Music Research Using Humdrum

A USER'S GUIDE

David Huron

NOTICE

This *User Guide* is a draft manuscript only. Several chapters are incomplete and further revisions are planned for the entire manuscript.

Please note that this *User Guide* describes Humdrum Version 2.3. Public release of this version is not expected before late 1999. Some users have access to Humdrum Version 2.2, but many users will have access only to Version 1.0. Please note that some older tools do not work precisely as described in this Guide. The following commands are available only with Humdrum Version 2.3: **db**, **hum**, **melac**, **ms**, **text** and the **-t** option for **yank**. Version 1.0 of the **simil** command has a known bug. In addition, the former **fill** command has been renamed **ditto**.

Comments, corrections and suggestions are welcome concerning this *User Guide*. Please send your remarks to:

David Huron (huron.1@osu.edu)
School of Music
1866 College Road,
Ohio State University
Columbus, OH 43210
U.S.A.

Music Research Using Humdrum

A USER'S GUIDE

David Huron

Preface

The sad truth about research is that many good scholarly projects are dominated by mindless repetitive tasks. Throughout history, scholars of exceptional intelligence and training have often appeared to squander their talent and time on activities that require only moderate intelligence and little training. Unfortunately, even simple mechanical tasks are often impossible to delegate to other people because the sheer mindlessness of the work can lead to carelessness in those who don't share a passion for the underlying research problem. At the same time, otherwise gifted scholars have occasionally failed to engage in the sort of "drudge work" they know is needed to support their claims. This understandable human failing has sometimes led to abstract theorizing that is only tenuously connected to the objects of study. In addition, many otherwise promising research ideas have been abandoned due to the discouraging volume of work entailed.

This book is about making music scholarship easier and more rigorous. It is no secret that computers have engendered a revolution in productivity for researchers in the sciences. With the increasing availability of large databases, a comparable revolution is now under way in the arts and humanities disciplines. Coupled with large musical databases, *Humdrum* provides resources for increasing the productivity of individual music scholars.

This book is intended to develop the reader's facility with Humdrum. The book is organized in a tutorial format with hundreds of examples ranging from text rhythms in Gregorian chant to rock-guitar fingerings. A wide range of applications are discussed from different periods, styles, and cultures. In addition to describing specific procedures, overall research strategies are also discussed. By the end of the book, attentive readers will have become 'power users' — able, for example, to answer detailed questions pertaining to Beethoven's orchestration in less than an hour.

Arts and humanities scholars rarely have access to the sorts of financial support that is common in the sciences. As a result, arts and humanities scholars seldom have the luxury of delegating tasks: for the most part, we have to do it all ourselves. My hope is that Humdrum will help to redress this imbalance of power, and provide music scholars with tools that genuinely contribute to their personal productivity.

David Huron
Columbus, Ohio

Acknowledgements

The first draft of this book was written while I was a visiting scholar at the Center for Computer Assisted Research in the Humanities at Stanford University. I am grateful to the Center's Director, Dr. Walter Hewlett, for the invitation, and for support and encouragement in writing this guide.

While developing Humdrum, I have benefitted over the years from the advice and critical support of many friends, colleagues, students, research associates and correspondents. I am indebted to Keith Orpen, Keith Mashinter, Jasba Simpson, Sandra Serafini, Randall Howard, Simon Clift and Maki Ishizaki, (of the University of Waterloo), Timothy Prime (Wilfrid Laurier University), Eleanor Selfridge-Field and Frances Bennion (Stanford University), Andreas Kornstädt (University of Hamburg), Bo Alphonce, Bruce Pennycook and Bruce Minorgan (McGill University), Paul von Hippel, Bret Aarden and Peter Pfordresher (Ohio State University), Perry Roland (University of Virginia), Michael Taylor (University of Belfast), Gregory Sandell (Loyola University), Carol Krumhansl (Cornell University), and Jordi Martin (UPF, Spain). I am especially indebted to my former research assistants Tim Racinsky and Kyle Dawkins for their superb programming efforts. My very warm thanks to each of these individuals.

How To Use This Book

The purpose of this book is to help you become a fluent and effective user of Humdrum. The book is intended to be read from the beginning to the end — since later materials assume knowledge of earlier chapters.

Humdrum is closely linked with software tools associated with the UNIX operating system. Although Humdrum runs on several different computer systems, some familiarity with UNIX is assumed. Specifically, readers should know how to log-in, how to edit text files, and how to rename and delete files. A number of popular books are available that cover these basic tasks.

Learning how to use the Humdrum software tools may be likened to learning the vocabulary of a new language. Initially, a small vocabulary won't allow you to say very much. But as your vocabulary expands, you can use previously learned words in entirely novel sentences. Similarly, the full power of the tools discussed in the initial chapters won't become clear until they are used in conjunction with tools introduced in later chapers.

You may be tempted to skip over certain chapters. For example, much of Chapter 18 discusses the representation of music for fretted instruments. For many readers this discussion will be of marginal interest. However, there are important lessons from this discussion that generalize to a wide variety of circumstances unrelated to fretted instruments, and later chapters will build on these lessons. Similarly, Chapter 34 ostensibly deals with the analysis of 12-tone and serial music. Again, some readers may be tempted to skip this chapter. However, the tools and procedures introduced in this chapter turn out to be useful well beyond this relatively narrow repertory. For example, a dance scholar interested in studying historical ballet choreographies would be ill-advised to skip either of chapter 18 or 34.

In general, the discussions alternate between descriptions of computer-based music representations and computer-based manipulation of these representations. Not all of the pre-defined Humdrum representations are discussed in this book, nor are all of the commands or options identified. For complete technical documentation of representations and commands, readers should refer to the *Humdrum Reference Manual.*

In the concluding chapters, the book offers some advice about pursuing systematic music research. We will point to several possible pitfalls in computer-assisted musicology and suggest ways to negotiate around these pitfalls. Not all questions of musical importance can be addressed using computer technology. However, a large number of musically useful problems can indeed be profitably pursued using the tools and methods described in this book.

In addition to a bibliography and appendices, the book provided three indexes. The *General Index* permits readers to search for specific topics, features or commands. The *Name Index* allows readers to find references to particular people, works, and musical genres. The *Problem Index* allows readers to look-up passages that discuss particular musical problems or questions.

This book describes the Humdrum Toolkit Release 2.0. Humdrum is available free of charge and may be downloaded via the internet. Refer to the following web site for details: `http://www.lib.virginia.edu/dmmc/Music/Humdrum/`

Contents

Chapter 1

Humdrum: A Brief Tour

Humdrum is a general-purpose software system intended to assist music researchers. Humdrum's capabilities are quite broad, so it is difficult to describe concisely what it can do. This chapter provides a brief tour through Humdrum: the goal is to give you a quick glimpse of Humdrum's capabilities and to help sketch some of the big picture. Don't worry if you don't understand everything in this chapter. All of the topics mentioned here will be covered more thoroughly in later chapters.

What Can Humdrum Do?

Although Humdrum facilitates exploratory investigations, it is best used when the user has a clear problem or question in mind. For example, Humdrum allows users to pose and answer questions such as the following:

- In the music of Stravinsky, are dissonances more common in strong metric positions than in weak metric positions?
- In Urdu folk songs, how common is the so-called "melodic arch" — where phrases tend to ascend and then descend in pitch?
- What are the most common fret-board patterns in guitar riffs by Jimi Hendrix?
- Which of the Brandenburg Concertos contains the B-A-C-H motif?
- Which of two different English translations of Schubert lyrics best preserves the vowel coloration of the original German?
- Did George Gershwin tend to use more syncopation in his later works?
- After the V-I progression, which harmonic progression is most apt to employ a suspension?
- How do chord voicings in barbershop quartets differ from chord voicings in other repertories?
- In what harmonic contexts does Händel double the leading-tone?

Although the Humdrum tools may take just seconds to compute an answer for any of the above problems, only a few of these questions are *easy* to answer using Humdrum. The primary impedi-

ment to a quick solution is the user's skill in interconnecting the right tools for the task at hand. The purpose of this book is to help you develop that skill.

The Humdrum Syntax and the Humdrum Toolkit

Humdrum consists of two distinct components: the *Humdrum Syntax* and the *Humdrum Toolkit.* The *Humdrum Syntax* is a grammar for representing information. Over the course of this book we'll encounter a variety of representation schemes that conform to this syntax. The syntax itself doesn't represent anything: it merely provides a common framework for representing all sorts of information. Within the syntax an endless number of representation schemes can be defined. Theoretically, any type of sequential symbolic data may be accommodated — such as medieval square notation, MIDI data, acoustic spectra, piano fingerings, changes of emotional states, Indian tabla notations, dance steps, Schenkerian graphs, concert programs — or even industrial chemical processes.

A number of representation schemes are pre-defined in Humdrum; however, users are free to concoct their own task-specific representations as necessary. For example, if you are interested in Telugu notation or Dagomba dance, you can design your own pertinent representation scheme within the Humdrum syntax. Humdrum representations may be meticulously crafted, or they may be invented in a matter of seconds. It is common to generate intermediate or "throw-away" representations that are used only for a single research task.

The *Humdrum Toolkit* is a set of more than 70 inter-related software tools. These tools manipulate ASCII data (text) conforming to the Humdrum syntax. If the ASCII data represents music-related information, then we can say that the Humdrum tools manipulate music-related information.

What kinds of manipulations can be done with Humdrum? The various Humdrum tools can be grouped roughly into the following sixteen types of operations.

