code + (rep,L,3);
}
```

Extend SDL vocabulary: new SDL instructions can be also defined by adding new entries in the symbol description file (`.lex`). To further define the semantics of these symbols, users need to redefine the recursive algorithm in Section 4.3.1 in order to support the new notation. Since SDL is already designed to satisfied most notation patterns, it is not recommended to create new instructions in the current instruction set. Below is an example of adding an instruction for a one measure repeat instruction:

```
14888 (X, <N>)
```

In the recursive algorithm one need to implement a new rule:

```
H( bar (X, n) ) => {
  if (n=1) return bar;
  else return bar H( bar (X, n-1) );
}
```

Extend DCL vocabulary: similar to extending SDL, one has to redefine both symbol description file and the scheduling algorithm for the dynamic events.

Extend to other applications: The MCFC, Score Model Manager API and the Live Score Display is designed with consideration of modularity such that they can be used to construct other related applications. Figure 7.7 shows a snapshot of a mxml viewer implemented using the notation panel. Figure 7.8 shows a score painter built upon on the Lexical analyzer inside the score compiler.

To sum up, the extensibility of the current framework is good for new grammars and control symbols but still limited for SDL and DCL. The program has good modularity and its components can be integrated into other music display applications.

7.4 Performance of Live Score Display

To demonstrate Live Score Display, I will perform three original pieces of music in my thesis defense. These three pieces of music show different functions of LSD, including the

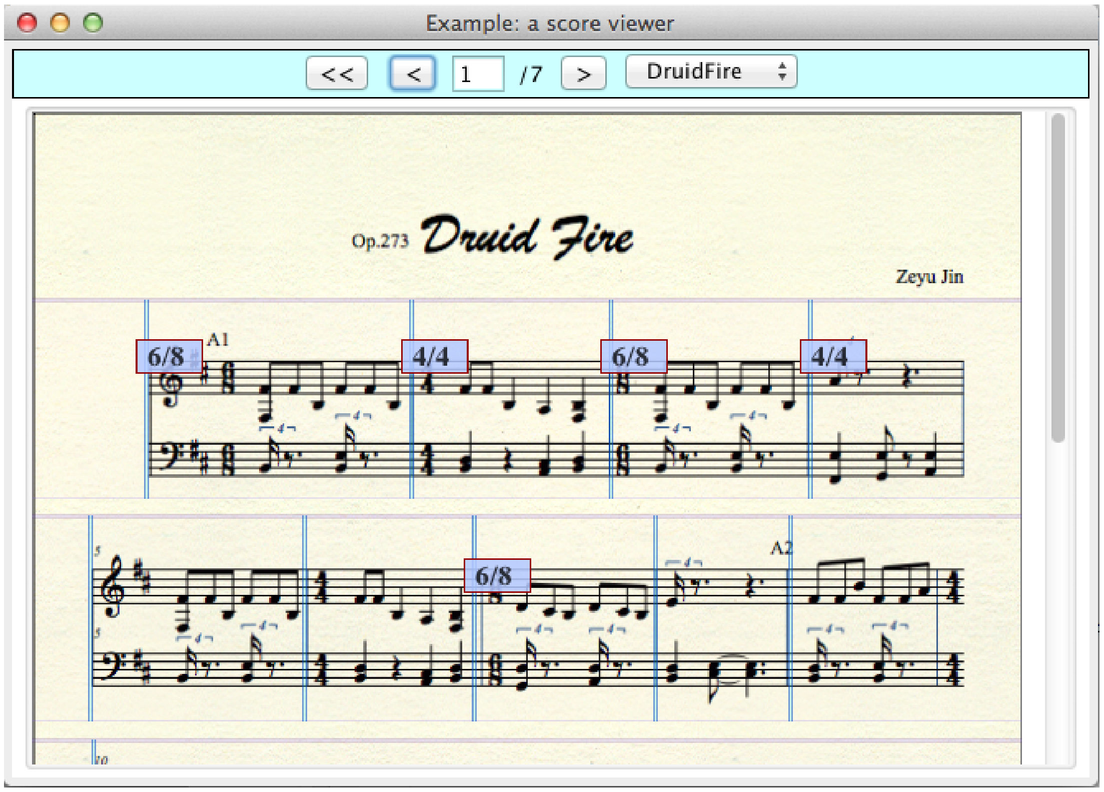
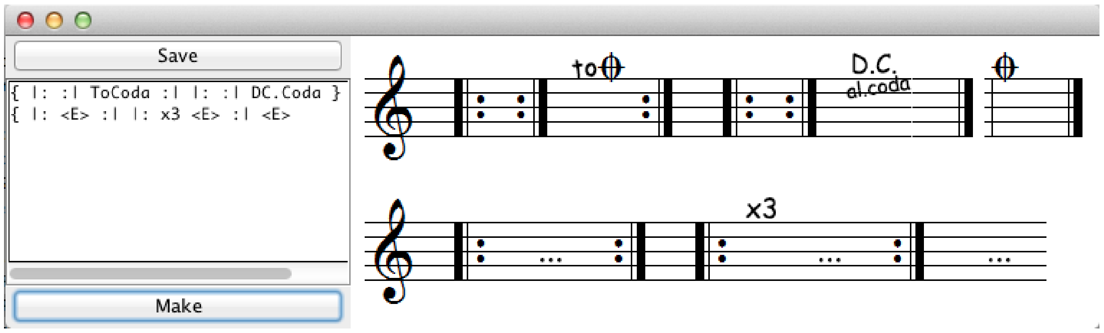


Figure 7.7: Example implementation of a notated score `.mxml` viewer based on the notation panel in LSD. The systems, barlines and the signature changes are marked in the notated score. The piece shown here is "Druid Fire" by Zeyu Jin. The notation is shown as an additional layer overlaying the score image.



