Silas S. Brown

A Representation and Conversion System for Musical Notation

Computer Science Tripos, Part II

St John's College

April 2000

Proforma

Name	: **Silas S. Brown**
College	: **St John's College**
Project Title	: **A Representation and Conversion System for Musical Notation**
Examination	: **Computer Science Tripos Part II, June 2000**
Word Count	: **11024**
Project Originator	: Silas S. Brown
Supervisor	: Dr M. Richards

Original Aims of the Project

The project aims to design and implement a system for translating musical notation between different formats, including Braille. The system should be extensible in the formats that it can deal with and the attributes of music that it can process.

Work Completed

The system was designed and implemented in a data-driven manner, including the design of a scripting language. Tests were conducted and possible extensions were conceived.

Special Difficulties

I have partial sight and had to find how to unintrusively adapt my supervisor's system. It took approximately 10 weeks to obtain a Braille music manual and it proved nearly impossible to obtain documentation for national Braille music standards of different countries. In December my palmtop was immersed in water and some notes were lost (although most was backed up). The MuseData online repository was unavailable for some months toward the beginning of the project, due to a misconfiguration at Stanford University. I was hindered by bugs in gcc, flex and bison.

Declaration of Originality

I Silas Brown of St John's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

Acknowledgements

I would like to thank Hiro Omori of Wolfson College for her sighted assistance with figure 2.1, Saqib Shaikh of RNIB New College Worcester for his code and translation tables discussed in section 3.7, and Sue Williams for the use of her Braille embosser.

This project would not be the same without a number of discussions on various email mailing lists. I would therefore like to acknowledge the subscribers of Brent Hugh's Braille music mailing list for their comments and discussion of Braille music notation, especially Roger Firman, Fattah Douk, Karen Gearreald, Bettie Downing, Bettye Krolick, Tracy Carcione, Pete Donahue, Samuel O. Flores, Andrea Fridlund, John Sanfilippo (who provided a translation table from US ASCII Braille to dot patterns) and Piotr Prokop (who enlightened me on the situation in Poland). Thanks are also due to the subscribers of the MusiXTEX list at gmd.de for answering some questions about M-Tx, PMX and MusiXTEX encodings, particularly Christian Mondrup of Denmark, Christof Biebricher of Germany, and Don Simons. Also some subscribers of the Association of C and C++ Users' programming questions list answered some questions about the finer points of the new C++ libraries; these people included Steve Love, Francis Glassborow, Kevlin Henney and Dave Rutlidge. Various people on `ucam.comp.linux` also helped by answering some questions about TEX typesetting while I was preparing this dissertation.

All trademarks in this dissertation are hereby acknowledged.

Chapter 1

Introduction

Music has been written down for centuries, but it was not until the twentieth century that the speed and availability of computers opened up new possibilities. Music publishing software has automated such tedious tasks as typesetting, transposition, and the writing out of parts; corrections can be applied seamlessly; and musicologists can make complex computer-aided searches through large online repositories such as MuseData[13]. All of this requires that the music is somehow stored in computer memory, and various ways of doing this are in use, including:

- Recording a performance digitally[18];
- Logging the events of an electronic performance, such as keyboard keys being pressed (a MIDI file[14] is such a log)[1];
- Storing the notation as a set of symbols and their positions on the page (some publishing software, including Sibelius[9], does this at least to an extent);
- Storing the notation in a way that does not rely on positioning information, usually by inventing a small computer language that can be used to express the music in a textual manner. This is a common method for music publishing software (including DARMS[27], Score[29], Finale[8], GNU LilyPond[6], PMS[12], MUP[4], ABC[33], MusiXTEX[31], and my own program Manuscript Writer[2]), and for musicology research.

These are listed in decreasing order of user control (over the final performance), and in increasing order of automation. The latter method allows high-level pro-

[1] In fact the earliest form of performance log pre-dates electronic computers — the pianola was invented in 1896.

cessing of the music as a logical entity, without the software having to interpret timing or positioning information.

One possibility that has been relatively little explored is that of using this malleability of the textual representations to put them to a use that is different from the original intention. For example, an input file written for a music publishing system might be converted to a form suitable for a different music publishing system, for a musicology research system, or for Braille transcription (using a computer-controlled Braille embosser).

The prospect of producing Braille music automatically is particularly exciting, since there are severe shortages of it. The most famous existing program is Goodfeel[22], which converts NIFF files to American Braille music, but there is not yet a large enough repository of NIFF files for Goodfeel to solve the Braille music shortage problem, and the American Braille music standard is not used worldwide.

There are many variations of Braille music across the world, and standards change over time. To facilitate the exchange of Braille music between countries, the Braille Music Subcommittee of the World Blind Union have agreed on a set of signs to be used internationally[24], but not everyone is familiar with these and the standard is priced quite highly in some countries. Also, the agreement is not yet total, and it is limited in scope to individual signs rather than overall layouts; hence Braille music continues to be diverse. Even if a complete international standard can be universally accepted and learned, there can still be individuals (especially teachers) who invent their own "house styles".

As well as there being a large number of Braille representations and textual representations of music, music itself contains a large number of possible attributes (things it can say about the notes), which modern composers are still adding to. Eleanor Selfridge-Field points out in [25] that attempts at interchange codes often fail due to their inability to represent some particular musical attribute that the authors hadn't thought of.

Given the above issues, what is needed is a conversion system that is *extensible* in the input formats it can take, the output formats it can give, and the musical attributes it can process. This project aims to implement such a system.

Chapter 2

Preparation

2.1 Requirements Analysis

The system needs to convert between formats of musical notation in a way that is extensible to new formats and new musical attributes. Among the output formats should be Braille music. It should be implemented in software for ease of distribution, and it should be able to complete the translation of a short piece of music in reasonable time on an average desktop computer (but efficiency is not the main concern).

Any extensible system must strike a balance between power and simplicity. Power is the versatility of extensions that can be written; simplicity is how easy they are to write. The limit of power is to ask the user to write the entire program, but that gives little simplicity; the limit of simplicity is to forbid any extensions, but that gives little power. Between these limits lies a complex non-linear relationship that depends on design decisions. These should make the system powerful enough to serve its purpose, but simple enough to justify its use as a tool.

2.1.1 Braille Output

The requirements for Braille output need particular attention, since it is not as authoritatively defined as input files for computer programs are. A text file for a music program either works or doesn't, but Braille is read by humans, who have numerous different preferences about how things should be written. This is of course the point of customisability, but there are some things that can be noted at this stage:

Braille is normally embossed as raised dots, so the system should be able to output a file that is readable by a computer-controlled Braille embossing machine ("embosser"). It should also be able to present it in a visual form suitable for inclusion in this dissertation, perhaps with annotations (useful in testing[1]).

The New International Manual of Braille Music Notation[24] is very concerned that Braille versions of printed music follow the printed edition exactly. For example, the Braille might include indications of which page and line number on the printed edition a given bar appears on, or exactly how the beamed notes are grouped together in a particular edition, or what the printed pitch is rather than the sounding pitch when "8va" is used. Such typesetting decisions are often made automatically by music publishing software, so they will vary according to which program was used to typeset the printed music. If the conversion system is expected to read the input of these typesetting programs and write out the Braille, then the only way that it can reproduce the print exactly is to emulate the algorithms of all typesetting programs, which is not a feasible thing to do in the timespan of this project. However, in my personal experience from RNIB New College Worcester, such meticulous attention to details from the printed edition is completely unnecessary (and sometimes even annoying) in the context of performance or teaching blind students; it seems to be only required by advanced score students and blind teachers of sighted students. Therefore, a program that does not reproduce every single typesetting decision will still be useful most of the time.

For this reason, it was decided that such typesetting information should only be included if it is explicitly stated in the input format (so people who need it have to choose an input format that states it), and then only if the user wishes it (ie. the Braille output code has been configured to include it).

2.2 Design Principles

Object orientation is appropriate for most of the project because of the amount of data processing involved and because inheritance and loose coupling make extensibility more natural (components of the system can be extended without disturbing the rest of it). Encapsulation, type checking and the Least Privilege Principle (that components of the system should be restricted from accessing what they do not need) also helps to detect coding errors. OOP is not appropriate for every component, however; lexers and parsers, for example, can be done more simply with tools like `lex` and `yacc` (or `flex` and `bison`), so not all of the project was written as classes (although encapsulation still applies).

[1] Annotated Braille might help sighted musicians to learn Braille music, but this is not a requirement.

Object orientation tends to lead to a design process that is a combination of the waterfall and spiral models; a number of prototypes were involved at the design stage.

The concept of an extensible system leads to a design philosophy of going from the specific to the general — instead of solving a specific problem, try to find a more general class of problems of which this is an instance, and solve that. This is not the same as retro-fitting generalisations to an existing *solution*, which can lead to a conceptually unclear structure; instead the generalisation is done at design time.

Because of this "generalisation" philosophy, an early decision was that **the core of the system shall not know anything about music**. All specification of music is done by the scripts and data; the rest of the system simply provides abstract concepts on which this can easily be built. Although the design of these concepts is driven by the needs of music, there should be as little special-case code as possible, which increases the likelihood that some new (or previously unsupported) attribute of music can be implemented using existing features.

2.3 Research

The MuseData repository[13] was found to be an extensive online source of musical data and Selfridge-Field[26] was a good overview of some existing musical codes. The Notation Interchange File Format (NIFF[11]) provided a little inspiration for the tuple space (section 2.5), but only a little (namely, the idea of holding data as an unsorted collection of facts — NIFF "chunks" — and that of using rational numbers); other ideas in NIFF don't fit this system's design philosophy.

Braille music had to be researched, since I do not read it myself. There is an overview in [10] but I needed more extensive documentation from which to derive data suitable for driving the program. It was helpful to join a mailing list[5] and in time I obtained the New International Manual[24], but this describes only one set of signs and does not describe overall layouts. English documentation of other standards is scarce.

Since the New International Manual is written by and intended for blind musicians rather than computer scientists, it often relies on examples with scant explanation rather than on formal or informal specifications. Using it involved an amount of guesswork, discussion with the mailing list, and reliance on the fact that the user can customise it.

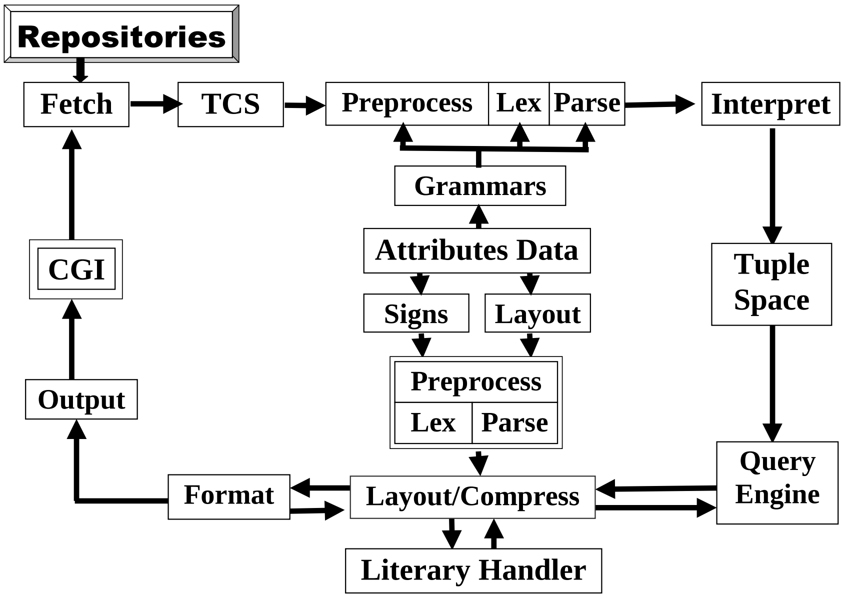

Figure 2.1: Overall structure of the project

2.4 Overall Structure of the Project

The overall structure of the project is shown in figure 2.1, with the arrows representing data flow. The reasoning behind each component is described below.

2.4.1 Tuple Space and Attributes Data

The internal method of storing the music is a modified version of the "tuple space" paradigm used by the parallel programming language Linda[16]. Each tuple is a collection of attributes and their values, and usually corresponds with an event in the music. Although each tuple has a type, "type" is just another attribute. The semantics of different attributes are kept track of in an "attributes database", which must be referred to when writing scripts that add to or use the tuple space; this ensures consistency. The actual attributes database used is given in Appendix A but it can of course be extended.

The tuple space is further discussed in section 2.5.

2.4.2 Grammars, Parsing and Interpreting

Input is read in to the tuple space by preprocessing, lexing and parsing it; the output of the parser is code in a simple interpreted language that is used to build up the tuple space. Parser generator scripts are used for the input parsers to maintain extensibility; the use of a textual interpreted language[2] means that it is not essential to compile them in with the rest of the system.

The parser generator scripts are themselves preprocessed so that the grammar may be specified in a simpler way (the full power of bison with arbitrary C code is not necessarily needed) — a grammar file is like the main part of a bison file except that anything in double quotes represents something to be output and anything in single quotes represents an input token (usually one character in music languages). An example grammar file for a simple music language is shown in figure 2.2; grammars for real music languages are more complicated (in order to be usable, the system has to parse the entire language even if it does not implement all of it).

The language in the quotes is a rudimentary easy-to-implement postfix notation; this is all that is necessary. The commands available are given in figure 2.3.

The interaction between preprocessor, lexer and parser is not always as straightforward with music languages as it is with conventional compilers. Macro definitions can be part of the main grammar (and hard to find by simple preprocessing), and it can be hard to tell at the lexer level what is a string literal and what should be tokenised (because string literals are not usually quoted). For this reason, much is done at the parser level, so the rules are occasionally cumbersome. Macro definitions require feedback to the lexer, which was not implemented (none of the attempted input languages required it).

2.4.3 Querying and Layout/Compress

The code to convert the tuple space into the output format is also script driven, using signs (symbols) and layout information. The layout language is imperative and is further discussed in section 2.6. It is possible that the output will then be compressed if the language allows for this (Braille music has some shorthands for avoiding repetition); this was not implemented due to difficulties in automating it (it required "musical sense").

[2]In the first prototype, the parser output was a sequence of instruction objects with arbitrary `execute()` methods, and it was possible to insert objects back upstream. This proved unnecessary and unwieldy.

```
/* -*- C -*- (for emacs) */

/* @GRAMMAR-NAME: Simple language to demonstrate grammar files */

%%

Start:
     /* Set some initial values */
     "!set octave 2 permanent !set notatedLength 1 permanent \
!set time 0 permanent" Commands;

Commands: /* epsilon */ | Commands Command;
Command: Note | OctaveChange | LenChange;

Note: NoteLetter OptAcc "!set realLength !get notatedLength t2 !add \
!wipe transient !set time !+ !get time !get realLength permanent !wipe t2";

NoteLetter: 'A' "!set letter a transient"
          | 'B' "!set letter b transient"
          | 'C' "!set letter c transient"
          | 'D' "!set letter d transient"
          | 'E' "!set letter e transient"
          | 'F' "!set letter f transient"
          | 'G' "!set letter g transient";
OptAcc: /* epsilon */
          | Sharp "!set accidental sharp transient"
          | Flat  "!set accidental flat transient";
Sharp: '+' | '#';
Flat: '-';
OctaveChange: '<' "!set octave !- !get octave 1 permanent"
            | '>' "!set octave !+ !get octave 1 permanent";
LenChange: 'L' "!set notatedLength !/" UnsignedInt "4 permanent";

Integer: UnsignedInt | '-' "!negate" UnsignedInt;
UnsignedInt: "!concat" Digit RestOfUnsignedInt;
RestOfUnsignedInt: /* epsilon */ "!empty" | UnsignedInt;
Digit: '0' "0"; Digit: '1' "1"; Digit: '2' "2"; Digit: '3' "3";
Digit: '4' "4"; Digit: '5' "5"; Digit: '6' "6"; Digit: '7' "7";
Digit: '8' "8"; Digit: '9' "9";
```

Figure 2.2: Grammar file for a simple music language

!set attribute value colour — assignment. “Colour” can be used for temporary assignment (eg. accidentals with limited scope); **!wipe** colour restores the previous value of all assignments of that colour (eg. after a barline).
!get attribute — refer to a previous assignment
!add — copy the entire set of assignments into a tuple and add it to the tuple space (discarding colour). Irrelevant attributes don't matter. Attributes beginning “x-” are used for temporary storage of values internal to the input format and need not be copied to tuple space.
!addif expr — do **!add** *if* the parameter evaluates as true (non-0); used for conditionally adding tuples
!savestate name, **!loadstate** name — used for preserving the entire state (eg. the starting state) for later
!usestate name — like **!loadstate** but does not take a copy (changes get written to the saved state)
!concat — concatenates its two arguments
Arithmetic operators **+**, **-**, *****, **/** and **!negate**
!test — !test x y z is like x?y:z in C (but everything is evaluated). Operators **<**, **>**, **<=**, **>=**, **==**, **!=**, **!strcmp**, **!and**, **!or** and **!not** are available. **!seq** executes both its arguments.
!empty returns the empty string, **!!** the character ! and **!space** a space. Quoting (and escaping quotes with \) is also possible.

Figure 2.3: Commands in the rudimentary postfix language

2.4.4 Formatting and Outputting

Output usually needs to be formatted to the page dimensions of, and then converted to the form necessary for, the output device, whether that is a Braille embosser, a means of visually presenting Braille, a text file, or something else.

It is conceivable that there will be feedback from the formatter to the layout engine, since the placing of line breaks can affect the symbols output — there are often rules about what should be done around a line break in a given context, particularly if a bar is being split across two lines. This in turn affects the line lengths and may alter the positioning of line breaks, and so on. This could be solved by using "branch and bound" at each possible line break, forcing the symbols to fit that break and evaluating the result[3]. It was not implemented due to its low priority (breaking the lines at points that do not require feedback is adequate in all but extreme cases).