1. **Visual display.** E.g., display a score beginning at measure 128; output the libretto from Act II, Scene 5; print the string parts for the Coda — including the Roman numeral harmonic analysis.
2. **Dural display.** E.g., play the bass trombone part slowly beginning at measure 70; play just the opening two measures from all of the works in a given repertory.
3. **Searching.** E.g., search for instances of a motive; locate any deceptive cadences; find all of the works that are composed for a given combination of traditional Japanese instruments.
4. **Counting.** E.g., how often do augmented intervals occur in Hungarian folk songs? What proportion of phrases do not begin with a pick-up or anacrusis?
5. **Editing.** E.g., change all up-stems to down-stems in measure 88 of the second horn part.
6. **Editorializing.** E.g., add an editorial footnote to a specified note; indicate that a passage differs from the composer's autograph.
7. **Transforming or translating between representations.** E.g., transpose from one key to another; calculate the harmonic intervals between two parts; represent a score according to scale-degrees; reconstitute chords so they are represented in set *normal form.*

8. **Arithmetic transformations of representations.** E.g., calculate the semitone spacings between successive notes, or determine points where parts cross in pitch.
9. **Extracting or selecting information.** E.g., extract the second verse; exclude the development section; isolate the third phrase; grab the second chord in each measure; select the brass parts; take the second endings when repeating the trio; choose the Dresden manuscript version.
10. **Linking or joining information.** E.g., assemble instrumental parts into a full score; tag notes with their harmonic function; coordinate heart-rate data from a listener with the musical score.
11. **Generating inventories.** E.g., list all the types of embellishment (non-chordal) tones from the most common to the least common; what chord functions are absent from a work?
12. **Classifying.** E.g., classify all chords as "open" or "closed" position; identify all secondary dominants; classify all intervals as either unisons, steps or leaps; classify various piano fingerings as either easy, moderate, difficult, or impossible.
13. **Labelling.** E.g., mark musical sections; label themes; identify French, Italian and German sixth chords; mark appropriate words in a vocal text as either "passionate," "apathetic," or "neutral." Mark sonorities as falling on either strong or weak metric positions.
14. **Comparison.** E.g., determine whether the Amsterdam and Manchester manuscripts for a work have identical pitches in the third movement; determine whether motets by John Dunstable are more similar to motets by Thomas Morley or by Lionel Power.
15. **Capturing Data.** E.g., import live or recorded MIDI data; import data from a notation program.
16. **Trouble-shooting.** E.g., identify any transgressions of notational conventions; check whether a score has been tampered with; get help when you're stuck.

If the above functions sound vague, that's because the corresponding Humdrum tools are similarly broad in their function. For example, the Humdrum **patt** tool can be used to locate Landini cadences, label statements of 12-tone rows, or search for piano fingering patterns in Liszt. All three tasks involve searching for a sequential pattern that might have concurrent features as well.

When using computers in research activities, it is important to maintain a keen awareness of their limitations. For example, computers are typically not good at *interpreting* data. Because much music scholarship hinges on interpretations, this would seem to preclude computers from being of much use. However, there are some things that computers do well, and when a skilled user intervenes to make a few crucial interpretations, the resulting computer/human interaction can lead to pretty sophisticated results — as we will see.

Although computers are typically not good at interpreting data, interpretive software will become increasingly important as computational musicology continues to develop. Humdrum provides a handful of tools that *interpret* data in various ways. For example, the **key** command implements the Krumhansl and Kessler method for estimating the key of a music passage. The **melac** command implements Joseph Thomassen's model of melodic accent. Other interpretive tools characterize syncopation or implement Johnson-Laird's model of rhythmic prototypes.

The names of some of the Humdrum tools will be readily recognizable by musicians. Humdrum

tools such as **key, pitch, record, tacet, trans** and **reihe** may evoke fairly accurate ideas about what they do. Ironically, the most recognizable tools are typically the least useful tools in the toolkit — because they are so specialized. The most powerful Humdrum tools have names such as **cleave, humsed, simil, recode, context, patt** and **yank.**

By itself, each individual tool in the Humdrum Toolkit is quite modest in its effect. However, the tools are not intended to be self-sufficient. They are designed to work in conjunction with each other, and with existing standard UNIX commands. Like musical instruments, their potential usefulness is greatly increased when they are combined with other tools. Musical problems are typically addressed by linking together successive Humdrum (and UNIX) commands to form one or more command *pipelines.* Although each individual tool may have only a modest effect, the tools' combined capacity for solving complex problems is legion.

Now that we've sketched an overview of Humdrum, we can consider in greater detail the two principal components of Humdrum: the *Humdrum Syntax* and the *Humdrum Toolkit.*

Humdrum Syntax

Humdrum data are organized somewhat like the tables of a spread-sheet. As with a spread-sheet, you can define or label your own types of data. As with a spread-sheet, the different *columns* can be set up to represent whatever type of data you like. The *rows* of data, however, have a fixed meaning — they represent successive moments in time; that is, time passes as you move down the page.

By way of example, consider Table 1.1. This table shows something akin to a piano-roll. The diatonic pitches are labelled in columns from C4 (middle C) to C5. Each row represents the passage of a sixteenth duration. The table entries indicate whether the note is *on* or *off.* The table encodes an ascending scale. As the table stands, it's not clear whether each note has a duration of an eighth note or whether two successive sixteenth notes are sounded for each pitch:

Table 1.1

		C4	**D4**	**E4**	**F4**	**G4**	**A4**	**B4**	**C5**
	1st 16th	ON	off	off	off	off	off	off	off
	2nd 16th	ON	off	off	off	off	off	off	off
	3rd 16th	off	ON	off	off	off	off	off	off
	4th 16th	off	ON	off	off	off	off	off	off
time	5th 16th	off	off	ON	off	off	off	off	off
	6th 16th	off	off	ON	off	off	off	off	off
↓	7th 16th	off	off	off	ON	off	off	off	off
	8th 16th	off	off	off	ON	off	off	off	off
	9th 16th	off	off	off	off	ON	off	off	off
	10th 16th	off	off	off	off	ON	off	off	off
	11th 16th	off	off	off	off	off	ON	off	off
	etc.								

Table 1.2 shows another example where different kinds of information are combined in the same table. Here the last column represents a combination of trumpet valves:

Table 1.2

	Pitch	Duration	Valve Combination
1st note	C4	quarter	0
2nd note	B3	eighth	2
3rd note	G4	eighth	0
4th note	F4	eighth	1
5th note	G4	eighth	0
6th note	A4	quarter	1-2
7th note	G4	eighth	0
8th note	Ab4	quarter	2-3

Humdrum representations can be very similar to the data shown in Tables 1.1 and 1.2. With just a few formatting changes, either table can be transformed so that it conforms to the Humdrum syntax. For example, Table 1.3 recasts Table 1.2 so that it conforms to the Humdrum syntax. Just four changes have been made: (1) the left-most column has been given a heading name so that *every* column has a label, (2) each column heading is preceded by two asterisks, (3) the columns have been left-justified so successive columns are separated by a tab, and (4) each column has been terminated with the combination of an asterisk and hyphen.

Table 1.3 A Humdrum Equivalent to Table 1.2

```
**Note      **Pitch    **Duration    **Valve Combination
1st note    C4         quarter       0
2nd note    B3         eighth        2
3rd note    G4         eighth        0
4th note    F4         eighth        1
5th note    G4         eighth        0
6th note    A4         quarter       1-2
7th note    G4         eighth        0
8th note    Ab4        quarter       2-3
*-          *-         *-            *-
```

It does not matter what characters appear in the table — numbers, letters, symbols, etc. (although there are some restrictions concerning the use of spaces and tabs). The table can have as many columns as you like, and can be as long as you like. Unlike spreadsheet columns, Humdrum "columns" can exhibit complicated "paths" through the document; columns can join together, split apart, exchange positions, stop in mid-table, or be introduced in mid-table. Humdrum also allows subsidiary column headings that can clarify the state of the data. Subsidiary headings can also be added anywhere in mid-table. Finally, Humdrum also provides ways of adding running commentaries; comments might pertain to the whole table, to a given row or column, to a given data cell, or to a particular item of information within a cell (such as a single letter or digit). Since the *columns* in Humdrum data can roam about the table in a semi-flexible way, they are referred to as **spines.** We'll see how these devices are used in later chapters.

The most common Humdrum files encode musical notes in the various cells of the table; the most common use of a spine is to represent a single musical part or instrument.

Some twenty or more representation schemes are pre-defined in Humdrum, but remember that users are always free to concoct their own representations as necessary. As in a spread-sheet, the

spines or columns can be used to represent whatever you like. In the following chapter, we'll look at the most commonly used of the pre-defined Humdrum representations — the *kern* representation. This representation gets its name from the German word for *core*; it is a scheme intended to represent the basic or core musical information of notes, durations, rests, barlines, and so on.

Humdrum Tools

The Humdrum software is not a program that you invoke like a word-processor or notation editor. Humdrum is not a big program that you start-up when you want to do music research. Instead, Humdrum provides a toolbox of *utilities* — most of which can be accessed at any time from anywhere in the system.

Any data that conforms to the Humdrum syntax can be manipulated using the Humdrum software tools. Since the tools can be interconnected with each other (and can also be interconnected with non-Humdrum tools) there are a lot of ways to manipulate Humdrum data. Much of this book will deal with how the tools can be interconnected to do musically useful things.

For the remainder of this chapter, we will describe a few Humdrum tools and illustrate how they might be used. Once again, the goal in this chapter is to give you an initial taste of Humdrum. Don't worry if you don't understand everything at this point.

Some Sample Commands

One group of tools is used to extract or select sections of data. Vertical spines of data can be extracted from a Humdrum file using the **extract** command. For example, if a file encodes four musical parts, then the **extract** command might be used to isolate one or more given parts. The command

```
extract -f 1 filename
```

will extract the first or left-most column or spine of data. Often it is useful to extract material according to the encoded content without regard to the position of the spine. For example, the following command will extract all spines containing a label indicating the tenor part(s).

```
extract -i '*Itenor' filename
```

Instruments can be labelled by "instrument class" and so can be extracted accordingly. The following command extracts all of the woodwind parts:

```
extract -i '*ICww' filename
```

Any vocal text can be similarly extracted:

```
extract -i '**text' filename
```

Or if the text is available in more than one language, a specific language may be isolated:

```
extract -i '*LDeutsch' filename
```

Segments or passages of music can be extracted using the **yank** command. Segments can be defined by sections, phrases, measures, or other any user-specified marker. For example, the following command extracts the section labelled "Trio" from a minuet & trio:

```
yank -s Trio -r 1 filename
```

Or select the material in measures 114 to 183:

```
yank -n = -r 114-183 filename
```

Or select the second-last phrase in the work:

```
yank -o { -e } -r '$-1' filename
```

Don't worry about the complex syntax for these commands; the command formats will be discussed fully in the ensuing chapters. For now, it is important only that you get a feel for some of the types of operations that Humdrum users might perform.