```
{ |: :| ToCoda :| |: :| DC.Coda }
{ |: <E> :| |: x3 <E> :| <E>
```

Figure 7.8: Example implementation of a abbreviated score painter built on the lexical analyzer in the score compiler. According to the string symbols, this application paints the abbreviated score used in this thesis.

formalized control flow notation, the dynamic control and the extended use of LSD as a visual storyteller:

"Controversy Annotation" modified from Bach's piece : this piece features extravagant use of control flow notations. The Music Display system and a MIDI player are conducted automatically by an HCMP conductor.

"Cannot Stop" by Zeyu Jin: the entire music is controlled on the fly. In performance time, one of the audience will hold an iPhone controller that controls the HCMP conductor to send out cues; the meaning of the cues are not told to the audience so he/she is supposed to give random cues that affect the piece. This piece requires two human performers, a HCMP-MIDI player and an audio drum machine. The first performer plays the music according to his/her LSD. The second player is responsible to conduct using HCMP controller by listening to the audio. An additional MIDI player is responsible for accompaniment.

"Hacker's Delight" by Zeyu Jin: this piece uses the score displayer as visual story teller. Two LSD with HCMP controllers will be set up. The computer that runs the first LSD system will be connected to the projector showing the images of the story and the other LSD shows the background music to the performer. A MIDI player and a HCMP Sample player will work in accordance with LSD. An automatic HCMP conductor will be used to control the pace of the entire performance. The audience has the ability to control the story lines using cues; and they can also require rewinding back it they are not satisfied with the end of the story.

To further evaluate the usability of LSD, we invited five testers from music technology program to use this system to annotate and play a score named "Maple Leaf Rag". A short tutorial is given before they started operating on their own. A questionnaire is provided to let them rate each question on the scale of 5 (1-bad, 2-not good, 3-moderate, 4-good and 5-very good). Table 7.1 shows the questions and their average rating.

Some helpful feedbacks include:

1. It is easy to learn LSD with a human instructor within 10 minutes; but when someone

Table 7.1: Tester's feedback result

Question	Average rate	Feedback Entry
How easy to learn LSD?	4.2 (good +)	(1)
How is the usability of the notation interface ?	4.6 (very good -)	(2)
How is the usability of the the arrangement interface ?	3.2 (moderate +)	(3)
How is the usability of the the display interface ?	3.8 (good -)	(4)
How do you rate the overall usability?	4.1 (good +)	(5)
Do you think it is useful in a real performance?	3.6 (good -)	(6, 7)

wants to learn it by oneself, we need to provide clearer instructions and guides.

2. In LSD, the definition of a measure is between a barline and the next barline or the end of the system. It implies that we should not notate the right most barline in every system, or another measure would be introduced. However, it is more intuitive for people to notate the last barline and regard it as a beat offset with value 0.

3. The arrangement interface needs improve, especially the way that flags are shown. Musicians may not find the concept of flags intuitive; showing them in natural language or with some explanation will help boost the overall usability of the arrangement interface

4. Performers have to use it several times before truly adapt to it.

5. The program should display more instructional tooltips to let users, especially musicians, know the functions and operations.

6. LSD can also be used to test people's sight reading.

7. It is not possible to judge the usefulness of LSD without involving the testers in real performance; this is why most testers rate "3" for the sixth question.

According to the ratings and feedback, the system is usable but needs to improve the user experience through its graphical design, tooltips and documentation. It is also advised that theoretical background the grammar, the flags and flattening must be either made clear to the users, or hidden by natural explanation. The arrangement interface is the one that suffers most from the lack of background information and needs improvement to be useful.

Chapter 8

Conclusions and Future Work

The interpretation of control flow symbols in music notation is an intricate problem that existing music theory solves only in a highly simplified and informal manner. We have described a framework that borrows from formal languages, formal semantics, and compiler theory. The framework allows us to describe precisely what scores have valid control flow directives and what these directives mean. Furthermore, we showed that we can model modern practices such as nested repeats and textual directives that have no conventional control flow notation. We can also model the practice we call "arrangement" in which additional directives override control flow conventions to perform the score in a new sequence, including on-the-fly arranging such as vamping a section until cued or taking an optional cut. The correctness and effectiveness of this approach is proved using tests.

Beyond theory, we created a flexible implementation for displaying music notation in live performance, mapping the performance position to the score position, looking ahead to page turns and non-sequential jumps in the score notation, and allowing the user to add annotations and arrangements. The Live Score Display system has a meta-notation system in which new control flow syntax can be introduced.

In the absence of interactive computer music, control flow semantics is mostly a theoretical problem. Regardless of theory, the "meaning" of a score is really whatever the performers choose to perform. However, the problem is much more concrete in the context of interactive systems. Computers and humans must agree on control flow semantics if we expect to perform conventional scores with computers. Computers can “understand” scores or we can “tell” computers what they mean. Either way, we need formal descriptive models to express our intentions. This work offers an initial step toward this goal.

Finally, the future work will focus on the three major limitations of the current framework and software: The first one is the limited extensibility of SDL and DCL: the current implementation does not provide a formal way to extend the vocabulary and the semantics for SDL or DCL; the user has to work in the developer’s level to re-code the algorithm. One way to solve this problem is to use CFG and an attribute grammar to create rules that convert new instructions to the existing ones. For example, when a user want to design the branching instruction `(br,L,n,B1,B2,B3)` that jumps to 3 labels according to the value of L and continues at the instruction `(endbr)`, he can create a rule just as in the score compiler:

```
E -> (br,L,n,B1,B2,B3) (lb,B1) E (lb,B2) E (lb,B3) E (endbr) {
  E.code = (jmp,L,1,B1) | E1.code | (lb,B1) | (jmp,L,2,B1) |
           E2.code | (lb,B2) | (jmp,L,1,B3) | E3.code | (lb,B3);
}
```

To realize this idea, we have to use more sophisticated compiler design including functions such as type checking because the parameters for `br` and `lb` are context sensitive in this rule.

The second limitation has to do with the fact we need users to notate all the systems and barlines in the score, which is quite trivial but a lot of work. One way to improve is to employ optical score recognition techniques [21] to help initially notate the score.

The third task is to improve the user experience of our current score display system. We will re-design the system according to feedback from testers. We also need to conduct user experience research to find out how musicians, non-musicians, technicians and non-technicians feel about the software in order to improve its usability.

Appendix A

Long Definition

A.1 Complete Symbol Definition

```
// preserved symbols
1000  {            // start of the score
1001  }            // end of the score
10000 (b,<R>,<R>)  // block
10001 (bar,<N>)    // barline
10002 (bo,<R>,<N>) // time point
10003 (ts,<N>,<N>) // time signature
10004 (&,<CL>,<N>) // section mark

// standard control flow symbols
10011 |:           // Forward (left) repeat sign
10012 :|           // Backward (right) repeat sign
10013 DC           // D.C. sign
10014 DS           // D.S. sign
10015 DC.Fine      // D.C. al Fine. sign
10016 DS.Fine      // D.S. al Fine. sign
```

```
10017 DC.Coda      // D.C. al Coda. sign
10018 DS.Coda      // D.S. al Coda. sign
10020 Segno        // Segno sign
10021 Fine         // Fine. sign
10022 ToCoda       // To-Coda sign
10023 Coda         // Coda. sign
10030 [            // Start of an "ending"
10040 ]            // End of the last "ending"

// SDL symbols
14001 (loop,<N>)   // loop label
14002 (lb,<N>)     // jump label
14003 (rep,<N>,<N>)      // repeat to P1 by P2 times
14004 (jmp,<N>,<N>,<N>)  // jmp to P3 if P1 is repeated P2 times
14004 (njmp,<N>,<N>,<N>) // jmp to P3 if P1 is not repeated P2 times