2.4.5 Interface and Repositories

The system is command-line driven, and can be operated as an online service (hence "CGI" in figure 2.1) with a suitable wrapper script. In the case of MuseData, the input needs to be fetched from an online repository; this usually involves fetching and concatenating several files, since MuseData encodes each part in a separate file.

2.4.6 TCS and the Literary Handler

Most music contains text — titles and other headings, instrument names, directions, lyrics and so forth. This text needs special handling. If the output is textual then there will be some way of distinguishing between text that represents music and text that is literary, and it may be necessary to deal with certain characters differently to ensure that this is honoured (eg. escaping quote characters). If the output is Braille then a text-to-Braille translation must take place, which can be complex.

Further, the text can be in any language, including languages with accents and languages not based on the Latin alphabet. The input file will be in a character set that is not necessarily ASCII, and the output may need different rules applied to it depending on language (especially for Braille).

[3]Fuzzy logic could be used for this. If the fuzzy rules are good enough then it may be possible to evaluate linearly without backtracking (break whenever the centroid exceeds a threshold).

For this reason, the literary handler is an encapsulated component in itself. Different literary handlers may be used depending on the output language[4]. The literary handler will have an expected input character set or encoding[5]; a utility like TCS (translate character set) can be used to change the encoding of the text in the input file if necessary to suit the literary handler, and the rest of the system need not be concerned with character sets, so long as it carries bytes verbatim — strings should not be null-terminated, for example (the C++ string class is adequate).

2.5 The Tuple Space

As outlined in section 2.4.1, the music is stored internally in tuple space. This is held in main memory, which is fast and usually adequate (virtual memory can be used if it isn't).

The data types of attributes and values can in theory be anything, but although the implementation uses templates to make this extension easy, only strings are used, because the textual scripting languages use strings. This forfeits some type safety but does simplify matters.

Strings can of course represent other data, such as numerical values; all arithmetic is done using rationals (integers or vulgar fractions of the form "N/D") — rationals allow arbitrary division of time without introducing the rounding errors that MIDI files are prone to.

It is not necessary for tuples to have pointers to each other, since any relationships (such as "occurs after") can be expressed by assigning values to dimensional attributes (such as "time" or "part" or "bar"). This has the advantage that the relationships can be addressed arbitrarily just by sorting on attributes (rather than by following structures or by searching a high-dimensional array), and if a particular relationship is unimportant for a particular purpose then it can easily be ignored. The disadvantage is that memory consumption is far from optimal (there is a lot of redundant data in shared attributes, which would not be alleviated by pointers because the size of the pointer is often comparable with the size of the data), but it is manageable (the memory consumption grows linearly with the number of tuples).

Allowing sorting by any attribute raises the issue of how to deal with tuples that do not have that attribute. The simplest thing is to insist that *every* at-

[4]More than one language in the same piece of music is common, but the main language's transcription method is normally used consistently even for foreign words.

[5]It is a mistake to insist on a single character set for everything, since there is no perfect character set (eg. Unicode fails to support a large number of Indic and Taiwanese characters).

tribute *must* have a value in *every* tuple (even if it is nonsensical), or at least provide defaults (such as empty).

2.5.1 Tuple or Attribute?

A given aspect of music might be interpreted as an event (or tuple) in itself, or as an attribute of another tuple. A dynamic, for example, may be either, but it's better to have it as a separate tuple in order to cope with dynamics that don't start exactly under notes. A tie could be an attribute of one or both of its notes or a separate tuple. A chord could be one tuple or several, and its pitches could be absolute or relative (like Braille interval signs).

In a sense it does not matter which method is chosen, so long as it is adequate and consistent. This is why the attributes database was used; it leads to a reasonable solution without spending inordinate lengths of time on musicology during the preparation.

The way attributes are represented also needs to be made clear, since musical terminology differs across formats and across countries — even something as deceptively simple as a note letter (A–G) is not universal; the French use "do ré mi", the Germans use H for B♭, and so on.

2.5.2 Propagation of Implicitness

Some attributes (such as accidentals) can have implicit effects further down the music. Sometimes this implicitness is optional, that is, it can be made *ex*plicit if the editor wants to draw attention to it. Some input formats (such as Manuscript Writer) can express this optional explicitness (so a flat will be printed whenever it occurs in the input, regardless of whether or not it is needed); other formats require all accidentals to be explicitly stated and the typesetting program decides which ones to print.

Since conversion should ideally preserve as much as possible of what was in the input file, editorial reminders should be preserved. Thus the *explicitness* of an attribute like an accidental needs to be preserved — if it is explicit *and could have been implicit*, then the output should (if possible) also express it explicitly. This is done by giving different attributes to explicit and implicit things, and the input format parser should take care of which one (or both) to set.

2.5.3 Time

Representing time and duration using rational numbers prevents rounding problems as stated above. It also eases coping with multiple simultaneous time signatures, by setting the unit of time to *one bar*[6] which is then divided. Nested tuplets are also not a problem (simply multiply the fractions). However, this data alone is not enough for notation — given a note that lasts $\frac{a}{b}$ of a bar, you have to know the time signature and the details of any tuplets before you know which note value to write.

For this reason, the attributes database defines two separate length attributes — "notated length" and "real length". The unit of notated length is the *crotchet* and it describes what note (or rest) value to use (independently of context); the unit of real length is the *bar* and it gives the real length for synchronisation purposes. Any additional typesetting information about tuplets is given separately.

2.6 Design of the Formatting Language

The formatting language is used to specify how the tuple space is converted to the output. It is a textual language with reserved words and identifiers. A problem with reserved words is that the introduction of new ones (or the translation of the lexer to another human language) can result in name clashes with existing code, but attempts at avoiding this risk led to a language that was much less intuitive. The language was designed by defining an output format in an intuitive way and observing the result.

Formatting code is written in plain text with arbitrary layout (ie. arbitrary amounts of whitespace and line breaks between the words, so legible indentation is possible). C++ style '//' comments are supported, and a macro preprocessor is used. A file of formatting code consists of any number of <u>format</u>, <u>procedure</u> or <u>sign</u> constructs, in any order. A <u>format</u> construct contains imperative code for the complete output process. The <u>sign</u> constructs provide data on the individual symbols to be output; most of the detail is provided by <u>sign</u> constructs, with a <u>format</u> controlling the overall layout.

More than one <u>format</u> can be specified; the <u>procedure</u> construct can be used for "service routines" for several similar formats. In Braille, for example, the individual bars in bar-over-bar format are practically identical to those in section-by-section format; the formats differ only in how the complete bars are arranged on the page. It therefore makes sense to define a procedure for outputting one bar and use it in both formats.

[6] If the barlines are not synchronous then some other unit length of time that can stand for a bar has to be used. Asynchronous barlines (and multiple simultaneous time signatures) are not always *avante-garde* — they are used in the minuet from Mozart's opera *Don Giovanni*, for example.

The syntax of the language is described below. Reserved words (which are case insensitive) are underlined in this discussion. A reserved "word" that includes whitespace (such as "end format") is only recognised as reserved in its entirety; whitespace is included for legibility (especially on speech synthesisers for blind people).

format "name" ... end format

Encapsulates the layout code for an output format[7]. "Name" is presented to the user during format selection, and may be an arbitrary string.

procedure identifier ... end procedure

Encapsulates code in a procedure.

call identifier

Executes the named procedure (and then returns to this point).

foreach identifier [where criteria] [local identifier-list] ... end foreach identifier

At any time during the execution of the code, a subset of the tuple space (initially all of it) is visible. The foreach construct is used to divide and subdivide the set of visible tuples, for example:

```
foreach part
foreach bar
// do something
end foreach bar
end foreach part
```

Notice that the end of a construct must match with its beginning; this catches mistakes, and makes the code more legible, especially on speech synthesisers (numerous consecutive "end" statements could be muddling).

What happens is that the system classifies all the currently-visible tuples according to the attribute specified by 'identifier'; tuples that have the same value of that attribute are grouped together. For each of these groups, in order

[7] The actual implementation uses "end format2" rather than "end format". This is because flex produced a lexer that didn't work, and neither myself nor my supervisor found a fault in my lex file, so we suspected a bug in flex. Adding a "2" somehow worked around it.

of increasing value[8], the code within the foreach construct is executed "on" that group, that is, the group is temporarily made to be the visible subset of the tuple space[9].

The optional 'where criteria' gives a boolean expression that is applied to each tuple as they are grouped; if this expression is false then the tuple is excluded from the operation. Thus "foreach offset where type is text" will take all (visible) tuples matching the criterion "type is text" and group them by attribute 'offset'. The boolean expression can be "attribute is value" or "attribute isnot value"; for numerical values, operators is>, is<, is>= and is<= are also available. Multiple criteria can be combined using the logical operators and and or; and has higher precedence but parentheses can be used.

Equality is denoted by is because there is also a was operator (along with wasnot, was>, was<, was>= and was<=), which applies to the value of that attribute in the last tuple that had it checked by a boolean expression[10]. This is most useful for criteria in sign statements (below), where one can say, for example, that the condition for a "set length to 2" symbol is "length is 2 and length wasnot 2" (the initial state is that an attribute wasnot anything).

The optional 'local identifier-list' gives a list of attributes (separated by whitespace) for which the tuple groups made by foreach should be independent for the was operator. That is, the following example creates an independent 'octave' for each 'part', and if something at the start of a bar referred to what the octave was then it would get the previous bar's value *in the same part*[11].

```
foreach bar
foreach part local octave
```

[8]Comparing numerically if possible (using fractions), otherwise lexicographically. It did not prove necessary to support arbitrary user-specified comparison functions, but the implementation is such that these can easily be added.

[9]On a supercomputer, the groups could conceivably be processed in parallel (and outputs subsequently collated), since there are rarely dependencies between them. However, a job large enough to warrant this is likely to consist of many pieces, and it would be more efficient to parallelise at *that* level (perhaps using distributed processing).

[10]In an expression of the form "A and B" where A is known to be false, or "A or B" where A is known to be true, B need not be evaluated because the final result is already known, so it becomes debatable whether or not the attributes mentioned in B should be counted as "checked". Since A could be a guard condition on whether B makes sense (eg. "type is note and octave is 3"), the design states that expressions are evaluated left-to-right and that attributes not evaluated are not counted as checked.

[11]Wrapping a "foreach part" within a "foreach bar" in this way assumes that the barlines are synchronised in all parts, which is not necessarily the case. However, if the output format cannot express asynchronous barlines then it is reasonable to assume that they are synchronised, at the risk of getting a mess when they're not.

// do something
end foreach part
end foreach bar

If there were more than one "foreach part local octave" construct, then they would refer to the same set of octaves. The initial value wasnot anything in each case.

foreach where criteria . . . end foreach

This variant of the foreach construct executes the code *for each individual tuple* that matches the criteria. The set of visible tuples is set to that one tuple in each case (local would make little sense here). This construct is needed infrequently, since in most cases have and sign are less cumbersome.

block [centre] . . . end block

A *block* is the basic unit of formatting[12]. Blocks are arranged in lines and pages like a wordprocessor arranges words, but blocks can be several lines high. Many music formats treat each bar as a block, with each part on its own line within it; there are also other uses for blocks, such as for titles, hence "centre"[13] (other formatting directives are conceivable).

This construct creates a block, to which the code within it may write. Any code that writes output *must* be within a block construct. Blocks are output in order of creation. It would make little sense for blocks to be nested.

sign appearance, identifier [, identifier [, criteria]]

This is not code, but is a top-level declaration of a sign that is used by the code. Sign statements occur outside format and procedure constructs. You cannot *in-line* signs in the code, since doing so would hinder customisability; addressing a sign by name is more intuitive and allows re-writing the sign table (sign statements can easily form a table — see Appendix B. The sign statements for Braille were originally typed into a spreadsheet).

The appearance of the sign in the final output is controlled by the first parameter. If the output is a text-based format then this is a quoted string; if it is Braille then this is a list of one or more dot patterns separated by whitespace (a

[12] To prevent confusion with these "blocks" and "blocks of code", this discussion calls "blocks of code" *constructs*.

[13] American spelling is also acceptable.

dot pattern is specified with digits, so 136 is ⠥ and 136 2 is ⠥ ⠂)[14] — 0 represents an empty cell. It is conceivable that other ways of specifying appearance can be added, although text is sufficient because it can be post-processed by a text-driven typesetting system.

The first identifier names the sign (eg. "sharp"), and the second, if present, names its category (eg. "accidental"). Both of these can be used to annotate the output, but the primary purpose is for reference by the code.

The criteria expression, if present, is used by the have instruction (below) to determine whether or not to output each sign for any given tuple. For example, the criteria for "sharp" might be "type is note and accidental is sharp"[15] (the condition on "type" is so that meaningless attributes are ignored). A sizable database of signs can be built up in this manner without having to write much code, since have handles entire categories at once; if, for example, there is an instruction "have ornament" then adding a new type of ornament is simply a matter of adding a new sign statement with category ornament and appropriate criteria.

have identifier-list

This takes a list of categories separated by whitespace. For each visible tuple, it goes through the categories in the given order, and for each category it evaluates the criteria of all the signs of that category, outputting any signs with criteria that are true for that tuple. Thus "have note accidental" will output zero or more signs from category "note", followed by zero or more from category "accidental", for each tuple. The core of an output script will often be a long have command like "have lenprefix bowing ornament articulation accidental octave note lendots tie fermata breath"; hence the order is easy to customise.

---

[14] A six-dot Braille cell has three rows and two columns; the dots in the first row are conventionally numbered 1, 2 and 3 and those in the second row 4, 5 and 6. An eight-dot Braille cell (sometimes used in America) additionally has dots 7 and 8 below, thus:

1	4
2	5
3	6
7	8

The implementation can handle cells with *any* number of dots in case more are added in future, but the specification of dot patterns would have to change (and so would the protocols of embossers).

[15] Notice, though, that there are two different namespaces — the sign category "accidental" need not match the tuple attribute "accidental", and the sign name "sharp" need not match the tuple value "sharp". This has to be the case otherwise the signs would be too strongly limited by the tuple space.

The order of the *tuples* that have reads is undefined unless given by foreach (eg. "foreach time") and/or by using different have commands to pick out different tuples — "have barline have note" will output barlines before notes but "have barline note" might get the tuples in the wrong order. (However, "have note accidental" *is* better than "have note have accidental" because, if there is more than one note, the latter will output all the notes first, followed by all their accidentals.)

The order of signs within a category is also undefined (in most cases there will be at most one); new categories should be created where order matters.

The special criterion first time can be used in an expression to mean "this is the first time that this point in this expression has been reached in this particular execution of the have command" (not first time is the opposite). For example, "type is note and not first time" will be true for all notes in a chord except the first; this can be useful if the output needs them to be treated differently.

output identifier-list

This takes a list of sign names (not categories) separated by whitespace, and unconditionally outputs them, in order.

literary identifier

This gets the value of the attribute specified by identifier, passes its text to the literary handler, and outputs the result. It is used for outputting such things as musical directions and numbers (but see digits below). The tuple that is chosen to get the value from is undefined; it is expected either that there will only be one tuple visible (because of foreach where), or that all the visible tuples will share the same value for that attribute (because of foreach).

There is also a version of literary that takes a quoted string of text, mainly indented for "Printed by..." messages.

digits identifier identifier-list

This is like literary, but outputs the integer value of the attribute specified by the first identifier, using digits specified by the other identifiers, which are sign names separated by whitespace, the first sign standing for 0 and the last for 9.[16] This is to bypass the literary handler, which might otherwise do something unwanted to the number. For example, Braille numbers are preceded by ⠼, which is correct

[16] It did not prove necessary to support radices other than 10, but this could be done by varying the length of the list.

behaviour in text (such as titles) but not always wanted in the music. Braille music also has a *different* set of digits called "lower digits" for time signatures; these can be handled by specifying appropriate signs in digits.

The justification for bypassing the literary handler is that *these* numbers are not "literary" (they are not text embedded in the music). They are defined by the *music* format, not the *literary* format.

nextline

Creates a new line within the current output block. Blocks initially have one line; nextline adds a line at the bottom and moves the output cursor to its beginning. Lines can be left blank.

When outside a block, nextline causes the previous block to be the last on its line.

pad identifier

Sets the current block's "pad character" to the sign named by 'identifier'. This will be the sign used to "pad out" all the lines in the block (from the current line onward) to the length of the longest. By default it is the "empty sign" for the format in use (eg. an empty Braille cell, or an ASCII space). The width of the sign should be 1, since otherwise the space might not divide evenly[17]. Pad can be used for "tracking dots" in some variants of Braille.

set identifier value

The was operator normally gets an attribute's value from the last tuple that had it checked. Set can override this; it sets the value of the given attribute as seen by was. This can be used to set the initial state, and to change state where necessary (eg. if an octave mark is normally output only when the octave changes, but the note after some text *must* have an octave mark, then set 'octave' to 'nonsense' after writing the text). When used in conjunction with local, set must be within a similar scope to take effect, eg.

```
foreach part local notatedLength
set notatedLength 1
end foreach part
...
```

[17] It is possible to support wider pad signs and switch to an alternative sign when the remaining space becomes too narrow, but this proved unnecessary.

<u>foreach</u> part <u>local</u> notatedLength
// do something
<u>end foreach</u> part

This avoids the confusion that might result from being able to <u>set</u> something that is later transparently copied to the locals.

2.7 Languages and Tools

A number of tools were used in implementing the system and for project maintenance; these are discussed under Implementation (section 3.1). Becoming familiar with them was part of the preparation. Also, since I had learned "classic C++", I wished to become more adept at "modern" C++, including the STL (Standard Template Library), string and stream handling, and so forth. While my aim was not to be a C++ "absolute purist", I tried to write readable, maintainable and generally "clean" code.