Two or more commands can be connected into a *pipeline.* The following command will let us determine whether there are any notes in the bassoon part:

```
extract -i '*Ifagot' filename | census -k
```

The following pipeline connects together four commands: it will play (using MIDI) the first and last measures from a section marked "Coda" at half the notated tempo from a file named `Cui`:

```
yank -s '*>Coda' Cui | yank -o ^= -r 1,$ | midi | perform -t .5
```

Some tools translate from one representation to another. For example, the **mint** command generates melodic interval information. The following command locates all tritones — including compound (octave) equivalents:

```
mint -c filename | egrep -n '((d5)|(A4))'
```

Incidentally, Humdrum data can be processed by many common commands that are not part of the Humdrum Toolkit. The **egrep** command in the above pipeline is a common computer utility and is not part of the Humdrum Toolkit.

Depending on the type of translation, the resulting data can be searched for different things. The following command identifies French sixth chords:

```
solfa file | extract -i '**solfa' | ditto | grep '6-.*4+' | grep 2
```

Locate all sonorities in the music of Machaut where the seventh scale degree has been doubled:

```
deg -t machaut* | grep -n '7[^-+].*7'
```

Count the number of phrases that end on the subdominant pitch:

```
deg filename | egrep -c '(}.*4)|(4.*})'
```

The following command identifies all scores whose instrumentation includes a tuba but not a trumpet:

```
grep -sl '!!!AIN.*tuba' * | grep -v 'tromp'
```

Some tasks may require more than one command line. For example, the following three-line script locates any parallel fifths between the bass and alto voices of any input file:

```
echo P5 > P5
echo '=       *' >> P5; echo P5 >> P5
extract -i '*Ibass,*Ialto' file | hint -c | pattern -s = P5
```

More complicated scripts can be written to carry out more sophisticated musical processes. In later chapters we'll encounter some scripts that contain 10 or more lines of commands.

Reprise

In this chapter we have seen that Humdrum consists of two parts: (1) a representation *syntax* that is similar to tables in a spread-sheet, and (2) a set of utilities or *tools* that manipulate Humdrum data in various ways. The tools carry out operations such as displaying, performing, searching, counting, editing, transforming, extracting, linking, classifying, labelling and comparing. The tools can be linked together to carry out a wide variety of tasks. Even one-line commands can carry out sophisticated operations. A few lines of Humdrum can also be used to write programs of some complexity. Users can write their own programs using the Humdrum tools, or they can add new tools that augment the functioning of Humdrum.

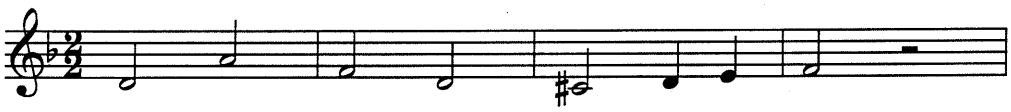
Chapter 2

Representing Music Using ***kern* (I)

The `**kern` representation can be used to represent basic or core information for common Western music. The `**kern` scheme can be used to encode pitch and duration, plus other common score-related information. In this chapter, we will introduce `**kern` through a series of tutorial examples. A more comprehensive description of `**kern` will be given in Chapter 6. Our intention here is to provide a quick introduction.

Consider the opening motive from J.S. Bach's *Art of Fugue* shown in Example 2.1. A corresponding `**kern` representation is given below the notation.

Example 2.1. J.S. Bach *Die Kunst der Fuge*



```
**kern
*clefG2
*k[b-]
*M2/2
=-
2d/
2a/
=
2f/
2d/
=
2c#/
4d/
4e/
=
2f/
2r
*-
```

In general, `**kern` is intended to represent the underlying *functional* information conveyed by a musical score rather than the visual or *orthographic* information found in a given printed rendition. The `**kern` representation is designed to facilitate analytic applications rather than music printing or sound generation. Nevertheless, both visual output and sound output can be generated from the `**kern` representation.

Notice that whereas the notation is laid out horizontally across the page, the Humdrum representation procedes vertically down the page. The representation begins with the keyword `**kern` which indicates that the ensuing encoded material conforms to the kern representation. The encoded passage ends with a special *terminator* token (`*-`).

The clef is identified as a G-clef positioned on the second line of the staff (i.e., a treble clef). The key signature is encoded as a single flat — B-flat: the minus sign here indicates a flat. The meter signature (2/2) is encoded next.

Lines that begin with an equals-sign indicate *logical* barlines. Musical works may begin with either a complete measure or a partial measure. In the `**kern` representation, the beginning of the first measure is explicitly indicated. In Example 2.1, the encoding for the first barline (`=-`) is a functional encoding that doesn't correspond to anything in the printed score. The minus sign following the equals-sign indicates that the barline is "invisible." The presence of this logical barline allows various Humdrum tools to recognize that the initial notes start at the beginning of the first measure — and are not "pick-up" notes prior to the first measure.

The durations of the notes are indicated by reciprocal numbers: 1 for whole-note, 2 for half-note, 4 for quarter-note, etc. The *breve* or double whole-note is a special case and is represented by the number zero.

Stem directions are encoded using the slash (/) for up-stems and the back-slash (\) for down-stems.

Pitches are represented through a scheme of upper- and lower-case letters. Middle C (C4) is represented using the single lower-case letter "`c`". Successive octaves are designated by letter repetition, thus C5 is represented by "`cc`", C6 by "`ccc`" and so on. The higher the octave, the more repeated letters.

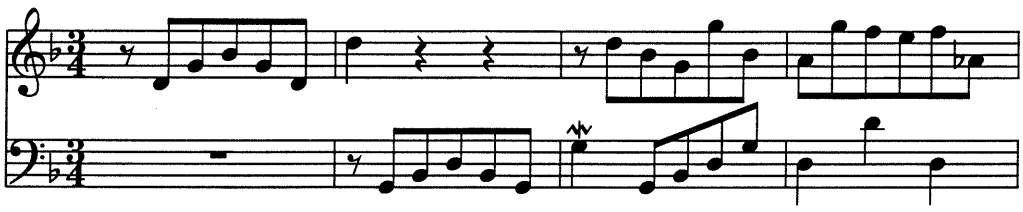
For pitches below C4, upper-case letters are used: "`C`" designates C3, "`CC`" designates C2, and so on. Changes of octave occur between B and C. Thus the B below middle C is represented as "`B`"; the B below "`CC`" is represented as "`BBB`", and so on. The lower the octave, the greater the number of repeated letters.

Accidentals are encoded using the octothorpe (`#`) for sharps, the minus sign (`-`) for flats, and the lower-case letter "`n`" for naturals. Accidentals are encoded immediately following the pitch letter name. Double-sharps and double-flats have no special representations in `**kern` and are simply denoted by repetition: (`##`) and (`--`) respectively. Triple- and quadruple accidentals are similarly encoded by repetition. Sharps, flats, and naturals are mutually exclusive in `**kern`, so tokens such as "`cc#n`" and "`GG-#`" are illegal.

In Example 2.2, two musical parts are encoded. In the corresponding `**kern` encoding, each musical part or voice has been assigned to a different musical staff — labelled `*staff1` and `*staff2`. Notice that the upper part has been encoded in the right-most column. The layout is

exactly as though the musical score were turned sideways.

Example 2.2. J.S. Bach, *Praeambulum* BWV 390.



```
**kern    **kern
*staff2   *staff1
*clefF4   *clefG2
*k[b-]    *k[b-]
*M3/4     *M3/4
=1-       =1-
2.r       8r
.         8d/L
.         8g/
.         8b-/
.         8g/
.         8d/J
=2        =2
8r        4dd\
8GG/L     .
8BB-/     4r
8D/       .
8BB-/     4r
8GG/J     .
=3        =3
4GWw\     8r
.         8dd\L
8GG/L     8b-\
8BB-/     8g\
8D/       8gg\
8G/J      8b-\J
=4        =4
4D\       8a/L
.         8gg/
4d\       8ff/
.         8ee/
4D\       8ff/
.         8a-/J
=5        =5
*-        *-
```

Each column contains its own separate information. Both columns have been labelled `**kern` and are terminated (`*-`). The columns are separated by a single tab. Barline information is also

encoded in each column. Notice that measure numbers have been added following the barline indicator (=). Although measure numbers may not be present in the printed score, it is normal to include them in `**kern` encodings.

A notable feature when representing multi-part music is the presence of place holders called *null tokens*. A null token is represented by a single isolated period character (.). Not all musical parts will have a new note with each successive sonority. A null token maintains the grid structure for a sustained pitch while another part is moving.

Rests are encoded by the lower-case letter 'r'. Notice that the first rest in the lower part has not been encoded as a whole rest. Instead, it has been rendered as a dotted half rest. This is a good illustration of how `**kern` is intended to be a *functional* rather than *orthographic* (visual) representation.

Another important difference between functional and orthographic representations is evident in the treatment of accidentals. In the `**kern` representation, all pitches are encoded without regard for what is going on around them. For example, in `**kern`, pitches are encoded with the appropriate accidental, even if the accidental is specified in a key-signature or is present earlier in the same measure. Hence the explicit encoding of all occurrences of B-flats in Example 2.2.

All pitches are encoded as absolute pitches. In `**kern`, even transposing instruments are always represented at (sounding) concert pitch. A special *transposition interpretation* is provided to indicate the nature of any transposing instrument — but the encoded pitches themselves appear only at concert pitch.