// DCL symbols
40011 (dloop,<N>)  // dynamic loop label
40012 (dlb,<N>)    // dynamic jump label
40101 (drep, <N>, <C>)  // repeat to P1 if cue C received
40102 (drep, <N>, inf)  // repeat to P1 until cue C received
40103 (drep, <N>, <N>)  // repeat to P1 P2 times until cue C received
```

A.2 Complete Syntax Definition for Well-Nested Control Flow Notation

```
S -> L } {
  S.code = L.code;
}
S -> L E } {
  S.code = L.code | E.code;
}

// Note that Fine. sign is only associated with S since
// it stands for the end of the piece

S -> L Fine E DC.fine } {
  S.code = L.code | E.code  | L.code;
}
S -> L Segno E Fine E DC.fine } {
  S.code = L.code | E0.code | E1.code | E0.code;
}

L -> L :| E {
  L.code = L1.code | L1.code | E.code;
}
L -> { E {
  L.code = E.code;
}
L -> L :| {
  L.code = L1.code | L1.code;
}
```

```
L -> LEND E {
  L.code = LEND.code | E.code;
}
L -> LEND {
  L.code = LEND.code;
}

// Note that DC. sign is only associated with L since
// it refers to the start of the piece

L -> L DC E {
  L.code = L0.code | L.code | E.code;
}
L -> L ToCoda E DC.coda Coda E {
  L.code = L0.code | E0.code | L.code  | E1.code;
}

// DS. sign can be used in both L and E

L -> L Segno E DS E {
  L.code = L0.code | E0.code | E0.code | E1.code;
}
L -> L Segno E ToCoda E DS.coda Coda E {
  L.code = L0.code | E0.code | E1.code | E0.code | E2.code;
}
LEND -> { E ND [ E ] {
  For n = ND.count downto 1
    LEND.code = LEND.code | E0.code
              | ND.ending[n];
  LEND.code = LEND.code | E0.code | E1.code;
}
```

```
E -> b {
  E.code = (b, b.beat, b.dur);
}
E -> |: E :| {
  E.code = E1.code | E1.code;
}
E -> |: E ND [ E ] {
  For n = 1 to ND.count
    E0.code = E0.code | E1.code
            | ND.ending[n];
  E0.code = E0.code | E1.code | E2.code;
}
E -> E E {
  E0.code = E1.code | E2.code;
}
E -> Segno E DS $ {
  E0.code = E1.code | E1.code;
}
E -> Segno E ToCoda E DS.Coda Coda $ {
  E0.code = E1.code | E2.code | E1.code;
}