Chapter 3

Implementation

3.1 Languages and Tools

3.1.1 C++

Most of the project is written in C++, for the following reasons:

- C++ provides object orientation. Given that the project is primarily concerned with manipulating data structures, the paradigms of classes, methods, inheritance and so forth are highly appropriate.
- C++ provides the Standard Template Library (STL) of pre-written container classes and algorithms.
- C++ provides encapsulation and helps enforce the "least privilege principle" — the idea that code should be constrained (using type checking, `const`, `private` and so on) in order to reduce the likelihood of undetected error.
- C++ provides transparency — operator overloading allows C++ objects to be treated like primitive types, and also allows homogeneous integration with existing facilities such as stream I/O and algorithms in the STL. **Operator overloading can be abused** but this does not rule out its use altogether; if the operation on the class is conceptually the same as an operation on a primitive type, then the use of operator overloading can improve code readability.

Java lacks the conciseness and transparency (and often the efficiency) of C++, and currently appears to be less stable in design and implementation. It also requires the end-user to have a Java environment. Python[32] is an object-oriented "scripting" language that is gaining increasing popularity; it has features similar to Java. Having investigated it I found that there was no incentive for learning it in preference to using C++.

I used the latest version of the GNU C++ compiler, which is mostly conformant to the standard.

Templates and their Alternatives

C++ templates enable code to be written for and applied to a whole class of types, thus solving a whole class of problems simultaneously. They are akin to parameterised types in ML. Another advantage of templates is that they enable code to be written in more abstract terms (eg. a "database" can hold a "datum", whatever a "datum" is) and this helps in encapsulation, therefore aiding in design by prototype.

Templates are not the only way to accomplish this. In Java, one could use interfaces, and in C++ without templates, one could use multiple inheritance. Conciseness and efficiency will often be lost with these alternatives, particularly when instantiating a template class with a primitive (which can be useful for testing). Templates can of course be instantiated to these alternatives if desired.

Templates can also offer more type safety than inheritance and interfaces. This is because the latter will allow the inadvertent mixing of heterogeneous types when templates would not. For example, if a template class `C` is instantiated to classes `A` and `B`, then calling an instance of `C` instantiated to `A`, using an object of type `B` instead of type `A`, will generate a compiler error. If, on the other hand, `C` accepts objects of type `S*`, where `S` is a superclass of `A` and `B` (or a Java interface implemented by both of them), then the error will be at runtime and may go undetected.

If a template class is only going to be instantiated once, then a simple use of `typedef` is an alternative to the template. However, in an extensible system it might be instantiated more times than the designer anticipated.

The main disadvantage of templates is the compiler errors that occur when things are wrong. Template code cannot be fully error-checked until it is instantiated, and any error messages that do occur will often be several kilobytes long, due to the inclusion of all the type and instantiation information in the error message. I usually deal with such unreadable error messages by simply looking at the line numbers and working out for myself what is wrong, although this is not always successful; another technique is to load the error message into emacs and do a series of search and replace operations on it in order to make it more readable.

```
Main: S {formatlangParsedScript=$1;};
S: /* epsilon */ {$$=makeNewScript();};
S: S PROC ID Code ENDPROC {$$=addProcedure($1,$3.data,$3.length,$4);};
S: S FORMAT String Code ENDFORMAT {$$=addFormat($1,$3.data,$3.length,$4);};
/* S: S SignData - done below */

Code: /* epsilon */ {$$=makeEmptyStatementList();}
           | Code Statement {$$=appendStatementToList($2,$1);};

Statement: Call | FOREACH Foreach {$$=$2;} | Block | Pad | Nextline
           | HAVE CategoryList {$$=$2;} | OUTPUT SignList {$$=$2;} | Literary
   | Digits | SET ID ID {$$=makeSet($2.data,$2.length,$3.data,$3.length);};
Call: CALL ID {$$=makeCall($2.data,$2.length);};

Foreach: ID OptWhereExp OptLocalList Code ENDFOREACH ID {
  // Check that the two IDs are equal:
  if($1.length!=$6.length || memcmp($1.data,$6.data,$1.length)) {
    yyerror("'Foreach' mismatch"); YYERROR;
  } else $$=makeForeach1($1.data,$1.length,$2,$3,$4);
};
Foreach: WhereExp Code ENDFOREACH {$$=makeForeach2($1,$2);};
OptWhereExp: /* epsilon */ {$$=NULL;} | WhereExp;
WhereExp: WHERE Criteria {$$=$2;};
OptLocalList: /* epsilon */ {$$=NULL;} | LOCAL LocalList {$$=$2;};
```

Figure 3.1: Part of the Bison file for the layout script parser

Templates can also make for slow compilation and large output code, depending on the compiler. Occasionally the compiler reports "internal compiler error" on error, which is not helpful.

3.1.2 Flex and Bison

Bison was used to generate parsers for the input files, and flex and bison were used to generate a lexer and parser for the layout script. The input lexer tracks line and column numbers for error reporting.

The bison "prefix" option was used to ensure that several parsers can be linked into the same executable without name conflicts. The Makefile can also generate standalone parsers. The input parsers use a global `FILE*` variable to output the postfix language; the layout script parser calls functions to generate a syntax tree, as shown in figure 3.1 (the functions are wrappers that construct C++ classes, but the bison output itself needs to be compiled as C).

3.1.3 Awk

An Awk script was used to generate the Bison input files from the grammar files, so that the full complexity of Bison need not be exposed (figure 2.2).

3.1.4 Make

The project's Makefile is complex. Dependencies are automatically updated with `g++ -MM` and stored in `.d` files; this idea was loosely based on an example in the `make` documentation, but I had to modify it to handle subdirectories. Because sed unfortunately uses the slash (/) character to separate regular expressions, and the slash character is also used as a directory separator in the Unix file system, a lot of `sed` work had to be done in ensuring that the makefile was subdirectory-safe, and this together with backslash escaping leads to `sed` expressions like

```
sed -e ":l s/\\\\//*/
tl :m s/*/\\\\\\\\\\\\//
tm"
```

to put a backslash before every slash in a pathname. It was a learning experience but it would perhaps have been quicker to use something else.

The Makefile also handles automatically running the various tools as required. Automatically-generated C++ files were given the extension `cpp`, and manually coded files were given the extension `c++`, so that they can easily be differentiated by the scripts.

3.1.5 Bash

Most of the project maintenance scripts were Bash shell scripts that used various Unix tools and were run periodically by `cron`. Snapshots of modified files were archived nightly, backups to Pelican were made weekly, and web pages were generated for viewing during supervisions (I could not find any existing "automatic HTML generator" that could do exactly what I wanted). The script to fetch MuseData files was also in Bash (using `wget`).

3.1.6 Emacs Lisp

The code editing was done in XEmacs, and because of my sight problem I used large fonts and colour syntax highlighting. Since I also needed to look at code during supervisions, and reconfiguring my supervisor's computer to my own settings would take time and could cause inconvenience, I decided to convert all of the code into HTML that mimics my settings. I did this by learning Emacs lisp and writing code to walk through the font data structures and output HTML[1].

[1] During the course of the project I was able to upgrade to XEmacs 21, and found that a similar feature (htmlize-buffer) had already been implemented.

XEmacs requires an X display to do this properly, since the "face" attributes are display-dependent, so the cron job had to open an emacs on my display rather than running emacs in batch mode. This makes little difference as I am invariably asleep when that cron job runs.

3.1.7 Web Access Gateway

Customising a web browser for a sight problem is generally fraught with difficulty, since there are so many bugs that lead to artifacts such as unreadable colour combinations (due to colour changes not being truly global), large fonts being printed on top of each other or not being used globally, irrelevant "banners" of links being enlarged to take up several screens, and so forth. Also, any configuration of a web browser assumes that you can see the configuration dialogues to begin with.

Before starting this project, I had attacked these problems by writing a CGI program in C++ which fetches a URL of your choice, sorts out bad HTML, transforms the document according to the options that you specify (of which there are many), modifies the links to point back through the CGI, and delivers it. The result is that I can browse the web on anyone's browser (albeit at a slower pace) without relying on the browser's customisability.

I used this "web access gateway"[3] to view the project web pages during supervisions, having previously constructed a CGI URL that encodes the ideal settings for them. I also made use of the gateway code when adding a CGI-based interface to the project, since the CGI-decoding and web security code was already written.

An alternative to my access gateway is BETSIE[23], a heavily-publicised Perl script written by the BBC in conjunction with RNIB (*after* I wrote the access gateway). I do not use BETSIE because it lacks customisability and its processing is inadequate for many pages.

3.1.8 Clig

Clig ("Command-Line Interpreter Generator") is a Tcl script that automatically generates (from a script file) C code for interpreting a program's command line into a `struct`, and keeps the manual page and `usage()` function synchronised with the options available. I used this, in conjunction with a code-generating shell script, to automatically generate and document the command-line interface, which is automatically updated by new grammar files. The program `man2html` was used to copy the manual page to the Web.

3.1.9 M4

The macro preprocessor M4 was used to preprocess the layout scripts, for convenience when a number of rules are identical but for one parameter (such as octave number).

Security notice: M4 allows the execution of arbitrary system commands. Care should be taken if users are allowed to upload layout scripts online.

3.2 Rationals

A class Rational, templated across integer types, along with a host of overloaded operators that allowed Rationals to be used in C expressions just like integers, was central to much of the system — used by both the postfix interpreter and the query system (in numeric sorting). All numbers are expressed as rationals, represented in string form as A or A/B where A and B are integers (there are cast operators to construct rationals from strings and vice versa).

To ease equality comparison, rationals are internally stored in a "normalised" form — they are simplified, the denominator is never negative, and 0 always has denominator 1. The normalisation code is shown in figure 3.2.

Although keeping rationals in simplified form reduces the risk of overflow, the highest representable number is inversely proportional to the denominator, and denominators tend to be multiplied during comparisons[2] and arithmetic operations — as denominators reach the square root of the largest representable integer, overflow becomes imminent.

Provided a suitably long integer type (32 bits) is used, this is not a problem in music because the denominators do not get large; implementing excessive overflow checking is likely to slow things down. If overflow checks were required then they could be implemented by noting that if $x \leq M$ and $y \leq M$ (for positive M) then $x + y \leq M$ if $x \leq 0$ or $y \leq M - x$, and $xy \leq M$ if $y \leq \left\lfloor \frac{M}{x} \right\rfloor$; similarly if $x \geq -M$ and $y \geq -M$ then $x + y \geq -M$ if $x \geq 0$ or $y \geq M + x$, and $xy \geq -M$ if $y \leq -\left\lfloor \frac{M}{x} \right\rfloor$. Thus all additions and multiplications can be checked for overflow before they are performed. The risk of overflow could conceivably be reduced by storing each rational as $a + \frac{b}{c}$, at the expense of further complicating the arithmetic.

Exceptions are thrown when illegal rationals are constructed (with denominators of 0, or from strings that are neither numeric nor of the form A/B). Encapsulating the exception class within the Rational class would be problematic

[2]For comparisons (like $<$ and $>$) it is valid to factor out the greatest common divisors of the numerators and the denominators before cross-multiplying, but this is not always helpful.

```
template<class IntType>
void Rational<IntType>::simplify() {
  if(!bottom) throw IllegalFractionException();
  // Normalise signs and 0:
  if(bottom<0) { top=-top; bottom=-bottom; }
  if(!top) bottom=1; else {
    // Euclid's algorithm for finding the greatest common divisor:
    IntType a=top,b=bottom;
    if(a<0) a=-a;
    if(b>a) { a=bottom; b=top; if(b<0) b=-b; }
    while(a%b) {
      IntType t=a/b, r=a%b;
      a=b; b=r;
    }
    top/=b; bottom/=b;
  }
}
```

Figure 3.2: Rational's normalisation code

because Rational is a template, so the exceptions merely share the namespace[3] (an alternative approach would be to have an encapsulated exception inheriting from a non-encapsulated one). It was established that all exceptions in the system should ultimately inherit from a general Exception class that has a `message()` method; this is useful for a top-level "catch" during testing.

3.3 The Postfix Interpreter

Writing a postfix interpreter is straightforward — the `readExpression` method calls `readAtom` to read the function or value. If it's a function then `readExpression` is recursively called as many times as necessary to get the arguments, then the function is evaluated and the result returned.

The class "Interpreter" handles this, and provides basic functions like arithmetic operators; all the application-specific functions (involving states and the tuple space) are provided by a subclass, making Interpreter a re-usable component.

[3]Namespaces are a feature of modern C++ — they are like Java packages.

Errors are thrown as exceptions and are caught by `readExpression`, which adds to the exception's stack trace and re-throws. If desired, the instruction stream can include comments (any value that isn't an argument is a comment) that locate the source of the instructions in the original grammar file; thus errors can be traced by examining the surrounding instructions (the interpreter's state is also printed for inspection).

A class called State handles the states described in figure 2.3; it contains a map[4] that maps each identifier to an ordered list (an STL linked list) of values and their "colours". Retrieval returns the value from the head of this list; setting a value adds to the head (but first deletes any previous setting of the same colour further down the list), and wiping a colour iterates through all lists deleting anything of that colour (this is acceptable because the lists are short). Copying to tuple space involves iterating through the map and adding each attribute, along with the head value of its list, to an instance of a class called Fact (a tuple), which has a map of identifiers to values. Both State and Fact are templated across identifier, value and colour types, but are instantiated to use strings.

3.4 The Tuple Space

A template class Database stores objects in an unsorted vector. It is instantiated to store Facts (tuples).

To assist testing, both Database and Fact have stream operators, so the tuple space can be directly inspected and modified (although this is tedious in all but the simplest of cases).

Database also has an instance of DateStamp, which is informed when the database is updated; it contains a counter that is incremented on update if it has been read since the last update (thus it is never incremented unnecessarily and overflow is unlikely). Thus a client can get the database's "date stamp" and later check to see if it has been modified since. This feature was used during testing, and might be needed in a more interactive system.

3.5 Querying the Tuple Space

The classes for querying the tuple space are shown in figure 3.3. A Cache is an abstract class that checks a DateStamp and updates its data if necessary. An Ordering is a cache of pointers to database objects (instantiated to Facts) and its constructor can get the addresses of all the objects in a database. Ordering

[4] Map is an STL class. The GNU implementation uses red-black trees.

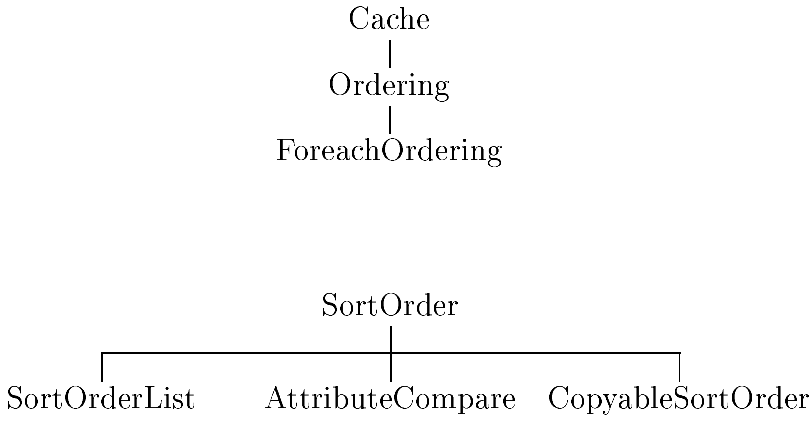

Figure 3.3: Class hierarchies for querying the tuple space

also has a pointer to a SortOrder, which it uses to sort its vector of pointers; thus the sorting operation leaves the original database untouched and it is possible to store several sorts simultaneously.

SortOrder is an abstract *function class* — it contains an `operator()` so that instances of it can be called like functions, and its purpose is to compare two values A and B and return true if A<B. This is what the STL `sort` function expects for its comparison function.

SortOrderList is an instantiation of SortOrder that contains an ordered list of other SortOrders — its comparison operator is shown in figure 3.4. AttributeCompare is a SortOrder that compares the values of a given attribute (first as Rationals, then as strings). It was also necessary to create a CopyableSortOrder, which encapsulates a pointer to another SortOrder — this is because the STL `sort` function expects it to be passed *by value* (not reference) and abstract classes can't have copy constructors.

ForeachOrdering is a specialisation of Ordering for use with the foreach command in the layout language. It can be created from the database, from another ForeachOrdering, or starting with an empty vector, and it has a SortOrderList and a method to append to it[5]. The foreach command fills one with currently-visible tuples, then adds to its sort order list[6] and inspects the result.

[5] The STL appends to lists in constant time.

[6] Since the nature of the foreach command implies that tuples that are visible are constant in the attributes that have been sorted on so far, it is not strictly necessary to add to a list; a single sort order would suffice. The list is retained for safety in any future expansion of the foreach command that does not guarantee so much. It is not a performance bottleneck.

```
template<class Datum>
bool SortOrderList<Datum>::operator() (const Datum* a,const Datum* b) const {
  vector<const SortOrder<Datum>* >::const_iterator i=theList.begin(),
    end=theList.end();
  while (i!=end) {
    if ((*(*i)) (a,b)) return true;
    else if ((*(*i)) (b,a)) return false;
    else i++;
  } return false;
}
```

Figure 3.4: SortOrderList's comparison operator

3.6 The Layout Script Interpreter

Each command in the layout language has its own class (inheriting from Script-Command) with an execute method. These are shown in figure 3.5 — the List class contains a sequential list of other commands, and other code-containing constructs (such as foreach) have a pointer to a ScriptCommand that is normally a List. The tree of instances of these classes is built up by functions that are called by a Bison-generated parser; part of the Bison file is shown in figure 3.1. There is also a separate hierarchy of Criteria classes for evaluating criteria (IsOperator contains the common functionality of all is operators, and WasOperator of all was operators).

The class Script contains the ScriptCommand tree and manages the static database of formats, procedures and signs. The ScriptCommand execute methods are passed a reference to a ScriptState, which they may change — it stores the state of the script interpreter, namely:

- Pointers to the parent Script, the output handler (which handles constructing the blocks), and the literary handler.
- The "was" status (a map of attributes to values for the was operators to use). It is necessary to have two of these, since the "was" status is changed as an expression is evaluated but the old status should continue to be used until that tuple is finished with. Therefore there is a "wasStatus" and a "newWasStatus", which is copied to "wasStatus" after the consideration of each tuple.