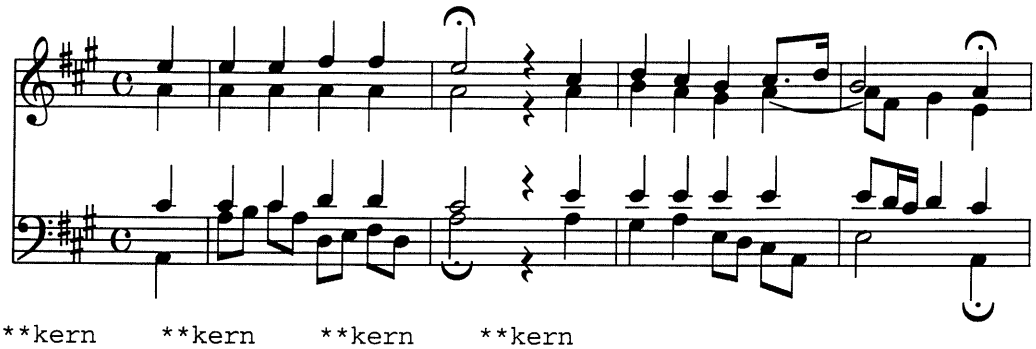
Finally, pitches in `**kern` are encoded as "nominally" equally-tempered values. A special *temperament interpretation* is provided to indicate if the tuning system is other than equal temperament.

Two other aspects of Example 2.2 are noteworthy. In measure 3 an inverted mordent appears on the first note in the bass part. Mordents are encoded via the letter 'M' — upper-case 'M' for whole-tone mordents and lower-case 'm' for semitone mordents. Inverted mordents are similarly encoded using the letter 'W'. In Example 2.2, it is not clear whether the mordent should be a semitone (consistent with a G harmonic minor figure) or a whole-tone mordent (consistent with a G melodic minor figure). The signifier 'Ww' is a special representation indicating that the ornament may be either one.

Also evident in Example 2.2 are the upper-case letters 'L' and 'J'. These encode beaming information. Each beam is opened by the letter 'L' and closed by the letter 'J' (think of left and right angles). Multiple beams and partial beams are discussed in Example 2.4.

Example 2.3 shows a four-part chorale harmonization by Bach. In this example, two musical parts share each of the two staves. Notice how the corresponding `**kern` encoding assigns each part to a separate column, but links the appropriate parts using the `*staff` indicators.

Example 2.3. *Nun danket alle Gott*, arr. J.S. Bach.



```
**kern      **kern      **kern      **kern
*staff2     *staff2     *staff1     *staff1
*clefF4     *clefF4     *clefG2     *clefG2
*k[f#c#g#]*k[f#c#g#]*k[f#c#g#]*k[f#c#g#]
*M4/4       *M4/4       *M4/4       *M4/4
4AA         4c#         4a          4ee
=1          =1          =1          =1
8A          4c#         4a          4ee
8B          .           .           .
8c#         4c#         4a          4ee
8A          .           .           .
8D          4d          4a          4ff#
8E          .           .           .
8F#         4d          4a          4ff#
8D          .           .           .
=2          =2          =2          =2
2A;         2c#;        2a;         2ee;
4r          4r          4r          4r
4A          4e          4a          4cc#
=3          =3          =3          =3
4G#         4e          4b          4dd
4A          4e          4a          4cc#
8E          4e          4g#         4b
8D          .           .           .
8C#         4e          [4a         8.cc#
8AA         .           .           .
.           .           .           16dd
=4          =4          =4          =4
2E          8e          8a]         2b
.           16d         8f#         .
.           16c#        .           .
.           4d          4g#         .
4AA;        4c#;        4e;         4a;
=:|!        =:|!        =:|!        =:|!
*-          *-          *-          *-
```

Once again, clefs, key signatures and meter signatures are encoded separately for each part. Notice how the meter signature has been encoded as 4/4 rather than 'common time'. This again re-

flects `**kern`'s preoccupation with functional information rather than orthographic information. (Later we will see how to encode the fact that the meter signature is visually rendered as 'C' rather than 4/4.)

In measure 2, pauses (`;`) have been encoded for all four voices even though only two pause symbols appear in the printed notation.

In the third measure, the last note of the alto part has been tied into the fourth measure. The `**kern` representation provides no generic means for representing "curved lines" found in printed scores. Since `**kern` is a "functional" rather than an "orthographic" representation, all lines are explicitly interpreted as either *ties, slurs* or *phrases.*

The open brace `{` denotes the beginning of a phrase. The closed brace `}` denotes the end of a phrase.

The open parenthesis `(` and closed parenthesis `)` signify the beginning and end of a slur respectively.

The open square bracket `[` denotes the first note of a tie. The closed square bracket `]` denotes the last note of a tie. The underscore character `_` denotes middle notes (if any) of a tie.

Slurs and phrase markings can be *nested* (e.g. slurs within slurs) and may also be *elided* (e.g. overlapping phrases) to a single depth. *Nested markings* mean that one slur or phrase is entirely subsumed under another slur or phrase. For example: `( ( ) )` means that a short slur has occurred within a longer slur. *Elisions* are overlaps, for example, where an existing phrase fails to end while a new phrase begins. In `**kern` the ampersand character (`&`) is used to mark elided slurs or phrases. For example: `{ &{ } &}` means that two phrases overlap — the first phrase ending after the second phrase has begun.

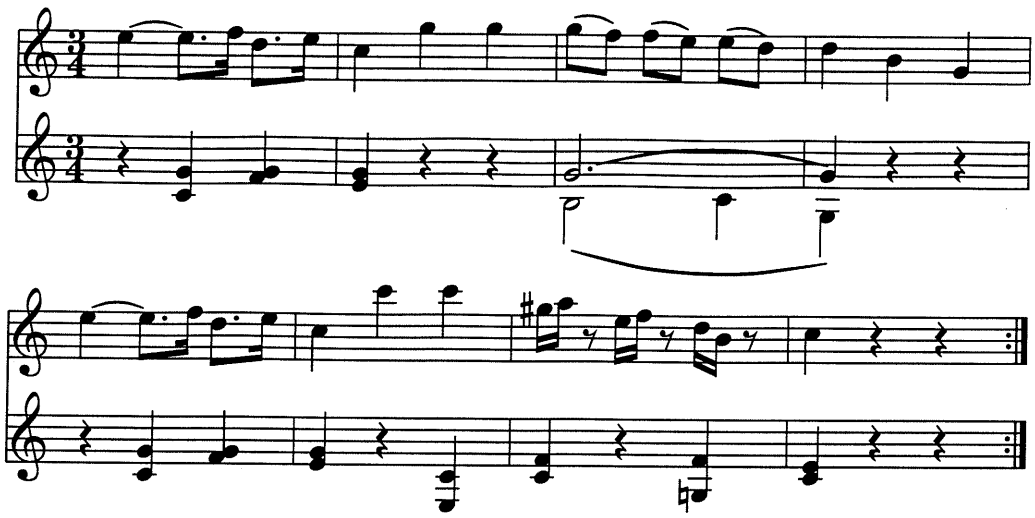
Example 2.3 ends with a repeat sign. The `**kern` representation makes a distinction between repeat signs that appear in the score and repeat signs that are obeyed in performance. In this example, only the visual or orthographic rendering of the barline has been encoded. Later, in Chapter 20, we will see how sectional repeats are functionally represented. The visual appearance of the final barline is encoded as follows: `=:|!`. The equals-sign indicates the logical presence of a barline. The colon indicates the repeat sign, followed by a thin line (`|`), followed by a thick line (`!`).

Example 2.4 shows a keyboard work by Franz Joseph Haydn. In this example, the lower staff appears to have two concurrent voices. The excerpt has been encoded using three spines, two of which encode material appearing on the same staff. Notice that key signatures are provided that explicitly indicate that there are no sharps or flats in the key signature.

The upper-most part in the first measure shows the use of partial beams. Partial beams that extend to the left are encoded by the lower-case letter '`k`'. Partial beams that extend to the right are encoded by the upper-case '`K`'. Letters are repeated for each partial beam present. In this case, only a single partial beam is used, so only a single '`k`' is encoded. Notice that the signifiers '`L`' and '`J`' are used only to encode complete beams.

In the second-last measure, double beams are used to join the sixteenth-note pairs. As a result, the beams are started with '`LL`' and end with '`JJ`'.

Example 2.4. Franz Joseph Haydn, *Sonata in C major, Hob. XVI: 35.*



```
**kern    **kern    **kern
*staff2   *staff2   *staff1
*clefG2   *clefG2   *clefG2
*k[]      *k[]      *k[]
*M3/4     *M3/4     *M3/4
=1-       =1-       =1-
4r        4r        [4ee\
4c/       4g/       8.ee]\L
.         .         16ff\Jk
4f/       4g        8.dd\L
.         .         16ee\Jk
=2        =2        =2
4e/       4g/       4cc\
4r        4r        4gg/
4r        4r        4gg/
=3        =3        =3
(2B\      ([2.g/    (8gg\L
.         .         8ff)\J
.         .         (8ff\L
.         .         8ee)\J
4c\       .         (8ee\L
.         .         8dd)\J
=4        =4        =4
4G\)      4g])/     4dd\
4r        4r        4b\
4r        4r        4g/
=5        =5        =5
4r        4r        [4ee\
4c/       4g/       8.ee]\L
.         .         16ff\Jk
4f/       4g/       8.dd\L
```

```
.           .           16ee\Jk
=6          =6          =6
4e/         4g/         4cc\
4r          4r          4ccc\
4E\         4c\         4ccc\
=7          =7          =7
4f\         4c\         16gg#\LL
.           .           16aa\JJ
.           .           8r
4r          4r          16ee\LL
.           .           16ff\JJ
.           .           8r
4Gn\        4f\         16dd\LL
.           .           16b\JJ
.           .           8r
=8          =8          =8
4c\         4e\         4cc\
4r          4r          4r
4r          4r          4r
=:|!        =:|!        =:|!
*-          *-          *-
```

Slurs are evident in the third measure. Open and closed slurs are represented by open (() and closed ()) parentheses respectively. Notice that the middle part in the third measure contains a tied note. Considering the presence of concurrent slurs in the other parts it is possible that an appropriate interpretation of the score would regard the middle voice as also slurred. Notice that in the `**kern` representation, notes can be tied, slurred, and phrased concurrently.

Comment Records

In any representation, some information may best be conveyed as an appended commentary, rather than as part of the encoded data. Humdrum comments are records (lines) that begin with an exclamation mark.

Humdrum distinguishes two basic types of comments. Comments that pertain to all spines in a file are referred to as *global comments* and begin with two exclamation marks (`!!`). Comments that pertain to a single spine are called *local comments* and begin with a single exclamation mark in each spine. Both types of comments are evident in Example 2.5.