ND -> ND [ E :| {
  ND.count = ND1.count | 1;
  ND.ending[ND.count] = E.code;
}
ND -> [ E :| {
  ND.count = 1;
  ND.ending[1] = E;
}
```

A.3 Well-formed common practice notation

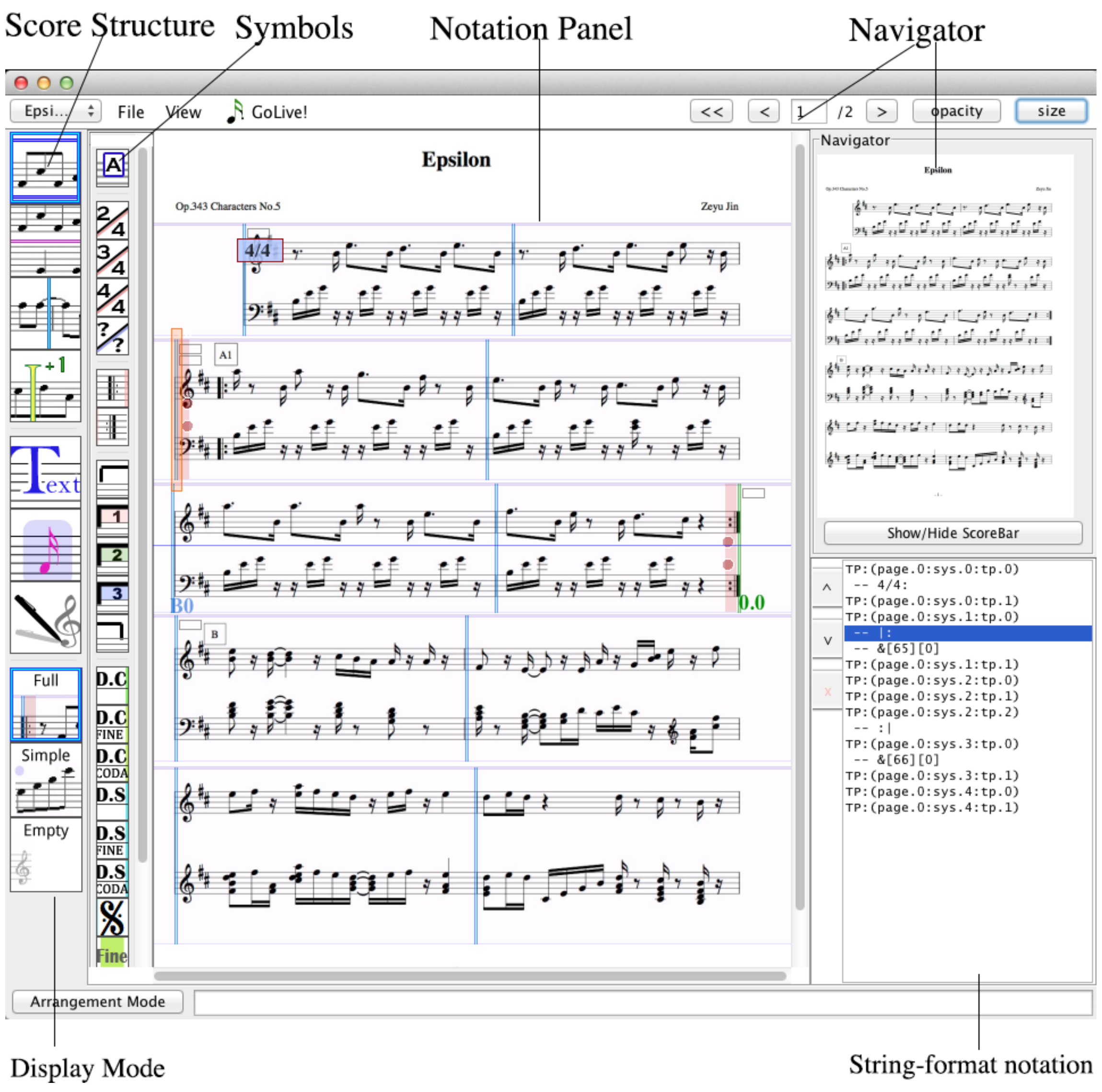
```
S -> b {
  S.code = b0.code;
}
S -> b :| b {
  S.code = b0.code | b0.code | b1.code;
}
S -> b [ b :| [ b ] {
  S.code = b0.code | b1.code | b0.code | b2.code;
}
S -> E DC E {
  S.code = E0.code | E0.code | E1.code;
}
S -> E Segno E DC E {
  S.code = E0.code | E1.code | E1.code | E2.code;
}
S -> E Fine E DCFine {
  S.code = E0.code | E1.code | E0.code;
}
S -> E Segno E Fine E DSFine {
  S.code = E0.code | E1.code | E2.code | E1.code;
}
S -> E ToCoda E DCCoda Coda E {
  S.code = E0.code | E1.code | E0.code | E2.code;
}
S -> E Segno E ToCoda E DCCoda Coda E {
  S.code = E0.code | E1.code | E2.code | E1.code | E3.code;
}
E -> b {
  E.code = b.code;
}
E -> E |: b :| E {
  E.code = E0.code | b.code | b.code | E1.code;
}
E -> E |: b [ b :| b ] E {
  E.code = E0.code | b0.code | b1.code | b0.code | b2.code | E1.code;
}

E -> ε {}
```

Table A.1: Test set T1

No.	#Page	Empty page/system	Time points	Mis-ordered insertion	Compilable
1	1				Y
2	4				Y
3	70				Y
4	1	1p			Y
5	3	1p			Y
6	3	3p			Y
7	3	1s			Y
8	3	1p + 1s			Y
9	3			systems	Y
10	3			blocks	Y
11	3	3s		empty systems	Y
12	3			all randomized	Y
13	4		(bar) only		Y
14	4		multiple (bo)		Y
15	4		4 symbols at one tp		Y
16	4		wrong-ordered (bo)		Y
17	4			symbols	Y
18	4		SDL used		Y
19	4		DCL used		Y
20	4	Real score: Maple Leaf Rag			Y
21	2	Real score: Ave Maria			Y
22	3	Real score: Epsilon			Y
23	4	Real score: Maple Leaf Rag (SDL used)			Y
24	1	Real score: Follow Your Heart by J.Mclauphlin			Y
25	3				Lex Error
26	3				Lex Error (LSD)
27	3				Lex Error (Par)
28	3				Lex Error (DCL)
29	3				Syntax Error
30	3				Syntax Error
31	3				Both Error
32	4	Real score: Maple Leaf Rag, 2 symbols at a tp swapped			Syntax

Appendix B

Live Score Display