Class	Subclasses	
Script		
ScriptState		
ScriptCommand	List	
	Call	
	Set	
	Block	
	Foreach	
	Pad	
	NextLine	
	Have	
	Output	
	LiteraryString	
	LiteraryID	
	Digits	
Criteria	And	
	Or	
	FirstTime	
	NotFirstTime	
	IsOperator	Is
		IsNot
		Is<
		Is>
		Is<=
		Is>=
	WasOperator	Was
		WasNot
		Was<
		Was>
		Was<=
		Was>=

Figure 3.5: Class hierarchies in the layout script interpreter

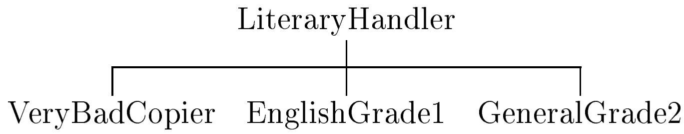

Figure 3.6: Class hierarchy of literary handlers

- A map of (attribute, bucket attribute, bucket value) → value. This is used by local to store a separate was status of any given attribute for each value of a "bucket" attribute (for example, a separate was status of 'octave' for each value of 'part').

- The current ForeachOrdering (previous values of it are kept on the call stack by Foreach).

- A DateStamp (see section 3.4), for FirstTime and NotFirstTime (section 2.6 page 18). The DateStamp is updated with every execution of have, and FirstTime and NotFirstTime check if it has changed — if so, then it must be a different execution of have and thus first time is true. FirstTime and NotFirstTime store the old value of the date stamp in a map in the ScriptState, indexed by `this` (the machine address of the command object it is associated with) — this is cleaner than storing them in the code tree because resetting the state is easier.

3.7 The Literary Handler

The abstract class LiteraryHandler contains a pure virtual method outputText that sends an arbitrary string of text to an output handler (which puts it in a block). The method is `const` because the strings that are passed to the literary handler are isolated and it should not refer back to a previous string.

The class hierarchy of literary handlers is shown in figure 3.6.

Initially I had a single class inheriting from LiteraryHandler, EnglishGrade1 (for Grade 1 English Braille), but then Saqib Shaikh of RNIB New College Worcester kindly provided his own literary Braille translation code, which I debugged and adapted to the project. Saqib's code can generate Grade 2 Braille[7] in any language based on the Latin alphabet[8]. For other languages it is possible to derive an appropriate class from LiteraryHandler; I had intended to write one

[7] Grade 1 is a letter-by-letter substitution; grade 2 has "contractions", or abbreviations.

[8] Saqib provided data files for English grades 1 and 2, for German grade 1 (including accents), and for US computer Braille.

for Japanese, which translates kanji to kana and transcribes that (eg. 日本語 → にほんご → ⠇⠮⠴⠐⠪ — see [17]), but there is a many-to-many mapping between kanji and kana and I was unable to obtain data on how to disambiguate it[9].

VeryBadCopier is for textual outputs; it just copies the text to the output. It is "very bad" because it does not handle special characters (for example, characters that mark the end of the text), but it is adequate for testing. Other literary handlers for textual outputs would be format-specific, because each format has its own set of characters that should not appear in text and its own ways of coping with them.

3.8 Formatting and Outputting

The classes for formatting and outputting are shown in figure 3.7. The basic unit to be output is SymbolForOutput, which contains the category name, sign name, and actual symbol (templated across symbol types). This is instantiated to CellForEmbossing (containing a BrailleCell, which is an array of booleans of size DotsInCell, currently 6) and CharForOutput (containing a `char` for textual output); further instantiation is possible. There are similar specialisations for OutputBlock (a block of symbols to be output, possibly with multiple lines), Outputter (which handles the low-level output), and OutputterHandler (which stores the blocks to be output and is responsible for formatting them). OutputterHandler is generalised to the abstract class ScriptOutputHandler so that Script need not be dependent on Outputter.

Outputter is at the lowest level — it provides a method `out` to output a single SymbolForOutput, and also `newLine`, `newPage`, `pageHeight` and `pageWidth`. PlainCharOutputter discards the category and sign name information and just outputs the `char` to its output stream; other Outputter classes do more. An "ASCII" embosser looks up the dot pattern in a translation table to get an equivalent ASCII character for sending to an embosser that takes ASCII characters; the translation table is read from a data file and consistency checks (that all possible dot patterns are covered and none is covered more than once) are performed. "Index" is a lower-level driver for the "Index" brand of embosser and was a fall-back (at one point it was doubtful that a complete translation table for ASCII could be obtained).

[9] There is a system that represents the kanji directly, but I was unable to obtain information about it, nor could I find anything on the Braille systems of other Asian languages.

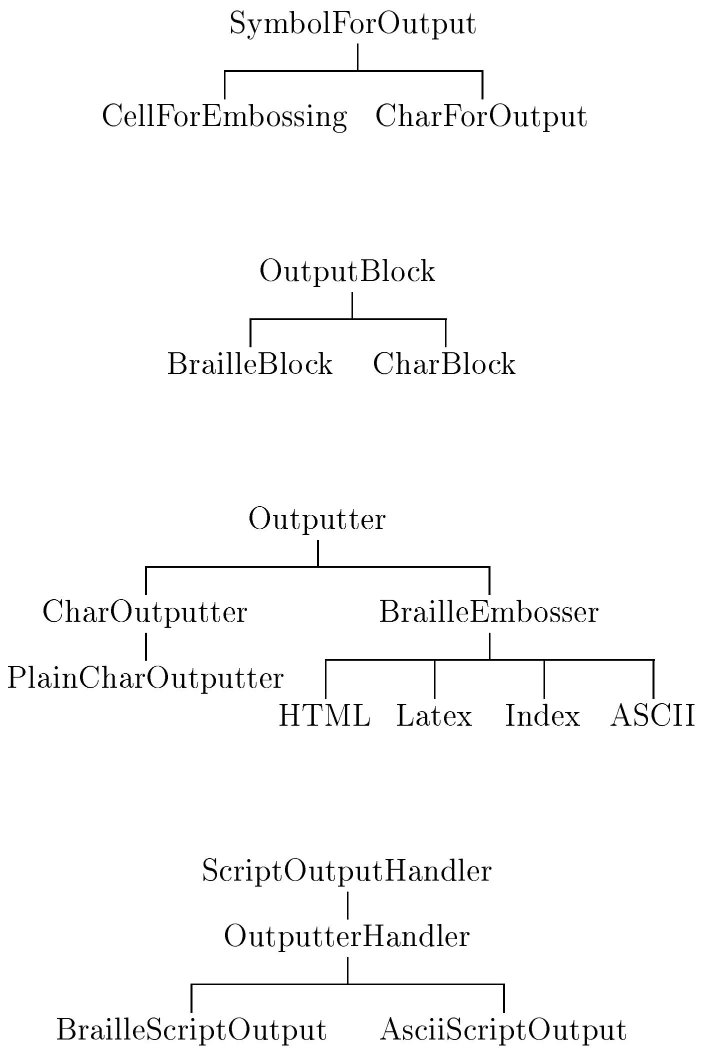

Figure 3.7: Class hierarchies for formatting and outputting

Set an iterator **current** to point to the first block

While **current** does not point past the end:

> Set **start** to **current**
>
> Advance **current** for as many blocks as will fit the width of the page (respecting formatting overrides like "centre", which makes a block exclusive to its line)
>
> Get the maximum height of blocks between **start** and **current** — if the output is paginated and the height exceeds the number of remaining lines on the page, output a page break
>
> For each n up to this maximum height:
>
> > If this page already contains lines, output a new line
> > Output whitespace to justify centred or right-justified blocks as necessary
> > Get each block between **start** and **current** to output its line n

Figure 3.8: Algorithm to format the blocks

Latex uses the sign and category data to present annotated visual Braille (as LaTeX code) for the dissertation. The algorithms for doing this took many attempts to get right, because annotating Braille music in this way is an original idea and there was no existing example of a suitable layout. HTML is similar but its output proved too unwieldy (it took the web browser some 20 minutes to render and there were alignment problems).

OutputBlock holds a vector of lines, each of which contains a vector of symbols and a padding symbol. It provides a method to output a given line number, and methods to return the block's height and width (which is maintained during construction rather than calculated on demand). The OutputHandler calls a function outputAllBlocks with its vector of completed blocks, and this function executes the algorithm outlined in figure 3.8 (it does not work well with blocks larger than a page, but these are rare).

3.9 The Scripts

Some grammar files and layout scripts for existing formats were written, as described in the following chapter. The grammar for MuseData Stage 2 files contains over 1000 rules, all inferred from informal documentation, and interpreting this represented most of the work in writing the grammar file. It was an iterative process, with the resulting parser being tested on real files to clarify the documentation. It was not feasible to eliminate all the shift/reduce and reduce/reduce conflicts (mostly due to epsilon reductions) in the grammar, but Bison appears to have resolved them correctly in all the test cases. Preprocessing of MuseData included converting DOS line breaks to Unix ones (via the tool `flip`).

The Braille output script was similarly inferred from informal documentation. It supports a reasonable number of musical attributes (some 150 Braille signs), but not everything (for example, interval signs, "in-accord" parts, blank bars, hairpins and lyrics are not supported, although the system can be extended to support them).

Chapter 4

Evaluation

The total amount of code (excluding automatically-generated code) written for this project was 5213 lines (166 Kb) in 41 files of C++, 1244 lines (41 Kb) in 16 files of make files and tool scripts and 1777 lines (86 Kb) in 7 files of notation descriptions (grammar data and output scripts).

4.1 Component Testing

Components were tested on simple cases as they were implemented, by writing simple wrapper `main` functions and test variable dumps to `cerr`. In most cases coding errors were found by the compiler and the tests revealed no further faults, although there was a logical error in the was logic that was only detected in testing, and there was a segmentation fault that had to be traced with `gdb`.

4.2 Testing the Entire System

Figure 4.1 shows a short composition in Manuscript Writer notation (the output from Manuscript Writer is shown in figures 4.2, 4.3 and 4.4). The system was set to convert this piece into two different layouts of Braille (bar-over-bar and section-by-section), in each case producing both annotated output[1] and output suitable for embossing. Part of this output is shown in figures 4.5, 4.6 and 4.7.

The output in figure 4.7 is in “ASCII Braille” (known as “American computer Braille” in the UK), which maps each cell to an ASCII character. This can be sent verbatim to most types of embosser if it were formatted to the correct page dimensions; actual Braille output has been successfully obtained in this

[1]The annotated output is necessarily cluttered; to read it, note that each label applies to the Braille cell that is vertically aligned with its *leftmost* edge.

```
x0litt.chr
x1scri.chr
x2goth.chr
~
$x0,3Page
$x1,6,0,TA Musical Diversion
$x0,4,0,AS S Brown 1996
$t0,-3; timebar etc parameters
w96,1h3,6z2y12#m0i3,5
p1,3$qDescant
$I-0/0/0/1,-1,-1,32,-1,74$c120t17o3$n2k1#j4,4$Rl2r.$Rl4o3v:1,6,8mf.f
 e.l8fl4GA l2b.l4>vpd# e.l8el4GE l2d#.l4vmpe
 c.l8cl4<BA >l8c<bl4b.'l8vpgfe l4b.l8bl4>C<B l2g.l4vmf.f
 e.l8fl4GA l2b.l4>d# e.l8el4FF vfl2g.l4vpg
 f.l8el4d#l8(fd) l4e.l8c<l4avpl8(ag) l4f.l8fl4GF l2e.l4vmfb

 (g.l8a)(ba)l4g >d.l8<al4>dl8(cd) l4e.l8dl4c(l8de) l4d.l8<bl4gl8vpab
 l4>c.l8<bab>cd c<bl4b'gl8(vmpab) >dc#l4c'<al8(vmfb>c) l4dl8(fe)l4d.l8r
 vfl4(<g.l8a)(ba)l4g >d.l8el4dl8ef l4g.l8fl4el8dc l4d.l8<bl4gl8(vpab)
 l4>c.l8<bl4al8gf l4g.l8fl4el8vpfg l4a.l8bl4GF l2e.vmfl4f
 e.l8fl4GA l2b.l4>vpd# e.l8el4GE l2d#.l4vmpe
 c.l8cl4<BA >l8c<bl4b.'l8vpgfe l4b.l8bl4>C<B l2g.l4vmf.f
 e.l8fl4GA l2b.l4>d# e.l8el4FF vfl2g.l4vpg
 f.l8el4d#l8(fd) l4e.l8c<l4avpl8(ag) l4f.l8fl4GF l2e.l4$Rr$R


 $|p2,3$qTreble
t10$I-1/0/0/1,-1,-1,48,-1,74k1#j4,4$Rl2r.$Rv:1,6,8mfo2l4a
 gbb>l8ce l4f.l8d#l4dvpf geEC <b.l8>fl4f<vmpb
 >ecl4ED# e.l8<bgbvpl4b >d#fED e.l8<bl4Bvmfa
 gbb>l8ce l4f.l8d#l4df l8eg(eg)l4bl8(bf) vfl4e.l8<vpbl4B>E
 (d#e)FB aeel8vp(fe) l2d#l4EC <b.l8gl4g>vmfe

 (dl8<b>c)(dc)D<B l4al8a>d<ab>(fg) l4eggl8fg l4gddl8vped
 l4el8efl4ec l8eded<l4bl8>vmpcd fefel4c#l8vmfde l2<al4a.l8>e
 vf(l4d.l8c)(dc)D<B l4al8>d<bl2a l4gl8bal4b>d gl8bgl4dl8(vmpc<b)
 l4gl8>cel4dd# el8<b>cl4<bl8vpab l4>el8cdl4C<B l2g.vmfl4a
 gbb>l8ce l4f.l8d#l4dvpf geEC <b.l8>fl4f<vmpb
 >ecl4ED# e.l8<bgbvpl4b >d#fED e.l8<bl4Bvmfa
 gbb>l8ce l4f.l8d#l4df l8eg(eg)l4bl8(bf) vfl4e.l8<vpbl4B>E
 (d#e)FB aeel8vp(fe) l2d#l4EC <b.l8gl4g$Rr$R


 $|p3,3$qTenor
t10$I-2/0/0/1,-1,-1,80,-1,74k1#j4,4$Rl2r.$Rl4r
 o2v:1,6,8mfl2ee l4bfbl8vpab l2>c.l4<a bfbl4vmpg
 l2al4GF l2gl4EvpG l2bl4AB l2e.l4r
 l2vmfee l4bfbl8ab l2>cd# vfl4e_<vpel8egl4b
 l2bb l4>c.l8<a>l4cvpc <bfga geevmfg

 b.l8al4gl8bg l2fl8fgab l4>c.l8dl4el8dc <l4b.l8gl4bl8>vpc<b
 l4a.l8>dl4c<a Ggg'l8vmpag l4Aaa'l8vmfga l2f.l8f'g
 vfl4(b.l8a)l4gl8bg l4f.l8gl4fl8ed l2el4gl8ag l4bgbl8vmpag
 l4el8gel4fl8ga l4el8bal4gl8vpfe l4cl8egl4ED# l2e.l4r
 o2l2vmfee l4bfbl8vpab l2>c.l4<a bfbl4vmpg
 l2al4GF l2gl4EvpG l2bl4AB l2e.l4r
 l2vmfee l4bfbl8ab l2>cd# vfl4e_<vpel8egl4b
 l2bb l4>c.l8<a>l4cvpc <bfga gee$Rr$R

 $|@
```