The first three records are global comments identifying the source and title of the piece. The fifth record encodes a local comment in each column. One local comment identifies that the lyrics are in the Ojibway language. On the same line, notice that the other spine also encodes single exclamation mark, but contain no text. Such isolated exclamation marks are referred to as *null local comments.*

Notice that *local comments* conform to the prevailing spine structure. Each spine begins with an exclamation mark and tabs continue to demarcate each spine. *Global comments* by contrast completely ignore the spines.

Example 2.5. Ojibway Song.

```
!! Ojibway Indian Song
!! Transcribed by Frances Densmore
!! No. 84 "The Sioux Follow Me"
**kern          **lyrics
!               ! In Ojibway
*clefF4         *
*M3/4           *
*k[b-e-a-d-g-]  *
8.d-            Ma-
16d-            -gi-
=1              =1
8d-             -ja-
16A-            -go
16A-            ic-
4d-             -kew-
4d-             -yan
=2              =2
etc.            etc.
*-              *-
```

Reference Records

A particularly important type of global comment is the *reference record.* Reference records are formal ways of encoding "library-type" information pertaining to a Humdrum document. Reference records provide standardized ways of encoding bibliographic information — suitable for computer-based access.

Humdrum reference records are designated by three exclamation marks at the beginning of a line, followed by a letter code, followed by an optional number, followed by a colon, followed by some text. The following example provides a set of reference records related to the "Augurs of Spring" section from Stravinsky's *Rite of Spring.*

Example 2.6.

```
!!!COM: Stravinsky, Igor Fyodorovich
!!!CDT: 1882/6/17/-1971/4/6
!!!ODT: 1911//-1913//; 1947//
!!!OPT: Vesna svyashchennaya
!!!XFR: Le sacre du printemps
!!!XEN: Rite of Spring
!!!OTL: Les augures printaniers
!!!PUB: Boosey & Hawkes
!!!YEC: 1945 Boosey & Hawkes
!!!AGN: ballet
!!!AST: neo-classical
!!!AMT: irregular
!!!AIN: clars corno fagot flt oboe
```

Reference records need not be in any particular order. The most important reference records (composer, title, etc.) are typically placed at the very beginning of a file since this makes inspecting the file easier. Less important reference records are typically placed at the end of the file.

Reference codes that begin with the letter 'C' pertain to the composer. The 'COM' code identifies the composer (surname first followed by given names). The 'CDT' code identifies the composer's birth and death dates. A special format is used in defining such dates, and so there are accurate ways to represent uncertainty, approximation, ranges of dates, and alternative dates. The *Humdrum Reference Manual* describes date formats in great detail.

Reference codes that begin with the letter 'O' pertain to the work or opus. The 'OTL' code identifies the title of the encoded material — in this case the '*Les augures printaniers*'. The 'OPT' code identifies the 'parent' work from which the encoded music belongs. The 'ODE' code identifies the name of a person or organization to which the work was dedicated. All three of these records (OTL, OPT and ODE) are encoded using the original language.

Reference codes beginning with the letter 'X' are used for translations. The 'XEN' code is used to provide an English translation. In this case, the title (*Vesna svyashchennaya*) is translated as *Rite of Spring*.

Reference codes beginning with 'P' pertain to publishing and imprint information. (Codes beginning with 'S' can be used to identify manuscript sources, library or archive locations, and other source-related data.)

Codes beginning with 'Y' identify copyright information. Humdrum defines separate codes for publisher of the electronic edition, publisher of the original source document, date of copyright, date of data release, country of copyright, copyright message, original copyright owner, original year of publication, and other information. The 'YEC' reference record shown in Example 2.6 simply encodes the date and copyright owner of the electronic document.

Codes that begin with 'A' identify analytic information concerning the document. The code 'AMT' provides a metric classification. Meters may be classified using combinations of the following keywords: `simple, compound, duple, triple, quadruple, irregular`. The 'AGN' code is used to provide a free-form text that helps to identify the genre of the work. In this case the genre is identified as `ballet`. Other suitable characterizations may include opera, string quartet, concerto, barbershop quartet, folksong, and so on. The 'AST' code can be used to identify the style or period of work. Once again, this is a free-form text record. Suitable keywords might include terms such as baroque, bebop, bossa nova, Ecole Notre Dame, minimalist, high-life, hip-hop, reggae, etc. Such analytic information is obviously interpretive and often open to disagreement. Nevertheless, explicit analytic information often proves useful in electronic documents.

An especially useful analytic reference record is the 'AIN' record for encoding instrumentation. This reference record follows a strict syntax. Each instrument has an official Humdrum abbrevation. Appendix I identifies a number of the more common instrument codes. Instrumentation reference records always specify the instrumentation in alphabetical order by instrument abbrevation separated by a single space. For example, the instrumentation for a woodwind quintet is given as:

```
!!!AIN: clars corno fagot flt oboe
```

In our discussion here we have only identified some of the more common types of reference records. A complete description of reference records is given in Appendix II.

Reprise

In this chapter we have introduced the Humdrum `**kern` representation and a few of the more important reference records. As we have seen, `**kern` can be used to encode core information for common musical scores; `**kern` is used to represent *functional* information rather than *orthographic* (visual) information. In Chapter 6 an expanded description of `**kern` will be given that includes a much wider variety of concepts and situations than we have encountered in this chapter. Appendices I and II provide expanded information pertaining to Reference Records.

Although we have only demonstrated the encoding of fairly simple information, we can already begin processing such data in musically useful ways. In the next chapter we will examine some simple processes.

Chapter 3

Some Initial Processing

Now that we have learned some things about Humdrum representations (and the `**kern` representation in particular), let's explore some basic processing tasks.

The *census* Command

The Humdrum **census** command provides basic information about an input stream or file. We can invoke the command by typing the command-name followed by the name of a file. The command:

```
census india01.krn
```

might produce the following output:

```
HUMDRUM DATA

Number of data tokens:      91
Number of null tokens:      0
Number of multiple-stops:   0
Number of data records:     91
Number of comments:         14
Number of interpretations:  7
Number of records:          112
```

Most commands provide *options* that will modify the operation of the command in a particular way. In UNIX-style commands, options follow after the command-name and are typically specified by a single letter preceded by a hypen. The **-k** option with the **census** command will give further information pertaining to the Humdrum `**kern` representation. With the **-k** option, the output includes the number of notes in the file, the longest, shortest, highest, and lowest notes, the maximum number of concurrent notes or voices, the number of rests, and the number of barlines. For example, the command:

```
census -k india01.krn
```

might produce the following *additional* output:

```
KERN DATA

Number of noteheads:        78
Number of notes:            78
Longest note:               1
Shortest note:              16
Highest note:               cc
Lowest note:                c
Number of rests:            1
Maximum number of voices:   1
Number of single barlines: 11
Number of double barlines: 1
```

Notice that a distinction is made between the number of notes and the number of noteheads. A tied note is considered to be a single "note," although it may be notated using two or more noteheads.

The output from **census** can be restricted to a particular item of information by "piping" the output to the UNIX **grep** command.

Simple Searches using the *grep* Command

The UNIX **grep** command is a popular tool for searching for lines that match some specified pattern. Patterns may be simple strings of characters, or may be more complicated constructions defined using the UNIX *regular expression* syntax. Regular expressions will be described in detail in Chapter 9. The command-name "`grep`" is an acronym for "get regular expression."

Useful patterns are often literal character strings, such as keywords. For example, the following command identifies whether the file `opus28.krn` contains the word "`Andante`":

```
grep 'Andante' opus28.krn
```

Every line containing the specified pattern will be output. If no match is found, no output is given.

Using a single command, all files in the current directory can be searched by substituting the asterisk (shell *wildcard*) in place of the filename. The following command identifies all instances where the word "`Andante`" occurs; all files in the current directory are searched:

```
grep 'Andante' *
```

Once again, every line containing the sought pattern is echoed in the output. If more than one pattern is found, each instance of the pattern will be output on a separate line. Whenever a "wildcard" is used as part of the filename, **grep** causes the *name* of each file to be prepended to the output for all patterns that are found:

```
opus28:!! Andante
opus29:!! Andante
opus46:!! Andante
```

```
opus91:!! Andante
opus98:!! Andante
```

By default, **grep** distinguishes upper- and lower-case characters, so the above command will not match strings such as "ANDANTE". However, the **-i** option tells **grep** to ignore the case when searching. E.g.,

```
grep -i 'Andante' *
```

Sought patterns may occur in any line, including data records and comments. The following command will identify the presence of any double-sharps in the file `schumann.krn`.

```
grep '##' schumann.krn
```

Pattern Locations Using *grep -n*

If a pattern is found, it is sometimes helpful to know the precise location of the pattern. The **-n** option tells **grep** to prepend the *line number* for each matching instance. The following command identifies the line numbers for lines containing a double sharp for the file `melody.krn`:

```
grep -n '##' melody.krn
```

The output might look like this:

```
1109:{4g##
1731:16g##
3002:16f##
```

— meaning that double sharps were found in lines 1109, 1731, and 3002 in the file `melody.krn`.

Counting Pattern Occurrences Using *grep -c*

In some cases, the user is interested in counting the total number of instances of a found pattern. The **-c** option causes **grep** to output a numerical *count* of the number of lines containing matching instances. For example, in the `**kern` representation, the beginning of each phrase is marked by the presence of an open curly brace ('{'). So the following command can be used to count the number of phrases in the file `glazunov.krn`:

```
grep -c '{' glazunov.krn
```

As noted, the **grep** command will search all lines (including comments) for matching instances of the specified pattern. If a curly brace were to appear in a comment or other non-data record, then our phrase-count would be incorrect. More carefully constructed patterns require a better knowledge of *regular expressions*. Regular expressions are discussed in Chapter 9.