```
TP:(page.0:sys.0:tp.0)
 -- 4/4:
TP:(page.0:sys.0:tp.1)
TP:(page.0:sys.1:tp.0)
 -- |:
 -- &[65][0]
TP:(page.0:sys.1:tp.1)
TP:(page.0:sys.2:tp.0)
TP:(page.0:sys.2:tp.1)
TP:(page.0:sys.2:tp.2)
 -- :|
TP:(page.0:sys.3:tp.0)
 -- &[66][0]
TP:(page.0:sys.3:tp.1)
TP:(page.0:sys.4:tp.0)
TP:(page.0:sys.4:tp.1)
```

Figure B.1: Live Score Display: Score Annotation Interface

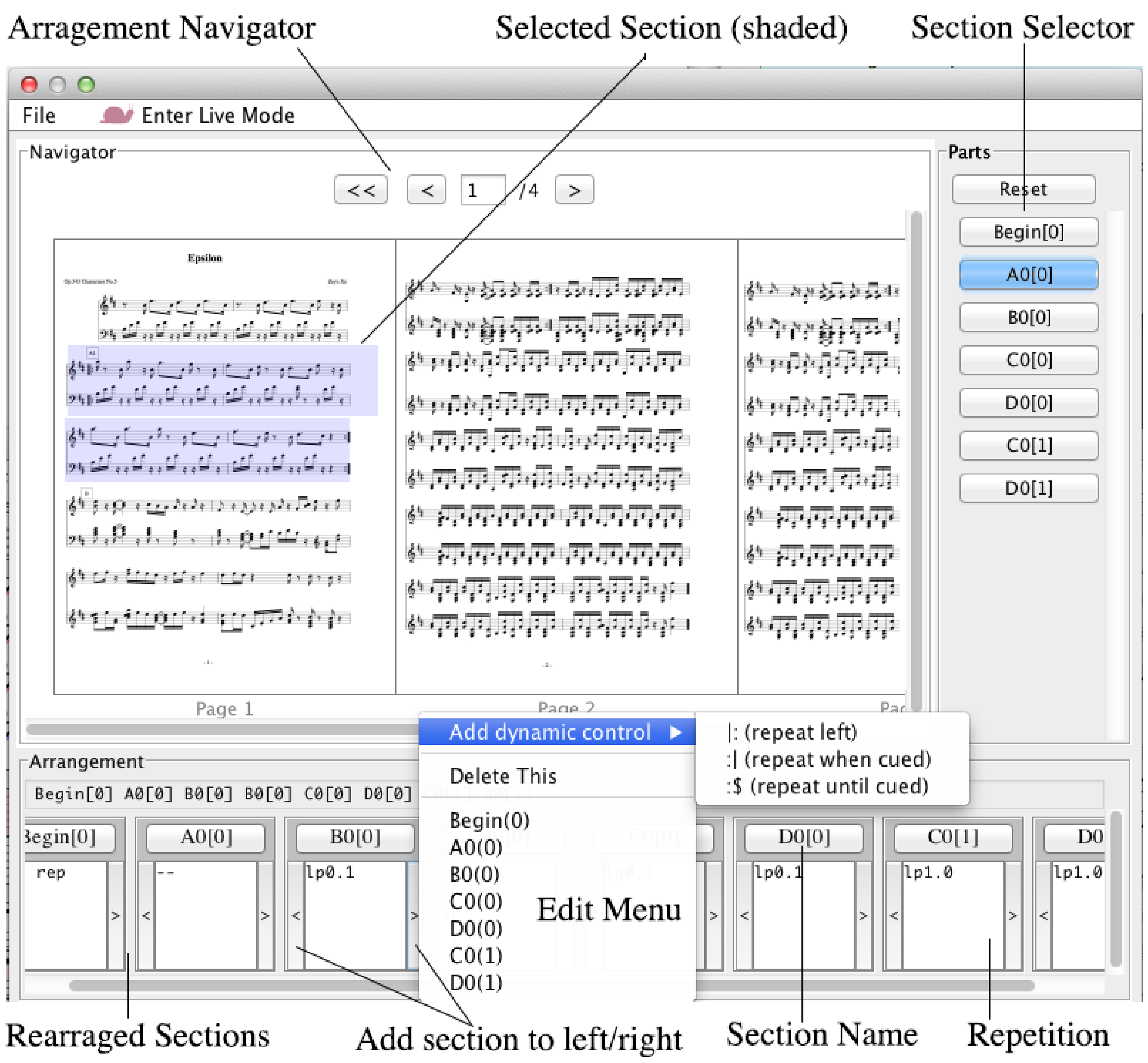
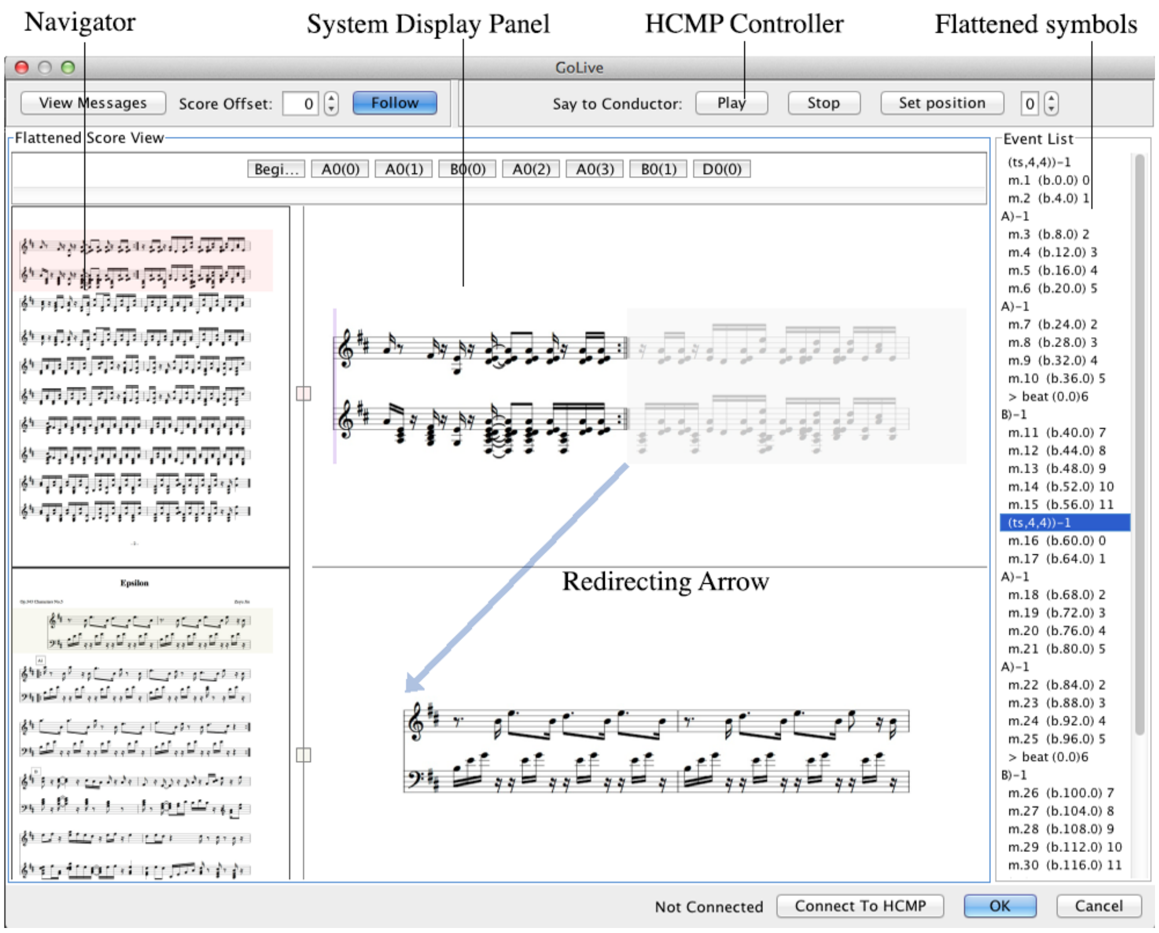


Figure B.2: Live Score Display: Arrangement Interface