Figure 4.1: Musical Diversion — original Manuscript Writer code
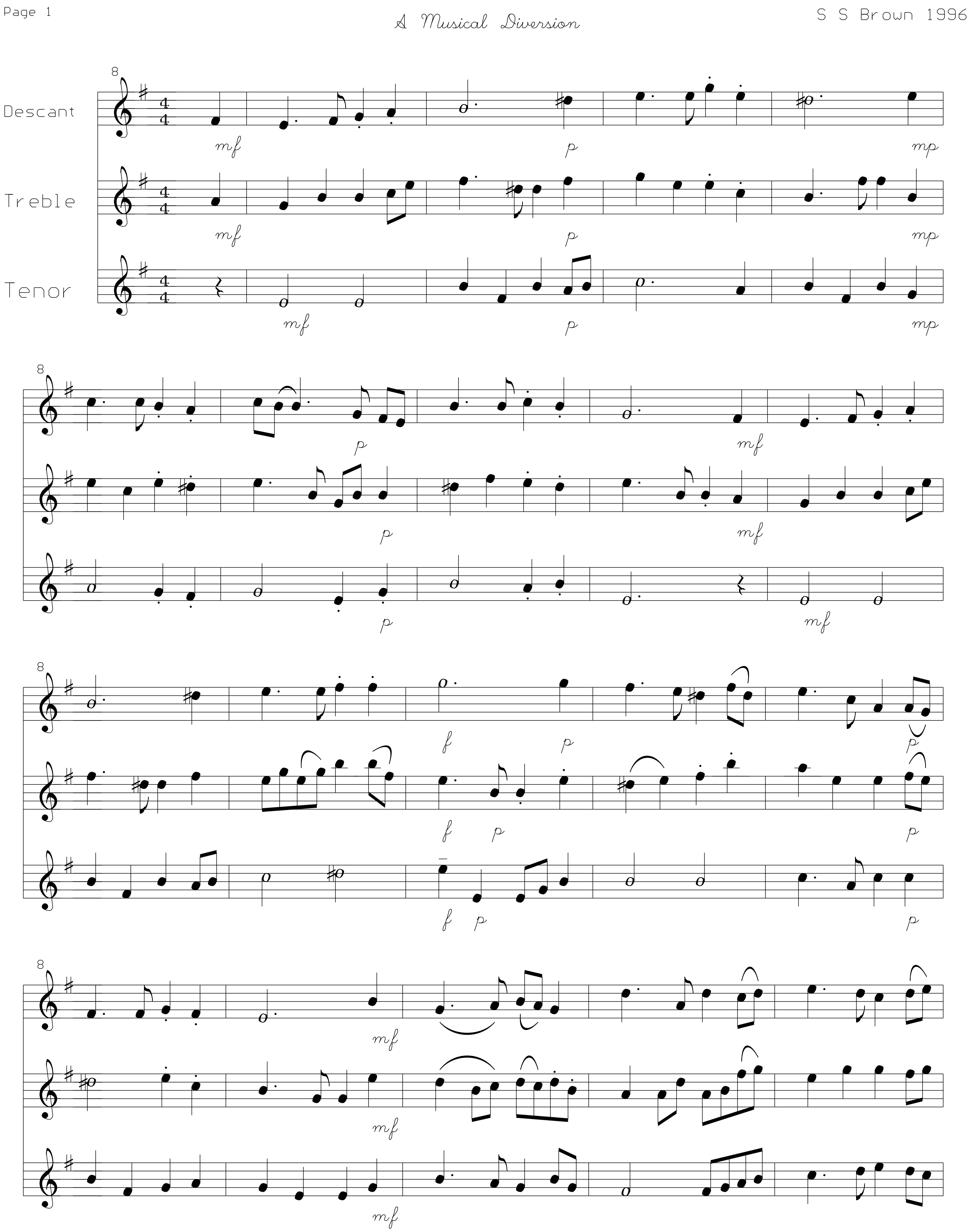
Figure 4.2: Musical Diversion — printout from Manuscript Writer (first page)
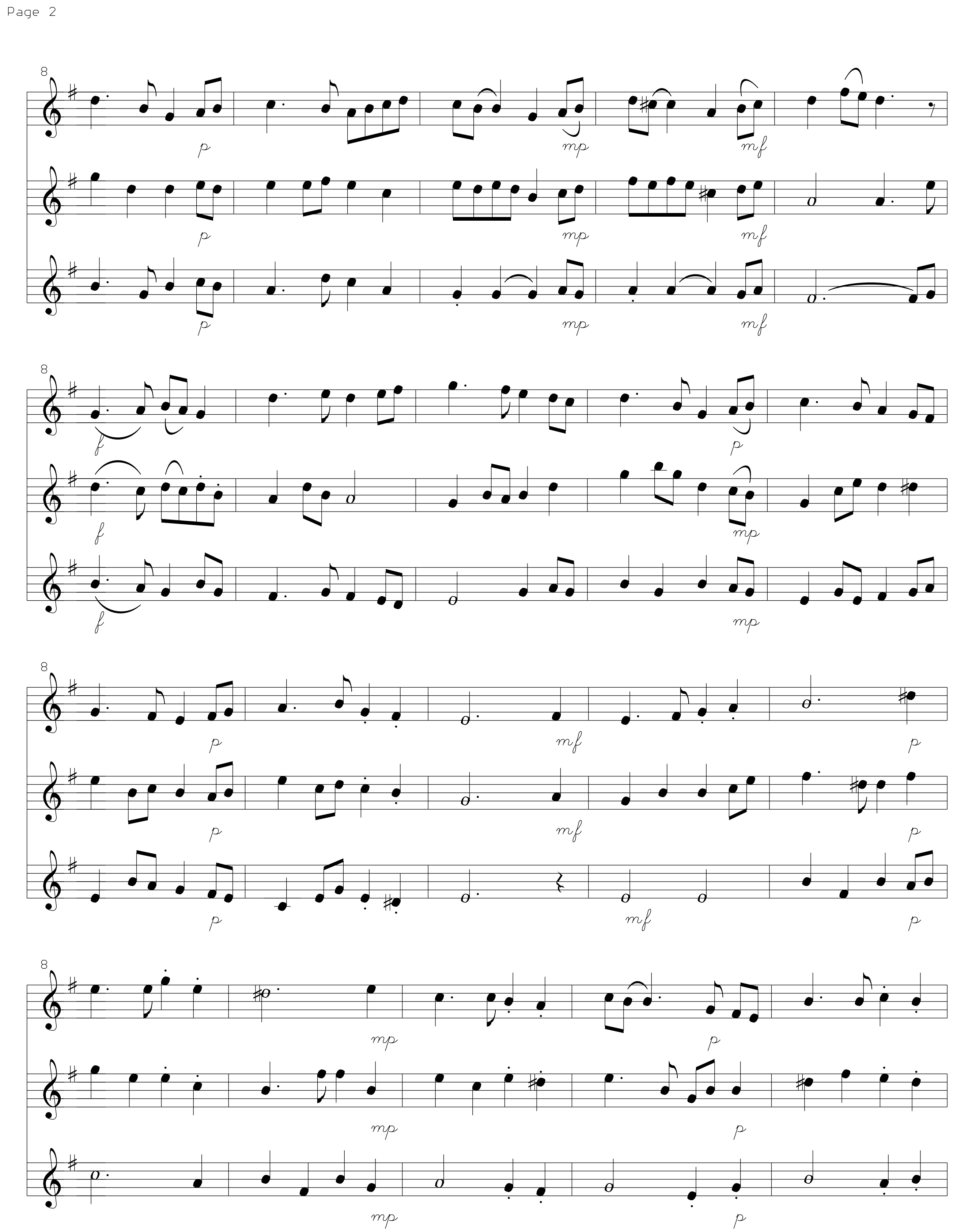
Figure 4.3: Musical Diversion — printout from Manuscript Writer (second page)
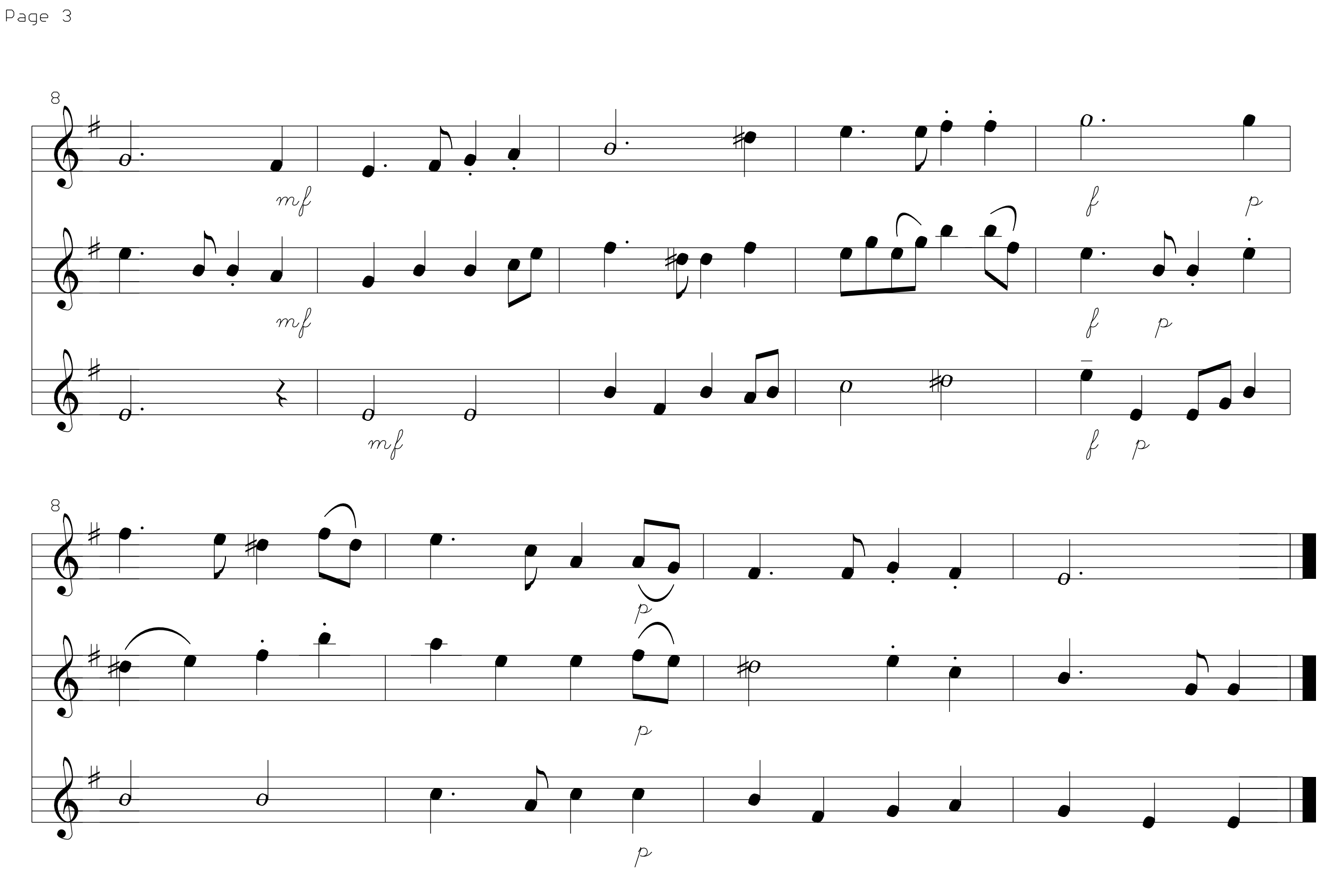
Figure 4.4: Musical Diversion — printout from Manuscript Writer (third page)

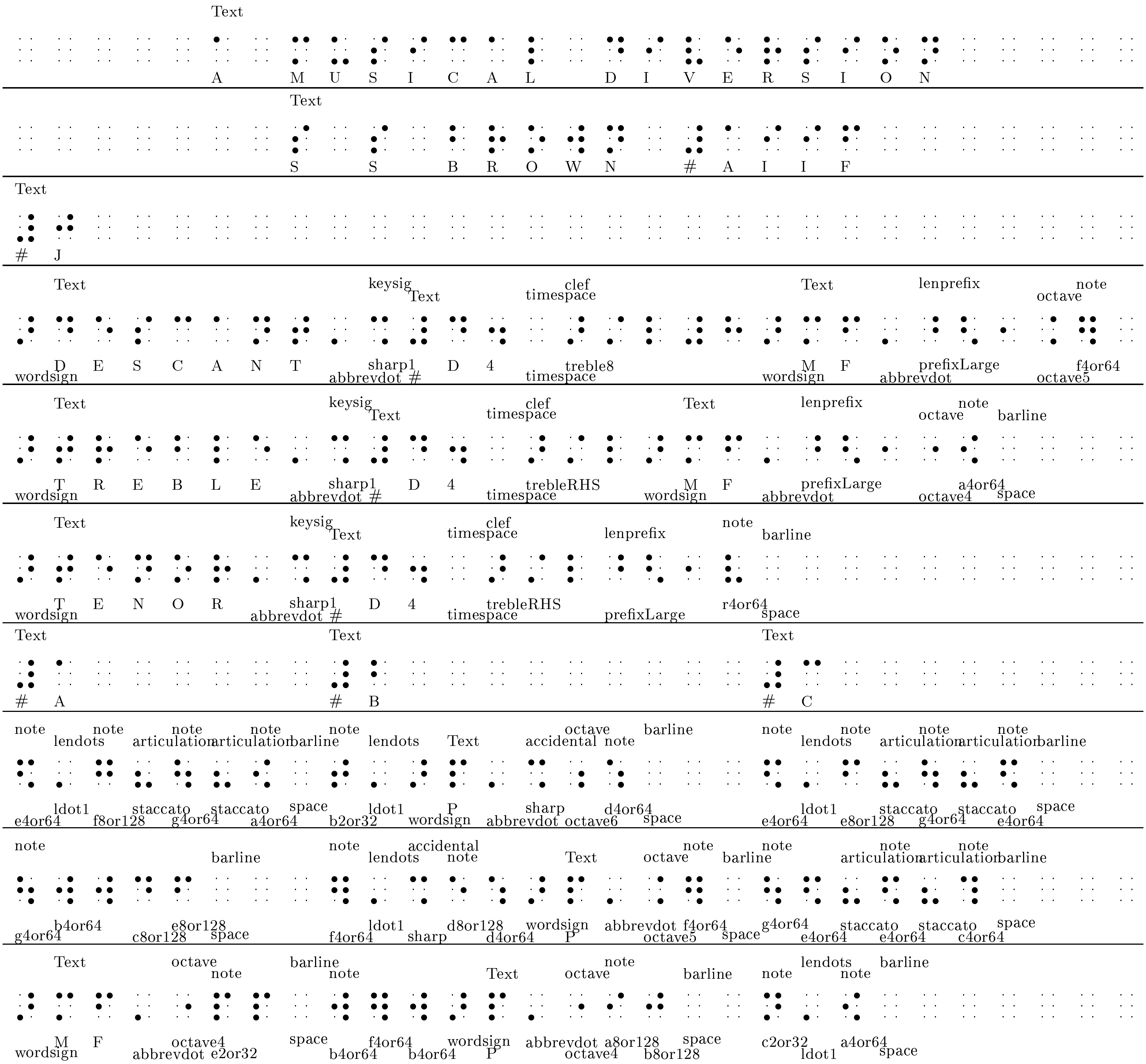

Figure 4.5: Musical Diversion — beginning of annotated Braille output in bar-over-bar format (LaTeX output)

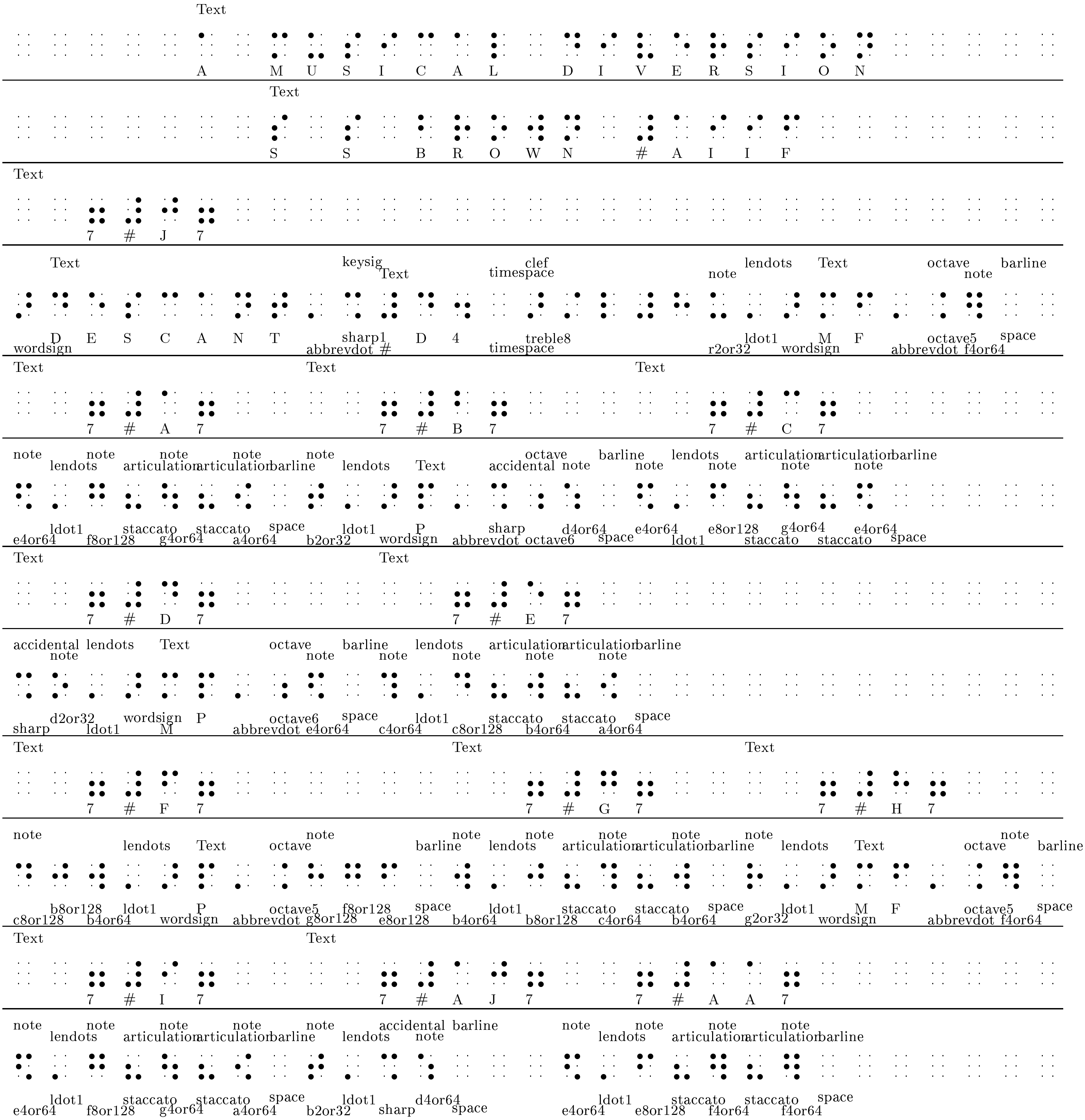

Figure 4.6: Musical Diversion — beginning of annotated Braille output in section-by-section format (LaTeX output)

way. ASCII Braille is also commonly used for posting Braille extracts on email discussion lists; I posted an earlier version of this output on Brent Hugh's Braille Music list[5] and it received an enthusiastic response.

To test the system's extensibility, a new output format that was not considered during design was added during testing. M-Tx[21] is a preprocessor for PMX[28], which in turn is a preprocessor for MusiXTEX[31], which is a set of macros for the TEX typesetting system[19]. M-Tx was chosen because it was quick to learn and provided a useful result (quality printed output); a useful subset of M-Tx was learnt and scripted mostly within a morning, and the M-Tx output from figure 4.1 is shown in figure 4.8 (part of the resulting MusiXTEX output is shown in figure 4.9; the typesetting is clearly different from Manuscript Writer's). This exercise found no major problems, although the M-Tx octave logic was a little cumbersome to express (M-Tx can only specify octave changes relatively, and octave boundaries are relative to the current note).