Searching for Reference Information

As we saw in Chapter 2, Humdrum files typically encode library-type information using reference records. For example, the composer's name is encoded in a `!!!COM:` record, and the title is encoded via the `!!!OTL:` record. In conjunction with the **grep** command, these three letter codes provide useful tags to search for pertinent information. For example, the following command will identify the composer for the file `opus24.krn`:

```
grep '!!!COM:' opus24.krn
```

The output might look like this:

```
!!!COM: Boulanger, Nadia
```

Once again, wildcards (i.e., the asterisk) can be used to address all of the files in the current directory. Hence the command:

```
grep '!!!COM:' *
```

will produce a list of all composers of files in the current directory. Similarly, the following command will generate a list of all of the titles:

```
grep '!!!OTL:' *
```

The output might look as follows:

```
foster11:!!!OTL: Oh! Susanna
foster12:!!!OTL: Jeanie with the Light Brown Hair
foster13:!!!OTL: Beautiful Dreamer
foster14:!!!OTL: Gwine to Run All Night (or 'De Camptown Race')
foster15:!!!OTL: My Old Kentucky Home, Good-Night
foster16:!!!OTL: We are Coming, Father Abraam
foster17:!!!OTL: Don't Bet Your Money on De Shanghai
foster18:!!!OTL: Gentle Annie
foster19:!!!OTL: If You've Only Got a Moustache
foster20:!!!OTL: Maggie by my Side
foster21:!!!OTL: Old Folks at Home
foster22:!!!OTL: Better Times are Coming
foster23:!!!OTL: When this Dreadful War is Ended
foster24:!!!OTL: Hard Times Comes Again No More
```

Remember that when a wildcard is used in filenames, **grep** prepends the filename prior to found patterns. These filename 'headers' can be eliminated by selecting the **-h** option for **grep**:

```
grep -h '!!!OTL:' *
```

(N.B. Some older versions of **grep** do not support all of the options described here. Filename headers can be stripped from the output by using the UNIX **sed** described in Chapter 14.)

We might place the resulting list of titles in a separate file using the UNIX *file redirection* construction. The output of a command can be placed into a file by following the command with a greater-than sign (>) followed by a filename. For example, the following command places the output from **grep** in a file called `titles`:

```
grep -h '!!!OTL:' * > titles
```

Beware that if the file `titles` already exists then it will be over-written and its previous contents lost. With the **-h** option the file `titles` might contain the following lines:

```
!!!OTL: Oh! Susanna
!!!OTL: Jeanie with the Light Brown Hair
!!!OTL: Beautiful Dreamer
!!!OTL: Gwine to Run All Night (or 'De Camptown Race')
!!!OTL: My Old Kentucky Home, Good-Night
!!!OTL: We are Coming, Father Abraam
!!!OTL: Don't Bet Your Money on De Shanghai
!!!OTL: Gentle Annie
!!!OTL: If You've Only Got a Moustache
!!!OTL: Maggie by my Side
!!!OTL: Old Folks at Home
!!!OTL: Better Times are Coming
!!!OTL: When this Dreadful War is Ended
!!!OTL: Hard Times Comes Again No More
```

The *sort* Command

The UNIX operating system provides a general sorting utility called **sort**. We might use this utility to rearrange the titles in alphabetical order:

```
sort titles
```

Rather than using an intermediate file, we can directly connect the **grep** and **sort** commands using a UNIX "pipe." The vertical bar (|) creates a connection between the output of one command and the input of the next command. We can combine the above two commands to create an alphabetical listing of all titles in the current directory:

```
grep '!!!OTL:' * | sort
```

File-redirection can be added at the end of a pipe so the final output is captured in a file. In the follow case, the alphabetized titles are placed in the file `titles`:

```
grep '!!!OTL:' * | sort > titles
```

The *uniq* Command

Bach often harmonized a chorale melody more than once. In the 185 chorales in the original 1784 edition, several duplicate titles are present. Suppose you want to create an alphabetical list of titles

— but you want to exclude duplicate titles.

The UNIX **uniq** command provides a useful utility for eliminating duplication. Without any option, **uniq** simply eliminates any successive repeated lines. For example, given the input:

```
1
1
1
2
2
3
```

the **uniq** command will produce the following output:

```
1
2
3
```

Note that **uniq** only discards *successive* repeated records; an input such as the following would remain unmodified by the **uniq** command:

```
1
2
3
1
3
1
```

Another important point about **uniq** is that successive lines must be *exact repetitions* in order to be discarded. For example, if one line has a trailing blank that is not present in the previous line, then the line is not discarded.

Returning to our problem of creating a list of unique titles for J.S. Bach's chorale harmonizations, we can use the following command pipeline.

```
grep -h '!!!OTL:' * | sort | uniq
```

Note that our "pipeline" consists of three successive commands with the outputs connected to the inputs using the UNIX pipe symbol (|). The **sort** command is essential in order to collect identical titles as successive lines before passing the list to **uniq**.

Suppose you wanted to ensure that all of the works in the current directory are composed by the same composer. The same command structure can be used, only we would search for reference records encoding the composer's name:

```
grep -h '!!!COM:' * | sort | uniq
```

Even if the current directory contains hundreds of works by one composer (say Beethoven) and just a single work by another composer, the presence of the odd score will be obvious without hav-

ing to look through long lists:

```
!!!COM: Beethoven, Ludwig van
!!!COM: Stamitz, Carl Philipp
```

Of course we can make similar lists for other types of information available in reference records. The AIN reference record encodes instrumentation. We could make a list of various instrumental combinations used for scores in the current directory:

```
grep -h '!!!AIN:' * | sort | uniq
```

Options for the *uniq* Command

Like **grep**, the **uniq** command provides several options that modify its behavior. The **-d** option causes only those records to be output which are *duplicated* (i.e. two or more instances). Conversely, the **-u** option causes only those records to be output that are truly *unique* (i.e. only a single instance is present in the input).

Suppose, for example, that we want to know which of the Bach chorales are harmonizations of the same tunes — that is, have the same titles. (Of course the same chorale might be known by two or more titles, but let's defer this problem until Chapter 25.) The **-d** option will only output the duplicate records:

```
grep -h '!!!OTL:' * | sort | uniq -d
```

The output will identify those titles which appear in two or more files in the current directory. The output might look as follows:

```
!!!OTL: Befiehl du deine Wege
!!!OTL: Christ lag in Todesbanden
!!!OTL: Christus, der ist mein Leben
!!!OTL: Das alte Jahr vergangen ist
!!!OTL: Ein' feste Burg ist unser Gott
!!!OTL: Erbarm' dich mein, o Herre Gott
!!!OTL: Herr, ich habe missgehandelt
!!!OTL: Herr, wie du willst, so schick's mit mir
!!!OTL: Ich dank' dir, lieber Herre
!!!OTL: Jesu, der du meine Seele
!!!OTL: Jesu, meiner Seelen Wonne
```

Having established which titles are duplicates, a logical next step might be to identify the specific files involved. We can use **grep** again to search for a specific title. Without the **-h** option, the output will identify the appropriate filenames. For example:

```
grep '!!!OTL: Befiehl du deine Wege' *
```

might produce the following output:

```
bwv270.krn:!!!OTL: Befiehl du deine Wege
bwv271.krn:!!!OTL: Befiehl du deine Wege
bwv272.krn:!!!OTL: Befiehl du deine Wege
```

Sometimes we would like to have an output that contains *only* the *filenames* containing the sought pattern. The **-l** option causes **grep** to output only filenames that contain one or more instances of the sought pattern:

```
grep -l '!!!OTL: Befiehl du deine Wege' *
```

The output would appear as follows:

```
bwv270.krn
bwv271.krn
bwv272.krn
```

The **-u** option for **uniq** causes only unique entries in a list to be passed to the output. This is often useful in identifying works that differ in some way from other works in a group or corpus. For example, in some repertory, you may remember that a particular work had a different instrumentation than the other works. But you may not be able to remember what the specific instrumentation was. Use the **-u** option for **uniq** to produce a list consisting of only those works whose instrumentation differs from all others:

```
grep -h '!!!AIN:' * | sort | uniq -u
```

As in the case of the **grep** command, **uniq** also supports a **-c** option which counts the number of occurrences of a pattern. For example, if we want to count the number of works by each composer in the current directory:

```
grep -h '!!!OTL:' * | sort | uniq -c
```

The output might appear as follows:

```
 9 !!!COM: Berardi, Angelo
 2 !!!COM: Caldara, Antonio
12 !!!COM: Zarlino, Gioseffo
 2 !!!COM: Sweelinck, Jan Pieterszoon
 4 !!!COM: Josquin Des Pres
```

Notice that the number of instances is prepended to the reference records.

Incidentally, if we wanted to rearrange this list in order of the number of works, we could pass the above output to yet another **sort** command. Since **sort** sorts from left to right, it will begin sorting according to the numerical values at the extreme left. The command

```
grep -h '!!!OTL:' * | sort | uniq -c | sort
```

will rearrange the above output as follows:

```
  2 !!!COM: Caldara, Antonio
  2 !!!COM: Sweelinck, Jan Pieterszoon
  4 !!!COM: Josquin Des Pres
  9 !!!COM: Berardi, Angelo
 12 !!!COM: Zarlino, Gioseffo
```

It is important to understand that the two **sort** commands in our pipeline achieve different goals but use the same process. The first **sort** command sorts the composer's names into alphabetical order. This is done so that the ensuing **uniq** command is able to count successive identical records. Since the **uniq -c** command prepends numerical counts, the subsequent **sort** sorts first according to the numbers to the left of the reference records.

As a final note, we might mention that, like **grep** and **uniq**, the **sort** command has several options. One option, the **-r** option, causes the output to be arranged in reverse order. This can be useful in producing lists that are ordered from the most-common to the least-common.

Reprise

In this chapter we have introduced some elementary ways of processing Humdrum files. We noted that the **census** command can be used to identify basic statistics about a file. The **-k** option for **census** provides basic information related to `**kern` files — such as the number of notes and rests, the highest and lowest notes, the number of barlines, etc.

In this chapter we also introduced simple searching techniques using the **grep** command; **grep** provides a useful way of locating particular patterns of text characters in files. We used **grep** to identify composers, titles, instrumentation and other information. Most of our examples were limited to searching for Humdrum reference records. In later chapters we will use **grep** in more sophisticated searches. We noted several useful options for **grep**: the **-c** option causes a count to be output of the number of instances of the pattern in each file. The **-i** option causes **grep** to ignore any distinction between upper- and lower-case characters when searching for patterns. The **-h** option causes **grep** to avoid outputting the filenames prior to found patterns when more than one file is searched. The **-l** option results in only the filenames being output. In a later chapter we will encounter a number of other useful options provided by **grep**.