```
(ts,4,4))-1
m.1  (b.0.0) 0
m.2  (b.4.0) 1
A)-1
m.3  (b.8.0) 2
m.4  (b.12.0) 3
m.5  (b.16.0) 4
m.6  (b.20.0) 5
A)-1
m.7  (b.24.0) 2
m.8  (b.28.0) 3
m.9  (b.32.0) 4
m.10  (b.36.0) 5
> beat (0.0)6
B)-1
m.11  (b.40.0) 7
m.12  (b.44.0) 8
m.13  (b.48.0) 9
m.14  (b.52.0) 10
m.15  (b.56.0) 11
(ts,4,4))-1
m.16  (b.60.0) 0
m.17  (b.64.0) 1
A)-1
m.18  (b.68.0) 2
m.19  (b.72.0) 3
m.20  (b.76.0) 4
m.21  (b.80.0) 5
A)-1
m.22  (b.84.0) 2
m.23  (b.88.0) 3
m.24  (b.92.0) 4
m.25  (b.96.0) 5
> beat (0.0)6
B)-1
m.26  (b.100.0) 7
m.27  (b.104.0) 8
m.28  (b.108.0) 9
m.29  (b.112.0) 10
m.30  (b.116.0) 11
```

Figure B.3: Live Score Display: Score Annotation Interface

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Prentice Hall, 2 edition edition, 2007.

[2] Willi Apel. Harvard Dictionary of Music. In *Harvard University Press*, page 696. Harvard University Press, 1970.

[3] Alfred W. Crosby. *The Measure of Reality: Quantification in Western Europe*. Cambridge University Press, 1997.

[4] Roger B Dannenberg. New Interfaces for Popular Music Performance. In *International Conference on New Interfaces for Musical Expression*, pages 130–135, New York University, New York, June 2005.

[5] Roger B Dannenberg. A Virtual Orchestra for Human-Computer Music Performance. In *Proceedings of the International Computer Music Conference 2011*, pages 185–188, San Francisco, August 2011.

[6] Tom Gerou and Linda Lusk. Repeat Signs. In *Essential Dictionary of Music Notation*, page 110. Alfred Pub Co, 1996.

[7] Nicolas E Gold and Roger B Dannenberg. A reference architecture and score representation for popular music human-computer music performance systems. In *International Conference on New Interfaces for Musical Expression*, pages 36–39, Oslo, June 2011.

[8] Michael Good. MusicXML for Notation and Analysis. In Walter B. Hewlett and Eleanor Selfridge-Field, editor, *The Virtual Score: Representation, Retrieval, Restoration,*, volume 12, pages 113–124. MIT Press, Cambridge, MA, 2001.

[9] Hal Leonard Corporation. *The Real Book: Sixth Edition*. Hal Leonard Corporation, 6th editio edition, 2004.

[10] John Hennessy, Norman Jouppi, Steven Przybylski, Christopher Rowen, Thomas Gross, Forest Baskett, and John Gill. MIPS: A microprocessor architecture. *ACM SIGMICRO Newsletter*, 13(4):17–17–22–22, December 1982.

[11] Walter B. Hewlett. MuseData : multipurpose representation. In *Beyond MIDI*, pages 402–447. MIT Press Cambridge, October 1997.

[12] Donald E Knuth. On the translation of languages from left to right. *Information and control*, 9:707–639, 1965.

[13] John R Levine, Tony Mason, and Doug Brown. *Lex & yacc*. O'Reilly Media, 2nd editio edition, October 1992.

[14] Marco Cantu. *Essential Pascal*. 2008.

[15] M Ochi. Music score display device. *US Patent 5,315,911*, 1994.

[16] Jukka Paakki. Attribute Grammar Paradigms Language Implementation - A High-Level Methodology in. *ACM Computing Surveys (CSUR)*, Volume 27:195–255, 1995.

[17] David Pager. A practical general method for constructing LR(k) parsers. *Acta Informatica*, 7(3):249–268, 1977.

[18] Vern Paxson, Will Estes, and John Millaway. Lexical Analysis With Flex, for Flex 2.5.37, 2012.

[19] Christopher Raphael and Jingya Wang. New Approaches to Optical Music Recognition. In *the 12th International Conference on Music*, number Ismir, pages 305–310, 2011.

[20] Gardner Read. *Music notation a manual of modern practice*. Allyn and Bacon, INC, Boston, 1964.

[21] Ana Rebelo, Ichiro Fujinaga, Filipe Paszkiewicz, Andre R. S. Marcal, Carlos Guedes, and Jaime S. Cardoso. Optical music recognition: state-of-the-art and open issues. *International Journal of Multimedia Information Retrieval*, 1(3):173–190, March 2012.

[22] Bill Schottstaedt. Common music notation. In *Beyond MIDI*, pages 217–221. MIT Press Cambridge, October 1997.

[23] Kurt Stone. *Music notation in the twentieth century: a practical guidebook*. W. W. norton & Company, 1980.

[24] Matthew Wright. Open Sound Control: an enabling technology for musical networking. *Organised Sound*, 10(03):193, November 2005.