The system was also tested on MuseData input — figure 4.10, for example, was generated by converting the file at `http://musedata.stanford.edu/musedata/bach/rasmuss/inventio/0772/stage2/01/s01` into M-Tx. The problem in bar 9 is due to PMX's poor handling of a clef change one semiquaver into the bar[2].

4.3 Efficiency

Efficiency was not a major consideration in the design, but it can be considered retrospectively. Figure 4.1 originally took 1 minute to convert on my 1997 hardware[3]; this was reduced to 13 seconds by doing the following:

- Enabling the output script handler to look up signs by category as well as by name. Previously they had been indexed by name only and a category search had to be linear; `gprof` showed that this was a bottleneck.

- Re-writing in C the constructor of a Rational from a string, which was extensively called during a tuple space sort and was previously using inefficient I/O stream code.

[2] This is what was expressed in the original MuseData file, but it disagrees with the Henle Urtext Ausgabe edition of the piece, which puts the clef before the barline.

[3] 133MHz Cyrix CPU, 32Mb RAM, Linux. The L2 cache was disabled because the motherboard (one of the notorious 1997 "VX Pro" boards that put several dealers out of business[15]) struggled at 66MHz. I later found that slowing it to 60MHz (and the CPU to 120MHz) allowed the L2 cache to be re-enabled safely — this led to a 30% speed increase despite the slower CPU, demonstrating that the program is largely bound by medium-range memory access.

```
                              A MUSICAL DIVERSION
                                S S BROWN #AIIF
#J                           #A      #B         #C      #D          #E
>DESCANT'%#D4 >/L#H>MF'^<1.] $'G8\8[ T'>P'%;:   $'F8\8$ %O'>MP';$   ?'D8W8[
>TREBLE'%#D4 >/L>MF'^<1"[    \WWDF   ]'%E:>P'.] \$8$8?  W'.G]>MP'"W .$?8$8%:
>TENOR'%#D4 >/L^<1V          >MF'"PP W]W>P'"IJ  N'[     W]W>MP'"\   S8\8]
#F          #G      #H           #I      #AJ    #AA
DJW'>P'.HGF W'J8?8W R'>MF'.]     $'G8\8[ T'%:   $'F8]8]
$'"JHJ>P'"W %:]8$8: $'"J8W>MF'"[ \WWDF   ]'%E:] FHF;BH^2W;BJG^2
R8$>P'8"\   T8[8W   P'V          >MF'"PP W]WIJ  N%O
#AB              #AC         #AD            #AE     #AF
>F';R'>P';\      ]'F%:G;BE^2 $'D[>P';B.IH^2 ]'G8\8] P'>MF'.W
>F'.$'>P'"J8W8.$ ;B%:$^28]8W [$$>P';B.GF^2  %O8$8?  W'H\>MF'.$
>F'_8.$>P'"$FHW  TT          ?'I?>P'.?      W]\[    \$$>MF'"\
#AG               #AH            #AI        #BJ        #BA     #BB
;B\'I^2;BJI^2\    ;:'.I;:D;BE^2  $'E?;BEF^2 :'J\>P'.IJ ?'JIJDE DJW\>MP'.I;BJ^2
:;BJD^2E;BD^28E8J [I.E"IJ;B.GH^2 $\\GH      \::>P'.FE  $FG$?   FEFEW>MP'.DE
W'I\JH            QGHIJ          ?'E$ED     W'HW>P'.DJ ['.E?[  8\\\>MP'"IH
#BC              #BD        #BE                   #BF     #BG
E%D?[>MF'.J;BD^2 :;BGF^2:'X >F'.\';BI^2J;BI^2\    ;:'F:FG \'G$ED
GFGF%?>MF'.EF    "S['.F     >F'.:';BD^2;BED^28E8J [.EJS   \JIW:
8[[[>MF'"HI      Q'GH       >F';B"W'I^2\JH        ]'H]FE  P\IH
#BH             #BI     #CJ         #CA      #CB      #CC     #CD
:'J\>P';B.IJ^2  ?'J[HG  \'G$>P'.GH  ['J8\8]  P'>MF'.] $'G8\8[ T'>P'%;:
\JH:>MP';B.DJ^2 \.DF:%: $"JDW>P'"IJ .$DE8?8W R'>MF'"[ \WWDF   ]'%E:>P'.]
W\W>MP'"IH      $HF]HI  $JI\>P'"GF  ?FH8$8%: P'V      >MF'"PP W]W>P'"IJ
#CE     #CF         #CG      #CH         #CI     #DJ          #DA     #DB
$'F8\8$ %O'>MP';$   ?'D8W8[  DJW'>P'.HGF W'J8?8W R'>MF'.]     $'G8\8[ T'%:
\$8$8?  W'.G]>MP'"W .$?8$8%: $'"JHJ>P'"W %:]8$8: $'"J8W>MF'"[ \WWDF   ]'%E:]
N'[     W]W>MP'"\   S8\8]    R8$>P'8"\   T8[8W   P'V          >MF'"PP W]WIJ
#DC             #DD              #DE         #DF            #DG     #DH
$'F8]8]         >F';R'>P';\      ]'F%:G;BE^2 $'D[>P'.I;BH^2 ]'G8\8] P'
FH;BFH^2W;BJG^2 >F'.$'>P'"J8W8.$ ;B%:$^28]8W [$$>P'.G;BF^2  %O8$8?  W'H\
N%O             >F'_8.$>P'"$FHW  TT          ?'I?>P'.?      W]\[    \$$
```

Figure 4.7: Musical Diversion in bar-over-bar "ASCII Braille". The textual parts are clearly visible because Grade 1 is being used.

```
                           Title: A Musical Diversion
                            Composer: S S Brown 1996
                     Bars/Line: 4
                     Style: MyStyle
                     MyStyle: Voices V1 V2 V3; Clefs G G G
                     Meter: 4/4
                     Sharps: 1
                     Indent: 0.12
                     Name: Descant Treble Tenor

f4 | ed f8 g4 o. a o. | b2d d4s |      ed e8 g4 o. e o. | d2ds e4 |
a4 | g b b c8 e |       f4d d8s d4 f | g e e o. c o. |    bd f8+ f4 b- |
r4 | e2 e |             b4+ f b a8 b | c2d a4 |          b f b g |

cd c8 b4 o. a o. | c8 b b4d g8 f e | b4d+ b8 c4 o. b o. | g2d f4 |
e c e o. ds o. |   ed b8 g b b4 |    ds f e o. d o. |     ed b8 b4 o. a |
a2 g4 o. f o. |    g2 e4 o. g o. |   b2 a4 o. b o. |      e2d- r4 |

ed f8 g4 o. a o. | b2d d4s |      ed e8 f4 o. f o. |
g b b c8 e |       f4d d8s d4 f | e8 g ( e g ) b4 ( b8 f ) |
e2 e |             b4+ f b a8 b | c2 ds |

g2d g4 |            fd e8 d4s ( f8 d ) | e4d c8 a4 ( a8 g ) |
e4d b8 b4 o. e o. | ( ds e ) f o. b o. | a e e ( f8 e ) |
e4 o_ e- e8 g b4 |  b2 b |               c4d a8 c4 c |

f4d f8 g4 o. f o. | e2d b4+ |     ( gd a8 ) ( b a ) g4 |
d2s e4 o. c o. |    bd g8 g4 e+ | ( d b8 c ) ( d c ) d o. b o. |
b f g a |           g e e g |     bd a8 g4 b8 g |

dd+ a8 d4 ( c8 d ) |   e4d d8 c4 ( d8 e ) | d4d b8 g4 a8 b | c4d b8 a b c d |
a4 a8 d a b ( f+ g ) | e4 g g f8 g |       g4 d d e8 d |    e4 e8 f e4 c |
f2 f8 g a b |          c4d d8 e4 d8 c |    b4d g8 b4 c8 b | a4d d8 c4 a |

c b b4 g ( a8 b ) | d cs c4 a ( b8 c ) | d4 ( f8 e ) d4d r8 |
e8 d e d b4 c8 d |  f e f e c4s d8 e |   a2- a4d e8+ |
g o. g g a8 g |     a4 o. a a g8 a |     f2d f8 g |

( g4d- a8 ) ( b a ) g4 |       dd+ e8 d4 e8 f | g4d f8 e4 d8 c |
( d4d c8 ) ( d c ) d o. b o. | a4 d8 b a2 |     g4 b8 a b4 d |
( b4d a8 ) g4 b8 g |           f4d g8 f4 e8 d | e2 g4 a8 g |

d4d b8 g4 ( a8 b ) | c4d b8 a4 g8 f |  g4d f8 e4 f8 g |   a4d b8 g4 o. f o. |
g b8 g d4 ( c8 b ) | g4 c8 e d4 ds |   e b8 c b4 a8 b |   e4 c8 d c4 o. b o. |
b4 g b a8 g |        e4 g8 e f4 g8 a | e4 b8+ a g4 f8 e | c4 e8 g e4 o. ds o. |

e2d f4 | ed f8 g4 o. a o. | b2d d4s |      ed e8 g4 o. e o. | d2ds e4 |
g2d a4 | g b b c8 e |       f4d d8s d4 f | g e e o. c o. |    bd f8+ f4 b- |
e2d r4 | e2 e |             b4+ f b a8 b | c2d a4 |          b f b g |

cd c8 b4 o. a o. | c8 b b4d g8 f e | b4d+ b8 c4 o. b o. | g2d f4 |
e c e o. ds o. |   ed b8 g b b4 |    ds f e o. d o. |     ed b8 b4 o. a |
a2 g4 o. f o. |    g2 e4 o. g o. |   b2 a4 o. b o. |      e2d- r4 |

ed f8 g4 o. a o. | b2d d4s |      ed e8 f4 o. f o. |
g b b c8 e |       f4d d8s d4 f | e8 g ( e g ) b4 ( b8 f ) |
e2 e |             b4+ f b a8 b | c2 ds |

g2d g4 |            fd e8 d4s ( f8 d ) | e4d c8 a4 ( a8 g ) |
e4d b8 b4 o. e o. | ( ds e ) f o. b o. | a e e ( f8 e ) |
e4 o_ e- e8 g b4 |  b2 b |               c4d a8 c4 c |

f4d f8 g4 o. f o. | e2d |
d2s e4 o. c o. |    bd g8 g4 |
b f g a |           g e e |
```

Figure 4.8: Musical Diversion in M-Tx format.
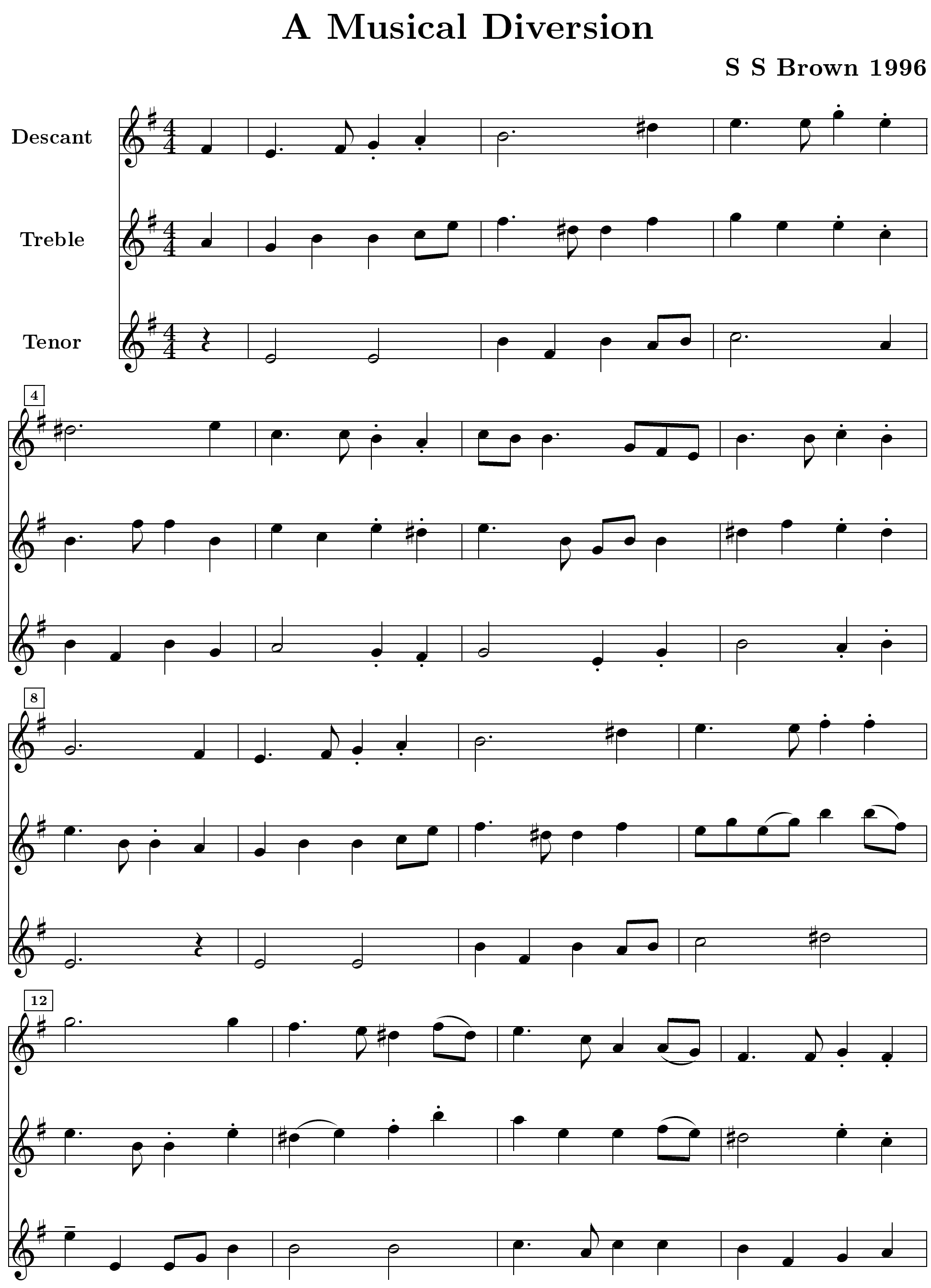
Figure 4.9: MusiXTEX output from figure 4.8.
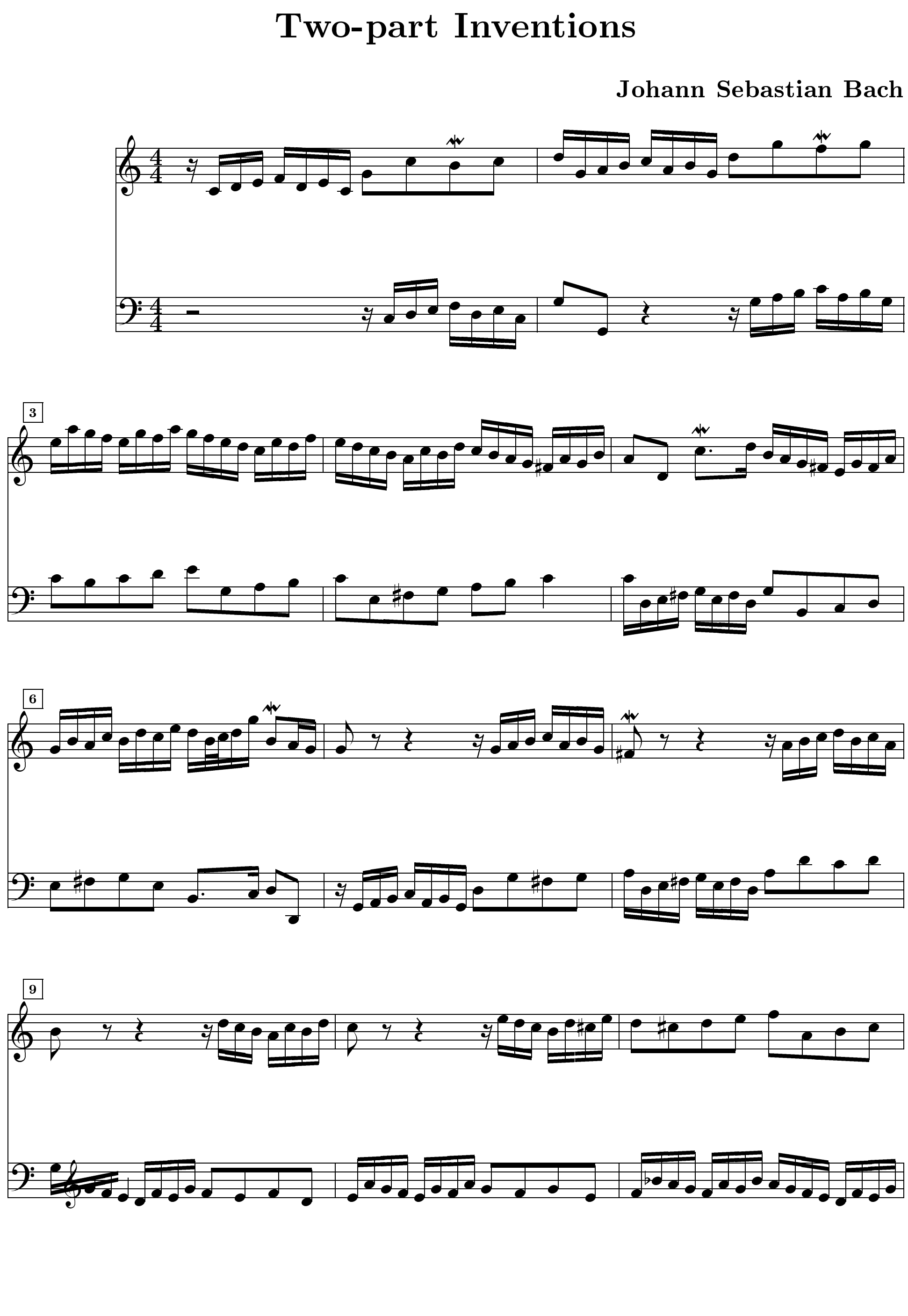
Figure 4.10: MusiXTEX output from MuseData input.