Also discussed in this chapter was the **uniq** command; **uniq** provides a useful utility for eliminating or isolating duplicate records or lines. Once again a number of useful options were introduced. The **-c** option causes **uniq** to prepend a count of the number of duplicate input lines. The **-d** option results in only duplicate input lines being noted in the output. The **-u** option does the reverse: only those input lines that are unique are passed to the output.

Finally, we introduced the UNIX **sort** utility. This command rearranges the order of successive input lines so they are in alphabetic/numeric order. The **sort** command provides a wealth of useful options; however, we mentioned only the **-r** option — which causes the output to be sorted in reverse order.

Chapter 4

Basic Pitch Translations

Many musical processes entail some sort of data translation in which one form of representation is transformed into another form of representation. There are innumerable examples of such musical "translations." For example, we might rewrite guitar tablatures as notated pitches. Pitches might be translated to tonic-solfa syllables. Scale-degrees might be analyzed as Roman-numeral harmonies. Figured bass notations might be "realized" as pitches. Successive pitches might be characterized as melodic intervals. Intervals might be rewritten as semitone distances. Pitch-class sets might be transformed to interval-class vectors.

In this chapter we introduce some basic musical translations. We will limit this initial discussion to translating between various representations related to pitch. These will include French, German, and international pitch designations, frequency, semitones, cents, solfège, scale-degree and MIDI pitch representations. In later chapters we will describe additional types of translations.

ISO Pitch Representation

The best-known system for representing equally-tempered pitches is the International Standards Organization (ISO) format consisting of a letter (A-G) followed by an optional sharp or flat, followed by an octave number. According to the ISO representation middle C is designated C4. Octave numbers change between B and C so that the B a semitone below C4 is B3. Humdrum provides a predefined ISO-like representation called `**pitch` illustrated below. Here we see an ascending chromatic scale in the left spine, with a concurrent descending chromatic scale in the right spine:

```
**pitch   **pitch
C4        C5
C#4       B4
D4        Bb4
D#4       A4
E4        Ab4
F4        G4
F#4       Gb4
G4        F4
G#4       E4
```

```
A4          Eb4
A#4         D4
B4          Db4
C5          C4
*-          *-
```

Notice that only upper-case letters are used for pitch-name and that the flat is represented by the lower-case letter 'b'. The small letter 'x' can be used to indicate double-sharps and the double-flat is represented by two small successive letters 'bb'. If a given pitch deviates from equal temperament, cents deviation can be indicated by trailing integers preceded by either a plus sign (tuned sharp) or a minus sign (tuned flat). The unit of the *cent* is one one-hundredth of a semitone, so the distance between C4 and C#4 is 100 cents. For example, the pitch corresponding to a tuning of A-435 Hz is 19 cents flat compared with the standard A-440, hence it is represented in **pitch as:

```
A4-19
```

Other pitch representations (such as **kern) can be translated to the ISO-inspired **pitch representation by invoking the **pitch** command. For example, consider the following **kern input:

```
**kern
=1
4g
4g#
4a
4cc
=2
*-
```

It can be translated to this **pitch output:

```
**pitch
=1
G4
G#4
A4
C5
=2
*-
```

using the following command:

```
pitch filename
```

Notice that the **pitch representation uses the same system for representing barlines as **kern. In fact, all of the pitch-related representations described in this chapter make use of the so-called 'common system' for representing barlines.

German Tonhöhe

The German system of pitch designations (*Tonhöhe*) is available in the Humdrum `**Tonh` representation. The German system is similar to the ISO system with the following exceptions. The pitch letter names include A,B,C,D,E,F,G,H and S. Sharps and flats are indicated by suffixes: '`is`' for sharps and '`es`' for flats, hence '`Cis`' for C-sharp and '`Ges`' for G-flat. Suffixes are repeated for double and triple sharps and flats. Special exceptions include the following: '`B`' for B-flat, '`H`' for B-natural, '`Heses`' for B double-flat (rather than '`Bes`'), and '`As`' and '`Es`' rather than '`Aes`' or '`Ees`'. '`S`' is a short-hand for '`Es`' (E-flat). As in the ISO pitch system, octaves are indicated by integer numbers with middle C represented as C4.

Although modern German practice has gravitated toward the ISO system, the traditional German system for representing pitches remains important in historically related studies, such as searching for 'B-A-C-H' and the pitch signature used by Dmitri Shostakovich ('D-S-C-H').

Data in the `**pitch` or `**kern` representations can be translated to `**Tonh` via the **tonh** command:

```
tonh filename
```

French Solfège

The common French system for pitch naming uses a so-called "fixed-do" method of diatonic pitch designations: *do, ré, mi, fa, sol, la* and *si* (rather than *ti*), where *do* corresponds to the English/German 'C'. In the Humdrum `**solfg` representation, solfège pitch names are used. Flats (*bémol*) and sharps (*dièse*) are abbreviated *b* and *d* respectively. When accidentals are encoded, the tilde character (`~`) is encoded following the solfège syllable and before the accidental. Double and triple sharps and flats are encoded via repetition. Hence, 'do dièse' (`do~d`) for C-sharp, 'la bémol' (`la~b`) for A-flat, 'sol double-dièse' (`sol~dd`) for G double-sharp, 'si double-bémol' (`si~bb`) for B double-flat, and so on. As with the German and ISO pitch representations, octave is designated by integers with `do4` representing middle C.

Frequency

For acoustic-related applications it may be helpful to translate to frequency. The Humdrum `**freq` representation can be used to represent frequencies for either pure or complex tones. Frequencies are encoded in *hertz* (abbreviated *Hz*) where 440 Hz means 440 cycles per second. In the `**freq` representation frequencies may be specified as integer or real values (with a decimal point).

Cents

The `**cents` representation provides a means for representing pitches in absolute units with respect to middle C (= 0 cents). In the `**cents` representation, all pitches are represented with respect to this reference. Thus C#4 is represented by the number 100, A4 is represented by 900, and A3 is represented by –300. As in the case of `**freq`, cents may be specified as integer numbers

or as real values (with a decimal point).

Semitones

A related pitch representation is `**semits`. In this case, all pitches are represented in numerical semitones with respect to middle C (= 0 semits). An ascending chromatic scale beginning on C4 would be represented by the ascending integers from 0 to 12. Pitches below middle C are represented by negative values. Fractional values can be represented using decimal points.

MIDI

Another way of representing pitch is provided by the Humdrum `**MIDI` representation. This representation closely mimics the commercial MIDI specification. The `**MIDI` representation allows MIDI inputs and outputs to be exported or imported by various Humdrum tools. A complete description of `**MIDI` will be given in Chapter 7.

Scale Degree — ***solfa* and ***deg*

Two different Humdrum representations are provided to describe scale-degree related information: `**deg` and `**solfa`. Both of these representations emphasize slightly different aspects of scale-degree information. Both representations assume some established or pre-defined tonal center or tonic pitch.

The `**solfa` representation represents pitch according to tonic solfa syllables. Pitches are designated by the syllables *do, re, mi, fa, so, la* and *ti* or their chromatic alterations as indicated in the following table:

basic	raised	lowered
do (*doe*)	di (*dee*)	de (*day*)
re (*ray*)	ri (*ree*)	ra (*raw*)
mi (*me*)	my (*my*)	me (*may*)
fa (*fah*)	fi (*fee*)	fe (*fay*)
so (*so*)	si (*see*)	se (*say*)
la (*la*)	li (*lee*)	le (*lay*)
ti (*tee*)	ty (*tie*)	te (*tay*)

Summary of **solfa** *Signifiers*

The `**deg` representation identifies scale-degrees by the numbers 1 (tonic) to 7 (leading-tone). These values may be chromatically altered by raising (+) or lowering (−). The *amount* of chromatic alteration is not indicated; for example, both a raised supertonic and a doubly-raised supertonic are represented as `2+`. A lowered dominant is represented as `5-`.

The `**solfa` representation differs from `**deg` in that pitches are represented without regard to major or minor *mode*. For example, in the key of C major, `**deg` will characterize A-flat as a lowered sixth scale degree (`6-`), whereas the same pitch will be a normal (unaltered) sixth scale degree in the key of C minor (`6`). In the case of `**solfa`, the A-flat will be represented as `le` —

whether or not the key is C major or C minor. Like `**deg`, the amount of chromatic alteration is not represented in `**solfa`. Once a pitch is raised, raising it further will not change the representation. For example, if the tonic is B-flat, then both B-natural and B-sharp will be represented by `di` in the `**solfa` representation.

In the case of the minor mode, `**deg` characterizes scale degrees with respect to the *harmonic minor* scale only.

Another difference between `**solfa` and `**deg` is that the `**deg` representation provides a way for encoding *melodic approach.* The caret (`^`) denotes an ascending melodic approach to the current note, whereas the lower-case letter `v` denotes a descending melodic approach. Repeated pitches carry no melodic approach signifier.

Some of the differences between the `**solfa` and `**deg` representations are illustrated in Example 4.1. (The corresponding `**kern` representation is given in the first spine.) Notice that `**solfa` does not encode any octave information. The `**deg` representation does not encode the octave of the starting pitch, but it does indicate contour information using the caret (`^`) for ascending and the lower-case `v` for descending pitches. Notice also the different ways of characterizing accidentals.

Example 4.1

```
!! Comparison of pitch-related representations.
**kern          **solfa         **deg
*M2/4           *M2/4           *M2/4
*c:             *c:             *c:
8.cc            do              1
16dd            re              ^2
=1              =1              =1
8.ee-           me              ^3
16dd            re              v2
4een            mi              ^3+
=2              =2              =2
8r              r               r
8b-             te              v7-
8an             la              v6+
8cc             do              ^1
=3              =3              =3
2bn             ti              v7
==              ==              ==
*-              *-              *-
```

Pitch Translations

Humdrum provides a number of commands for translating between the various pitch-related representations described above. Typically, the command name is the same as the name of the output representation. For example, translating to the `**solfg` representation can be accomplished with:

```
solfg inputfile > outputfile
```

Translating to the German **Tonh representation:

```
tonh inputfile > outputfile
```

Translating to ISO **pitch:

```
pitch inputfile > outputfile
```

Similarly, the **freq** command translates pitch-related inputs to the **freq representation, the **cents** command translates appropriate inputs to the **cents representation, and so on.