- Compiling with full optimisation.

The system is CPU-bound (there is little disk activity); it will run faster on newer machines but may still be slow for online services or for batch-converting large repositories.

Further improvements in speed may require design changes. According to `gprof`, the system still spends 35–45% of its time doing the string comparisons necessary to evaluate criteria in sign statements. If more data types are introduced into the tuple space then the need for string processing can be reduced; if each value is stored in more than one form then the need for repeated conversion can be reduced (at the expense of more memory). It is also possible to save time (at the expense of stronger coupling) by doing more preprocessing on the input's grammar files, so that the resulting code creates the tuple space *directly* rather than outputting instructions to be interpreted, and by substituting integers for identifiers.

4.4 Testing on a Substantial Work

Beethoven's famous Fifth Symphony was used. Its MuseData files total some 4 Mb. Further corrections to the MuseData grammar had to be made before they would parse. M-Tx was unable to typeset the work as PMX is limited to 7-stave systems, but Braille was produced[4]. The layout algorithms have poor locality of reference in the tuple space, so my system thrashed intensively when its physical RAM was exceeded, but I had the opportunity to use a more powerful system[5] and obtained the following readings:

Movement	Bars	Parts	Lines parsed	Tuples generated	Braille pages	CPU time (min : sec)
1	502	18	25,592	25,262	229	1:11
2	247	18	17,112	16,530	123	0:47
3	375	18	19,416	18,736	168	0:52
4	446	23	49,869	48,922	211	2:27
Total	1570		111,989	109,450	731	5:17

Since MuseData files usually generate one tuple per line[6], the above data

[4] Section-by-section Braille was used as bar-over-bar is unwieldy in symphonic scores. Standard Braille page dimensions (40 by 25) were used for the page count.

[5] 450MHz Pentium III, 256Mb RAM, Linux. Thanks to Moray Allan for the use of this.

[6] Lines that do not generate tuples include comments and general settings as well as unsupported events.

indicates that about 98% of the lines were supported. This can be improved by extending the system.

The figure of 211 Braille pages for the fourth movement looks anomalous given the higher number of tuples. This is because the style of that movement makes for briefer Braille — it has an abundance of major scales and arpeggios, so most notes do not need extra signs like accidentals, octave marks or length modifiers, and its shorter bars allow more optimal use of space (line breaks are very near the margin).

Chapter 5

Conclusion

This project has proved the concept of an extensible conversion system as given in the introduction. The language of grammar files relieves the extensions programmer from hand-coding parsers, and the layout scripting language eases the process of supporting new output formats — the system can be extended without much programming knowledge.

Although Bison simplified the construction of input parsers, it was not a perfect solution, since Backus-Naur Form grammars (particularly LALR(1) grammars) are only at their most elegant and intuitive when the language was designed for them. Many music languages can be compared to early programming languages before Algol introduced BNF, and languages designed for *ad hoc* parsing can have large grammars that take effort to maintain (as was the case with MuseData). Also, context-dependent lexing is common, which is similar to early languages in which `IF IF=THEN THEN THEN=ELSE ELSE ELSE=THEN` is valid; a top-down parser with lexer tie-in, such as CUP[7], might be appropriate for this. A common idiom in binary formats[1] (if they were to be supported) is a number n followed by n objects (like Hollerith strings in early FORTRAN); this cannot be represented by a finite context-free grammar (unless n has an upper bound and the grammar is repetitive). It would be interesting to find other solutions for automatically generating parsers.

The layout language was intuitive for most things, but can occasionally be cumbersome (for example, for Braille interval signs); there is scope for alternative approaches.

[1] such as the Amiga Module format, which is sample-based but musical data can sometimes be extracted

5.1 Possible Extensions

This project is extensible by design, so extensions such as more input and output formats, more musical attributes, and literary handlers for more languages, are naturally possible. Other possibilities include:

- Giving the system more "musical sense". At present it is simply a converter; it would be difficult (although possible) to use it for musical processing like transposition, and it cannot cope with input formats that are ambiguous and rely on musical sense to interpret correctly. One such format is Braille music itself — the system can convert *to* Braille music, but some blind composers are more interested in converting *from* Braille music so they can share their work without having to learn a textual representation. However, Braille music was not designed as an input language for computers; it is easy to write ambiguous notation in it that requires "musical sense" to interpret, so parsing it presents many challenges. Other examples of input languages that require "musical sense" to interpret include MIDI, audio, and scanned images (which require error correction).

- Alternatively, it should be possible to use the same design in systems that deal with notations other than music, for example mathematics, integrated circuit designs, maps, or UML. The design is probably too complex for simpler conversion tasks like wordprocessor binaries, images, and digital audio, but it might be usable for *message brokers*, which are used for integrating heterogeneous database systems[20][30].

- Currently, authors of grammar files and of output description files have to agree on how they will name musical attributes, in order to avoid disparity (the same thing being given different names) and collisions (different things being given the same name). This is an instance of the *nominalisation problem*, which is as-yet unsolved. A solution might include giving each author an identifier domain and inventing some form of arbiter that does intermediate processing on the tuple space[2].

There is certainly scope for further projects and further research.

[2] I have experimented with such arbitration by writing a Prolog program that builds a DAG of attribute converters and outputs code in the language of the Japanese Fifth Generation Project, KL/1[1]. The experiment had limited success due to the problem of how to generate suitable attribute converters (some of which need to deal with a plurality of tuples simultaneously), but it did express the nominalisation problem in a different form.

Bibliography

[1] The KLIC Association. *KLIC (an implementation of the parallel logic programming language KL/1).* c/o Chikayama Lab., Department of Frontier Informatics, School of Frontier Science, The University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656, Japan. `http://www.klic.org/software/klic/`.

[2] Silas S. Brown. *Manuscript Writer.* `http://www.flatline.org.uk/~silas/mwrhome/`.

[3] Silas S. Brown. *Web Access Gateway.* `http://www.flatline.org.uk/~silas/access.html`.

[4] Arkkra Enterprises. *Music Publisher (Mup).* `http://www.arkkra.com/`.

[5] Brent Hugh et al. *Braille music discussion list.* `http://www.topica.com/lists/braillem/`.

[6] Han-Wen Nienhuys et al. *LilyPond — The GNU Project Music Typesetter.* `http://www.cs.uu.nl/~hanwen/lilypond/`.

[7] Scott E. Hudson et al. *CUP Parser Generator for Java.* `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[8] Phil Farrand. *Enigma Transportable File Format.* Coda Music Technology, 6210 Bury Drive, Eden Prairie, MN 55346–1718, USA, 1987. U.S. patent #4,960,031.

[9] Ben & Jonathan Finn. *Sibelius: The Music Notation Software.* Sibelius Software Ltd, 75 Burleigh St, Cambridge CB1 1DJ. `http://www.sibelius-software.com/`.

[10] Roger Firman. *Braille Musical Notation: An Overview*, chapter 22. In Selfridge-Field [26], 1997.

[11] Cindy Grande. *The Notation Interchange File Format*, chapter 31. In Selfridge-Field [26], 1997.

[12] Philip Hazel. *Philip's Music Scribe*. 33 Metcalfe Road, Cambridge CB4 2DB.

[13] Walter B. Hewlett. *MuseData: Multipurpose Representation*, chapter 27. In Selfridge-Field [26], 1997.

[14] Walter B. Hewlett, Eleanor Selfridge-Field, David Cooper, Brent A. Field, Kia-Chuan Ng, and Peter Sitter. *MIDI*, chapter 2. In Selfridge-Field [26], 1997.

[15] Red Hill. *Avoiding Poor Motherboards*, Oct 1998. `http://www.redhill.net.au/hw-boards3.html`.

[16] Scientific Computing Associates Inc. *C-Linda User's Guide and Reference Manual*.

[17] Mitsuji Kadota. *Japanese Braille Tutorial*, Oct 1997. `http://buri.sfc.keio.ac.jp/access/arc/NetBraille/etc/brttrl.html`.

[18] Tim Kientzle. *A Programmer's Guide to Sound*. Addison Wesley Developers Press, 1998.

[19] Donald E. Knuth. *The TEXbook*. Computers and Typesetting. Addison-Wesley, 1986.

[20] Jay H. Lang. *IBM MQSeries Integrator V2.0: The Next Generation Message Broker*. `http://www-4.ibm.com/software/ts/mqseries/library/whitepapers/mqintegrator/msgbrokers.html`.

[21] Dirk Laurie. *M-Tx: Music from Text*, Nov 1998. `ftp://ftp.gmd.de/music/musixtex/software/mtx/mtxdoc.ps`.

[22] Bill McCann. *GOODFEEL Braille Music Translator*. Dancing Dots Braille Music Technology, 1754 Quarry Lane, P.O. Box 927, Valley Forge, PA 19482, USA. `http://www.netaxs.com/~ddots/`.

[23] Wayne Myers. *BETSIE (BBC Education Text to Speech Internet Enhancer)*. `http://www.bbc.co.uk/education/betsie/`.

[24] The Braille Music Subcommittee of the World Blind Union. *New International Manual of Braille Music Notation*. Studie-en Vakbibliotheek voor

Visueelen Anderszins Gehandicapten (SVB), Molenpad 2, 1016 GM Amsterdam, 1996. Sold in the UK by RNIB Customer Services, PO Box 173 Peterborough PE2 6WS.

[25] Eleanor Selfridge-Field. *Beyond Codes: Issues in Musical Representation*, chapter 33. In [26], 1997.

[26] Eleanor Selfridge-Field, editor. *Beyond MIDI: The Handbook of Musical Codes*. MIT Press, 1997.

[27] Eleanor Selfridge-Field, editor. *DARMS*, chapter 11–15. In Selfridge-Field [26], 1997.

[28] Don Simons. *PMX – a Preprocessor for MusiXTEX*. Dr. Don's PC and Harpsichord Emporium, Redondo Beach, California, USA, Dec 1999. `ftp://ftp.gmd.de/music/musixtex/software/pmx/pmx210.ps`.

[29] Leland Smith. *SCORE*, chapter 19. In Selfridge-Field [26], 1997.

[30] 4 Tier Software. *OpenMOM Message Broker — White Paper*, 1999. `http://www.xing.com/whitepaper.html`.

[31] Daniel Taupin, Ross Mitchell, and Andreas Egler. *MusiXTEX: Using TEX to write polyphonic or instrumental music*, Apr 1999. `ftp://ftp.gmd.de/music/musixtex/musixdoc.ps`.

[32] Guido van Rossum et al. *Python: An Interpreted, Interactive, Object-oriented Programming Language*. `http://www.python.org/`.

[33] Chris Walshaw. *ABC2MTEX: An easy way of transcribing folk and traditional music*. School of Maths, University of Greenwich, London, Jan 1997.

Appendix A

The Musical Attributes Database

Attribute name	Context	Values	Notes
accent	type=note	non-0 if the note is accented	
accidental	type=note	sharp, flat, natural, dblsharp, dblflat	See also implicit-accidental
barline-type	type=barline	normal, dashed, double, repeat-start, repeat-end, repeat-startend	For invisible, don't have barline event
barNumber		Integer	The bar number of a barline event is that of the previous bar
bowing-type	type=note	up, down, none	
breath	Notes, barlines etc	0 or 1	1 if a breath mark comes after this symbol (above it for barlines)
clef-type	type=clef	treble8va, treble, soprano, bass8va, alto, tenor, bass, treble8down, bass8down	
close-ending	stop-ending	yes, no	yes if a vertical descending line is drawn at the close of the time-bar; no otherwise

Attribute name	Context	Values	Notes
ending-number	start-ending and stop-ending	Integer from 1 upwards	An "ending" (volta) is a timebar used in repeats.
fermata	notes, rests, barlines	1 if there is a fermata on the symbol	
has-roll	type=note	0 or 1	
has-trill	type=note	0 or 1	
implicit-accidental-a through implicit-accidental-g	type=note	sharp, flat, natural, dblsharp, dblflat	The one used is the one that corresponds to the note letter.
keysig-acctype	type=keysig	sharp, flat, natural (the accidental type of the key signature)	
keysig-numaccs	type=keysig	Integer number of accidentals in the key signature	
letter	type=note	a, b, c, d, e, f, g (the English letter name of the note)	
mordent-type	type=note	normal, inverted, none	
notatedDots	notes and rests	Integer	The number of dots written after the note to lengthen it
notatedLength	Notes and rests	A rational, 1 being a crotchet, 1/2 being a quaver etc	Related to realLength by the time signature and any tuplets the note is in. notatedLength does not include the effect of dots.
note-head-type	type=note	normal, diamond, crossed, none	
octave	type=note	Integer ANSI octave number	

Attribute name	Context	Values	Notes
part	Any part-specific event	Integer, 1 being the first part	
realLength	Any event with length	A rational with 1 = the length of a bar. time + realLength = end time	
staccatissimo	type=note	0 or 1	
staccato	type=note	non-0 if the note is staccato	
stems	Notes, rests etc	up, down, automatic, none	Which direction the stems go; used for multiple voices on one stave
tenuto	type=note	non-0 if the note has a tenuto sign	
text	type=text	String	The text itself, in whatever character set is being used
text-offset	type=text	Integer	Used as a sort key when several text events of the same type occur at the same time, but not necessarily present. Can be a verse number.
text-type	type=text	header, title, composer, copyright, above-stave, below-stave, stave-label, lyric	
tieToPrevious	type=note	non-0 if the note is tied to the previous note in the part	
time	Any event with a time or a starting time	A rational, 1 being the length of a bar; 0 is the start of the piece	

Attribute name	Context	Values	Notes
timesig-bottom	type=timesig	Integer bottom number of time signature	
timesig-top	type=timesig	Integer top number of time signature	
tuplet-bottomval	type=tuplet	Integer; the number to print after the colon in a tuplet (eg. the 2 in 3:2, "3 in the space of 2"), if there is one.	See tuplet-topval
tuplet-topval	type=tuplet	Integer; the number to print on a tuplet (assumed to be referring to that many notes of length 'length')	
turn-type	type=note	normal, inverted, none	
type		note, rest, volume, volume-change, tuplet, keysig, timesig, openphrase, closephrase, text, barline, segno, pedal-up, pedal-down, start-ending, stop-ending, clef	
volume-string	type=volume, type=volume-change	ppp, pp, p, mp, mf, f, ff, fff etc; cresc, dim etc (< or > for hairpins)	

Appendix B

The Braille Signs Database

Sign	Name	Category	Criteria
⠽	c1or16	note	type is note and first time and letter is c and (notatedLength is 4 or notatedLength is 1/4 or notatedLength is 1/64)
⠵	d1or16	note	type is note and first time and letter is d and (notatedLength is 4 or notatedLength is 1/4 or notatedLength is 1/64)
⠯	e1or16	note	type is note and first time and letter is e and (notatedLength is 4 or notatedLength is 1/4 or notatedLength is 1/64)
⠿	f1or16	note	type is note and first time and letter is f and (notatedLength is 4 or notatedLength is 1/4 or notatedLength is 1/64)
⠷	g1or16	note	type is note and first time and letter is g and (notatedLength is 4 or notatedLength is 1/4 or notatedLength is 1/64)

Sign	Name	Category	Criteria
⠮	a1or16	note	type is note and first time and letter is a and (notatedLength is 4 or notatedLength is 1/4 or notatedLength is 1/64)
⠾	b1or16	note	type is note and letter is b and (notatedLength is 4 or notatedLength is 1/4 or notatedLength is 1/64)
⠍	r1or16	note	type is rest and (notatedLength is 4 or notatedLength is 1/4)
⠝	c2or32	note	type is note and first time and letter is c and (notatedLength is 2 or notatedLength is 1/8)
⠕	d2or32	note	type is note and first time and letter is d and (notatedLength is 2 or notatedLength is 1/8)
⠏	e2or32	note	type is note and first time and letter is e and (notatedLength is 2 or notatedLength is 1/8)
⠟	f2or32	note	type is note and first time and letter is f and (notatedLength is 2 or notatedLength is 1/8)
⠗	g2or32	note	type is note and first time and letter is g and (notatedLength is 2 or notatedLength is 1/8)
⠎	a2or32	note	type is note and first time and letter is a and (notatedLength is 2 or notatedLength is 1/8)
⠞	b2or32	note	type is note and first time and letter is b and (notatedLength is 2 or notatedLength is 1/8)
⠥	r2or32	note	type is rest and (notatedLength is 2 or notatedLength is 1/8)

Sign	Name	Category	Criteria
⠹	c4or64	note	type is note and first time and letter is c and (notatedLength is 1 or notatedLength is 1/16)
⠱	d4or64	note	type is note and first time and letter is d and (notatedLength is 1 or notatedLength is 1/16)
⠫	e4or64	note	type is note and first time and letter is e and (notatedLength is 1 or notatedLength is 1/16)
⠻	f4or64	note	type is note and first time and letter is f and (notatedLength is 1 or notatedLength is 1/16)
⠳	g4or64	note	type is note and first time and letter is g and (notatedLength is 1 or notatedLength is 1/16)
⠪	a4or64	note	type is note and first time and letter is a and (notatedLength is 1 or notatedLength is 1/16)
⠺	b4or64	note	type is note and first time and letter is b and (notatedLength is 1 or notatedLength is 1/16)
⠧	r4or64	note	type is rest and (notatedLength is 1 or notatedLength is 1/16)
⠙	c8or128	note	type is note and first time and letter is c and (notatedLength is 1/2 or notatedLength is 1/32)
⠑	d8or128	note	type is note and first time and letter is d and (notatedLength is 1/2 or notatedLength is 1/32)
⠋	e8or128	note	type is note and first time and letter is e and (notatedLength is 1/2 or notatedLength is 1/32)

Sign	Name	Category	Criteria
⠛	f8or128	note	type is note and first time and letter is f and (notatedLength is 1/2 or notatedLength is 1/32)
⠓	g8or128	note	type is note and first time and letter is g and (notatedLength is 1/2 or notatedLength is 1/32)
⠊	a8or128	note	type is note and first time and letter is a and (notatedLength is 1/2 or notatedLength is 1/32)
⠚	b8or128	note	type is note and first time and letter is b and (notatedLength is 1/2 or notatedLength is 1/32)
⠭	r8or128	note	type is rest and (notatedLength is 1/2 or notatedLength is 1/32)
⠰⠣⠂	prefix256	lenprefix	(type is rest or type is note) and notatedLength is 1/64 and notatedLength wasnot 1/64
⠐⠣⠂	prefixLarge	lenprefix	(type is rest or type is note) and notatedLength is>= 1/2 and notatedLength was<= 1/8
⠠⠣⠂	prefixSmall	lenprefix	(type is rest or type is note) and notatedLength is<= 1/8 and (notatedLength was>= 1/2 or notatedLength was 256) and notatedLength isnot 256
⠀	space	barline	type is barline
⠅	dashedbar	barlinetype	type is barline and barline-type is dashed
⠣⠅	doublebar	barlinetype	type is barline and barline-type is double
⠣⠶	rptstart	barlinetype	type is barline and barline-type is repeat-start

Sign	Name	Category	Criteria
⠣⠆	rptend	barlinetype	type is barline and barline-type is repeat-end
⠣⠶⠣⠆	rptend	barlinetype	type is barline and barline-type is repeat-startend
⠈⠈	octave0	octave	octavelogic(-2)
⠈	octave1	octave	octavelogic(-1)
⠘	octave2	octave	octavelogic(0)
⠸	octave3	octave	octavelogic(1)
⠐	octave4	octave	octavelogic(2)
⠨	octave5	octave	octavelogic(3)
⠰	octave6	octave	octavelogic(4)
⠠	octave7	octave	octavelogic(5)
⠠⠠	octave8	octave	octavelogic(6)
⠩	sharp	accidental	type is note and accidental is sharp
⠣	flat	accidental	type is note and accidental is flat
⠡	natural	accidental	type is note and accidental is natural
⠩⠩	dblsharp	accidental	type is note and accidental is dblsharp
⠣⠣	dblflat	accidental	type is note and accidental is dblflat
⠖	trill	ornament	type is note and has-trill is 1
⠠⠲	turn-above	ornament	type is note and turn-type is normal
⠠⠲⠇	inv-turn-above	ornament	type is note and turn-type is inverted
⠐⠖	mordent	ornament	type is note and mordent-type is normal
⠐⠖⠇	inv-mordent	ornament	type is note and mordent-type is inverted
⠣⠃	downbow	bowing	type is note and bowing-type is down
⠣⠄	upbow	bowing	type is note and bowing-type is up

Sign	Name	Category	Criteria
⠩	sharp1	keysig	type is keysig and keysig-acctype is sharp and keysig-numaccs is 1
⠣	flat1	keysig	type is keysig and keysig-acctype is flat and keysig-numaccs is 1
⠡	natural1	keysig	type is keysig and keysig-acctype is natural and keysig-numaccs is 1
⠩ ⠩	sharp2	keysig	type is keysig and keysig-acctype is sharp and keysig-numaccs is 2
⠣ ⠣	flat2	keysig	type is keysig and keysig-acctype is flat and keysig-numaccs is 2
⠡ ⠡	natural2	keysig	type is keysig and keysig-acctype is natural and keysig-numaccs is 2
⠩ ⠩ ⠩	sharp3	keysig	type is keysig and keysig-acctype is sharp and keysig-numaccs is 3
⠣ ⠣ ⠣	flat3	keysig	type is keysig and keysig-acctype is flat and keysig-numaccs is 3
⠡ ⠡ ⠡	natural3	keysig	type is keysig and keysig-acctype is natural and keysig-numaccs is 3
⠼ ⠙ ⠩	sharp4	keysig	type is keysig and keysig-acctype is sharp and keysig-numaccs is 4
⠼ ⠙ ⠣	flat4	keysig	type is keysig and keysig-acctype is flat and keysig-numaccs is 4
⠼ ⠙ ⠡	natural4	keysig	type is keysig and keysig-acctype is natural and keysig-numaccs is 4
⠼ ⠑ ⠩	sharp5	keysig	type is keysig and keysig-acctype is sharp and keysig-numaccs is 5
⠼ ⠑ ⠣	flat5	keysig	type is keysig and keysig-acctype is flat and keysig-numaccs is 5
⠼ ⠑ ⠡	natural5	keysig	type is keysig and keysig-acctype is natural and keysig-numaccs is 5

Sign	Name	Category	Criteria
⠼⠋⠩	sharp6	keysig	type is keysig and keysig-acctype is sharp and keysig-numaccs is 6
⠼⠋⠣	flat6	keysig	type is keysig and keysig-acctype is flat and keysig-numaccs is 6
⠼⠋⠡	natural6	keysig	type is keysig and keysig-acctype is natural and keysig-numaccs is 6
⠼⠛⠩	sharp7	keysig	type is keysig and keysig-acctype is sharp and keysig-numaccs is 7
⠼⠛⠣	flat7	keysig	type is keysig and keysig-acctype is flat and keysig-numaccs is 7
⠼⠛⠡	natural7	keysig	type is keysig and keysig-acctype is natural and keysig-numaccs is 7
⠀	timespace	timespace	type is timesig
⠤	staccato	articulation	type is note and staccato is 1
⠠⠦	staccatissimo	articulation	type is note and staccatissimo is 1
⠨⠦	accent	articulation	type is note and accent is 1
⠸⠦	tenuto	articulation	type is note and tenuto is 1
⠄	ldot1	lendots	(type is rest or type is note) and notatedDots is 1
⠄⠄	ldot2	lendots	(type is rest or type is note) and notatedDots is 2
⠄⠄⠄	ldot3	lendots	(type is rest or type is note) and notatedDots is 3
⠈⠉	singletie	tie	(type is rest or type is note) and tieToNext is 1
⠸	tupstart	tupletA	type is tuplet
⠄	tupend	tupletB	type is tuplet
⠜⠂	halfbreath	breath	breath is 1
⠣⠇	fermnormal	fermata	(type is note or type is rest) and fermata is 1

Sign	Name	Category	Criteria
⠇⠣⠇	fermbarline	fermata	type is barline and fermata is 1
⠡⠉	pedup	pedal-up	type is pedal-up
⠣⠉	peddown	pedal-down	type is pedal-down
⠬	segno	segno	type is segno
⠜⠌⠇	trebleRHS	clef	type is clef and clef-type is treble and clef-type wasnot bass
⠜⠌⠅	trebleLHS	clef	type is clef and clef-type is treble and clef-type was bass
⠜⠼⠇	bassLHS	clef	type is clef and clef-type is bass and clef-type wasnot treble
⠜⠼⠅	bassRHS	clef	type is clef and clef-type is bass and clef-type was treble
⠜⠬⠇	altoclef	clef	type is clef and clef-type is alto
⠜⠬⠐⠇	tenorclef	clef	type is clef and clef-type is tenor
⠜⠌⠇⠼⠓	treble8	clef	type is clef and clef-type is treble8va
⠜⠼⠇⠼⠓	bass8	clef	type is clef and clef-type is bass8va
⠜⠌⠇⠼⠦	treble8b	clef	type is clef and clef-type is treble8down
⠜⠼⠇⠼⠦	bass8b	clef	type is clef and clef-type is bass8down
⠄	tracking		
⠐	musichyphen		
⠜	wordsign		
⠄	abbrevdot		
⠚	U0		
⠁	U1		
⠃	U2		
⠉	U3		
⠙	U4		
⠑	U5		

Sign	Name	Category	Criteria
⠋	U6		
⠛	U7		
⠓	U8		
⠊	U9		
⠴	L0		
⠂	L1		
⠆	L2		
⠒	L3		
⠲	L4		
⠢	L5		
⠖	L6		
⠶	L7		
⠦	L8		
⠔	L9		
⠰⠃	openphrase	openphrase	type is openphrase
⠘⠆	closephrase	closephrase	type is closephrase

Appendix C

Contact details

Should anyone wish to contact me about this project after I have left Cambridge, there are a number of email addresses I have at present that might remain operational for a few years. Feel free to canvass these addresses in order to find whether any of them still work.


```
silas@flatline.org.uk
silas@accu.org
access@tardis.ed.ac.uk
silasbrown@bigfoot.com
silas@earthling.net
silas@biosys.net
```


You can also try my home address:


```
Silas S. Brown
''Pipers''
5 Gulliver's Orchard
Shipton Gorge
BRIDPORT
Dorset
DT6  4LS
UK
```

Appendix D

Project Proposal

A Representation and Conversion System for Musical Notation

Silas S. Brown (ssb22), St John's College

13th October 1999

D.1 Introduction

There are numerous ASCII-based input languages for driving music typesetting software (for example, DARMS, Score, GNU Lilypond, PMS, MUP, ABC, MuTEX, MWR and so on), and it would be useful to be able to convert between some of these. It would also be useful to convert these formats into Braille, since there is a large repository of music existing in these formats because most music publishers keep electronic copies of their scores for future editions[1]. Making use of this data is likely to yield much better results than attempting to transcribe from MIDI, which is a performance-based format that does not record details of notation, and although there is a large repository of MIDI files in the public domain, their average quality is fairly low from a notation-based viewpoint.

Since there are a variety of Braille music standards in use worldwide, it would not be feasible to do the necessary research and implementation to support all of them within this project, and this is further complicated by the fact that the standards can change, and individuals and organisations can have non-standard preferences in their notation. It is therefore necessary to parameterise the format of Braille that can be used, so that it can be customised. An ideal extension of this would be to parameterise formats in general, so that, within reason, support for arbitrary new input and output formats can be added to the program with scripts.

This project aims to design and implement a set of data structures for representing musical notation in a general and extensible way, a language for accessing the structures, some parser generator scripts for reading a number of ASCII-based formats into the structures, and some code to transform them into a number of output notations including Braille. It should parameterise at least the Braille output, but an extension is to parameterise as much as possible. The end results of this project should include the code, details of the data structures, transcripts of interactions in the language (including error reporting), and an example of a real piece of music that has been converted into various forms.

D.2 Resources Required

In the event of my own system being stolen, destroyed, or otherwise rendered inoperable, if I am to continue working efficiently then I will need access to a system running X11 (such as Linux), for reasons described below. Since the

[1]In an informal conversation, a representative of a major music publisher told me that, if software existed to translate their in-house music language into Braille, they would be happy to run it. This alone could solve the Braille music shortage problem.

PWF can no longer contain such systems, I would need access to one of the CL research systems. (This is only in the event of my own system becoming inoperable.)

The reasons are as follows: Since I am partially sighted, I need to heavily customise the user interface of whatever system I am using if I am to work efficiently. My own machine (which has a 21" monitor) runs Linux and X11 in low resolution, and makes use of X11's ability to scroll around a desktop that is larger than the screen, so workspace is not overly constricted by the low resolution and large print size. The X servers under Windows NT on the PWF cannot do this, and attempts to make it readable usually result in menus and dialogues being printed off the edge of the desktop and impossible to get to. Although the Computing Service has set up a terminal with a low-resolution screen, they have been unable to solve the X server problem, and I believe that this is because of Windows NT's technical limitations[2]. X11, however, is a solution that is known to work.

D.3 Starting Point

I have A-level music and I have a music notation program in C++, capable of writing a number of formats including printed notation, MIDI, and one particular type of Braille notation. It is unlikely that this code will be included in the project (I began coding shortly after beginning to learn C++ in 1993, and much of it is somewhat "hacked out").

D.4 Substance and Structure

The major work items in this project are:

1. The design of a flexible abstract data type suitable for representing musical notation in a general way.

2. The implementation and testing of code to handle that structure, and of a language for accessing and updating it. The language for updating should be designed with the rest of the project in mind, since it is likely that it

[2] The X server is designed to run in user mode, without direct access to the video hardware. Since such direct access is required in order to provide a responsive virtual desktop, the server does not contain the code to do this and wouldn't be able to even if I could run it as Administrator.

will be the expected output of the input language parsers and the expected input of the output notation synthesisers.

3. The design and implementation of an error handler. As well as syntax checking, this should allow user-specified constraints and assertions on the data, and error reports should be helpful, pinpointing the true source of the error as much as is feasible. A fall-back position is to simply state where the error was detected.

4. Write some parser-generator scripts for reading a number of ASCII-based formats into the internal representation.

5. Design and implement algorithms for transforming the internal representation into at least parameterised Braille and preferably all the input formats.

6. Test on a real piece of music.

7. (Possible extension.) The design of a grammar specification language that can be translated into a script for a parser generator such as yacc, and the implementation of such a translator and a synthesiser that can do the reverse process, ie. marshallise the knowledge base in a form that conforms to the specified grammar. The extra language is needed because pure yacc is Turing-powerful (it can include arbitrary C code) and it would not be feasible to reverse the process to generate a synthesiser unless the yacc input is restricted. This means that it is not feasible for this project to aim to support all possible representation languages (parser generators such as yacc would be inappropriate for that anyway as they are restricted to particular classes of grammar), but it is still possible for the user of the tool to tailor the grammar of the marshallised knowledge base to an extent.

8. The implementation of error checking on this extra language, to whatever degree proves feasible.

The data type used for representing musical notation should be flexible and extensible. It should have the following features:

- It is assumed that it can be held in memory in its entirety.

- It should not force a "note" as the basic unit, since some events (such as changes in dynamic) can occur independently of notes. The language should include features for addressing such items as "the nth *note* in this bar", "the nth *event* in this bar" (which may not necessarily be a note), "the nth *attribute* of this note", "the nth *ornament* of this note" (which will access a subset of the attributes), and so on. **This nomenclature should be extensible.** The most general retrieval query might take the form "make an array of all items of type T that are contained by (or contain) object O".

- Items such as phrasemarks and crescendi can be specified in a similar fashion; they can in a sense "contain" other events ("contain" in this instance meaning "applies to the same time as"). The advantage of the above discipline is that, if things such as dynamics and phrasemarks are irrelevant to a particular request, they will be transparent to it; a request for *notes* in a *bar* will not return excessive information about whether those notes also happen to be in a *phrase*, and if the database is extended to include new ways of structuring, existing code that uses it will still work.

- Constraints and assertions can often take a similar form, such as "no slur is contained by more than one part"[3]; of course, error checking should be more helpful than simply stating that a constraint has failed somewhere.

- It would be tempting to implement rule systems for inferring facts that are *implied* by the notation. An example of this is "if a note is explicitly flat, then all the notes in the rest of that bar on that stave with the same letter name are *im*plicitly flat (this being a different attribute) unless otherwise stated." However, this is dangerous because not all input languages actually have this rule; some music programs, when faced with input that means "B flat followed by B", will make the second B explicitly natural. Since what is explicit and what is implicit depends on the language, it would be better to leave it to the input and output algorithms to keep track of the current state and change it as necessary; however, information about what was originally explicit and what was implicit *should* be kept in the structures, since otherwise information could be unnecessarily lost (for example, editorial "reminder" accidentals that should not technically be there but that the editor wishes to include).

- However, typesetting decisions (such as the placement of beams) should be treated differently. The aim of this project is not to make a fully-featured music typesetter, and it would be silly to apply, for example, a mediocre beam-inference algorithm to data entering the intermediate format, and

[3]Technically, this is not entirely true, unless a grand staff is counted as one part and *avant-garde* notation is ignored.

then output instructions that explicitly override a far superior algorithm in the destination typesetting software. Indeed, it could be argued that even *explicit* typesetting decisions can be dropped, since they are usually made explicit to overcome a limitation of some particular software package. However, this may not always be the case, and some typesetting or formatting decisions will have to be made when the output is to be read by a human rather than a computer (Braille being an obvious example). This issue must be resolved as part of the project, perhaps involving user customisation.

The language for adding to the knowledge base should ideally be designed such that it is possible to write a yacc (or similar) script of the form:

```
S: t1 t2 ... tn { puts("..."); };
t1: p1 p2 ... pn { puts("..."); };
...
```

which will translate a user-described input file into instructions for building the knowledge base. The advantage of this form is that it is feasible to reverse it to make a synthesiser. However, since it may not be the only form with this property, the project will not restrict itself to it; it is mentioned here only as proof of concept.

D.5 Plan of Work

The following *outlines* a timetable:

Week	Work to be done	Deliverables
1–2	Design the abstract data type, and the language for accessing and updating it. Manually test the design by choosing a reasonably complex score and checking that it can represent all the features in that score.	A description of the data type and an informal specification of the language. An argument that it can represent everything in the test score.

Week	Work to be done	Deliverables
3–4	Implement and test the language used for accessing the abstract data type. Write one parser generator script so that some existing music can be easily converted to it, and interact with the database to manually test that it worked.	Code for the language; parser generator script; transcript of test interactions
5–6	Investigate how the Braille output should be parameterised; alternatively, if feasible, investigate how I/O in general can be parameterised and use the Braille as an example. Investigate what typesetting or formatting decisions need to be made and what algorithms should be used to make them; solve the problem of how formatting decisions should be treated.	
7–8	Finish implementing the Braille output code, and add code for other output languages if possible. Write more parser generator scripts for input languages, or find a way to parameterise families of input languages.	Output code; parser generator scripts
9–10	Implement and test the constraint handler, and design a set of constraints. Make the error messages as helpful as is feasible. Write the progress report	Code for the constraint handler; transcript of test interactions; progress report
11–12	Finish anything that has fallen behind, or work on the extension of a grammar specification language.	
13–14	The same continued.	

Week	Work to be done	Deliverables
15–16	Write up; include a demonstration on a real piece of music.	
17–18	Write up	
19–20	Write up	Dissertation

In the event of serious infrastructure or social disruption as a result of the Year 2000 problem, it may not be possible to keep to this schedule. However, I suspect that, if such an event occurs and lasts a significant length of time, the University will have no choice but to temporarily close down, and the submission deadlines will probably be postponed.