In a few cases, the command names are slightly modified. All Humdrum command names employ lower-case letters only, so **MIDI output is generated by the **midi** command (rather than the **MIDI** command), and **Tonh output is generated by the **tonh** command.

Examples 4.2 and 4.3 compare several parallel representations of the same pitch-related information. In both examples, the pitch information has been derived from the **kern data shown in the left-most spine. The duration information in the **kern data is not available in the other representations. However, the 'common system' for barlines is used throughout.

Example 4.2 shows four pitch naming systems: ISO pitch, German Tonhöhe, French solfège, as well as **kern. Notice the different ways of treating accidentals such as the D-sharp and B-flat. Also note the German use of H for B-natural.

Example 4.2

```
!! Comparison of pitch-related representations.
**kern      **pitch     **Tonh      **solfg
*M2/4       *M2/4       *M2/4       *M2/4
*C:         *C:         *C:         *C:
8.cc        C5          C5          do5
16dd        D5          D5          re5
=1          =1          =1          =1
8.ee        E5          E5          mi5
16dd#       D#5         Dis5        re~d5
4ee         E5          E5          mi5
=2          =2          =2          =2
8r          r           r           r
8b-         Bb4         B4          si~b4
8a          A4          A4          la4
8c          C4          C4          do4
=3          =3          =3          =3
2bn         B4          H4          si4
==          ==          ==          ==
*-          *-          *-          *-
```

In Example 4.3 four of the more technical representations are illustrated, including frequency and cents. Notice that the **MIDI representation uses key-numbers to represent pitch: key-on events

are indicated by positive integers (between two slashes) and key-off events are indicated by negative integers. More detail concerning **MIDI is given in Chapter 7.

Example 4.3

```
!! Comparison of pitch-related representations (continued).
**kern      **semits     **cents     **MIDI       **freq
*M2/4       *M2/4        *M2/4       *Ch1         *M2/4
*C:         *C:          *C:         *M2/4        *C:
*           *            *           *C:          *
8.cc        12           1200        /72/         523.25
16dd        14           1400        /-72/ /74/   587.33
=1          =1           =1          =1           =1
8.ee        16           1600        /-74/ /76/   659.26
16dd#       15           1500        /-76/ /75/   622.25
4ee         16           1600        /-75/ /76/   659.26
=2          =2           =2          =2           =2
8r          r            r           /-76/        r
8b-         10           1000        /70/         466.16
8a          9            900         /-70/ /69/   440.00
8c          0            0           /-69/ /60/   261.63
=3          =3           =3          =3           =3
2bn         11           1100        /-60/ /71/   493.88
==          ==           ==          ==           ==
.           .            .           /-71/        .
*-          *-           *-          *-           *-
```

Not all of the above pitch-related representations can be translated directly from one to another. Table 4.1 shows the possible translations supported by Humdrum Release 2.0 commands. The input representations are listed from right to left. Under each column, those commands that will translate *from* the given format are identified. For example, the **cents representation can be translated to **freq, **kern, **pitch, **semits, **solfg, and **tonh. Notice that **deg data cannot be translated to any other format since **deg representations do not encode absolute pitch height. Note also that when translating to the **kern representation, only pitch-related information is translated: duration, articulation marks, and other **kern signifiers are not magically generated.

Table 4.1

Input Representation

	**cents	**deg	**freq	**kern	**MIDI	**pitch	**semits	**solfa	**solfg	**Tonh
cents			•	•		•	•		•	•
cocho			•							
deg				•		•			•	•
freq	•			•	•	•	•		•	•
kern	•		•		•	•	•	•	•	•
midi				•						
pitch	•		•	•	•		•	•	•	•
semits	•		•	•	•	•			•	•
solfa				•		•			•	•
solfg	•		•	•	•	•	•		•	
tonh	•		•	•	•	•	•	•	•	

Transposition Using the *trans* Command

A common pitch-related manipulation is transposition. The **trans** command has the user specify a *diatonic offset* and a *chromatic offset*. The diatonic offset affects the pitch-letter name used to spell a note. The chromatic offset affects the number of semitones shifted from the original pitch height. The two types of offset are completely independent of each other. For common transpositions, both the diatonic and chromatic offsets will need to be specified. For example, in transposing up a minor third (e.g. C to E-flat), the diatonic offset is 'up two pitch-letter names,' and the chromatic offset is 'up three semitones.' The appropriate command invocation is:

```
trans -d +2 -c +3 input > output
```

The diatonic offset can be a little confusing because traditional terminology labels perfect unisons by the number 1 (e.g. P1) rather than zero. So transposing up a perfect fifth involves a diatonic offset of +4 letter names, and a chromatic offset of +7 semitones:

```
trans -d +4 -c +7 input > output
```

We can transpose without changing the diatonic pitch names. For example, the following command will transpose down an augmented unison (e.g. C# to C):

```
trans -d 0 -c -1 input > output
```

Conversely, we can respell the diatonic pitches without changing the overall pitch height. For example, the following transposition will transpose "up" a diminished second (e.g. from F-sharp to G-flat):

```
trans -d +1 -c 0 input > output
```

Modal transpositions are also possible by omitting the chromatic offset option. Consider, for ex-

ample, the following C major scale:

```
**kern
c
d
e
f
g
a
b
cc
*-
```

We can transform this using the following diatonic transposition:

```
trans -d +1
```

The resulting output is the Dorian mode:

```
**kern
*Trd1
d
e
f
g
a
b
cc
dd
*-
```

When using the **-d** option alone, **trans** eliminates all accidentals in the input. This can be potentially confusing, but it is often useful. Suppose you have a passage in the key of E major which you would like to translate to E Dorian. First transpose so the tonic is D using only the **-d** option; then transpose exactly so the tonic is E again:

```
trans -d -1 Emajor | trans -d +1 -c +2 > Edorian
```

For some changes of mode (such as melodic to harmonic minor), you may need to use the **humsed** command described in Chapter 14 to modify accidentals for specific scale degrees.

Notice the addition of a "tandem interpretation" to the above example (`*Trd1`). Whenever **trans** is invoked, it adds a record indicating that the encoding is no longer at the original pitch. *Transposition tandem interpretations* are similar in syntax to the **trans** command itself. In the above example, `*Trd1` indicates a diatonic shift up one letter name. The tandem interpretation `*Trd-1c-2` would indicate that a score has been transposed down a major second. The **trans** command also provides a **-k** option that allows the user to specify a replacement key signature for the output.

The **trans** command can be used in conjunction with any of the appropriate pitch-related representations, such as `**pitch`, `**kern`, `**Tonh`, and `**solfg`.

Key Interpretations

In order for the **solfa** or **deg** commands to translate from other pitch representations, the encoded music must contain an explicit key indication. Keys are explicitly represented by a single asterisk, followed by an upper- or lower-case letter, followed by an optional accidental, followed by a colon. The octothorpe (#) indicates a sharp and the hyphen (–) indicates a flat.

Upper-case letters indicate major keys; lower-case letters indicate minor keys. By way of illustration, the following key interpretations indicate the keys of C major, C minor, B-flat major, and F-sharp minor:

```
*C:
*c:
*B-:
*f#:
```

Key interpretations usually appear near the beginning of a representation, and key interpretations can be redefined at any place in a score.

Pitch Processing

Apart from transposition, translating from one representation to another provides opportunities for different sorts of processing. Suppose, for example, we wanted to know whether the subdominant pitch occurs more frequently in one vocal repertory than in another repertory. We can use **solfa** in conjunction with **grep**'s **-c** option to count the number of occurrences. (For the following examples, we will assume that the inputs consist of only a single spine, that barlines are absent, and that appropriate interpretations are provided indicating the key of each work.) First we need to count the total number of notes in each repertory.

```
census -k repertory1.krn
census -k repertory2.krn
```

Next we translate the scores to the **solfa** representation and use **grep -c** to count the number of occurrences of the number 'fa':

```
solfa repertory1.krn | grep -c fa
solfa repertory2.krn | grep -c fa
```

The proportion of subdominant pitches can be calculated by simply comparing the resulting pattern count with the number of notes identified by **census**.

Recall that one of the differences between the `**solfa` and `**deg` representations is that the `**deg` output contains an indication of the direction of melodic approach. The caret (^) indicates approach from below, whereas the lower-case **v** indicates approach from above. Suppose we wanted to determine whether the dominant pitch is more commonly approached from above or from below. Assuming a monophonic input, we can once again use **grep** to answer this question. First let's count how many dominant pitches ('5') are approached from above ('v'):

```
deg repertory.krn | grep -c v5
```

The caret has a special meaning for **grep** which will be discussed in Chapter 9. We can escape the special meaning by preceding the caret by a backslash. In order to count the number of dominant pitches approached from below we can use the following:

```
deg repertory.krn | grep -c \^5
```

Recall that some scale tones are spelled differently depending on whether the mode is major or minor. For example, in A major the mediant pitch is C sharp; but in A minor the mediant pitch is C natural. The **deg** and **solfa** commands produce subtly contrasting outputs that make one or the other command better suited depending on the user's goal. The **deg** command would represent C sharp in A major, and C natural in A minor by the same scale degree — `3`. In the key of A major, C natural would be characterized as a lowered mediant (`3-`) and in A minor, C sharp would be characterized as a raised mediant (`3+`). By contrast, the **solfa** command characterizes pitches with respect to the tonic alone and ignores the mode. Hence, **solfa** would designate C sharp as '`mi`' whether the key was A major or A minor. Similarly, C natural would be designated '`me`' in both A major and A minor. The differences between **deg** and **solfa** allow users to distinguish chromatically altered scale tones in a manner appropriate to the task.

Uses for Pitch Translations

Occasionally it is useful to process a given representation to the *same* representation. The **kern** command translates various pitch-related representations to the `**kern` format. The **-x** option eliminates any input data that do not pertain to pitch. When applied to a `**kern` input, this option allows us to filter out durations, articulation marks, phrasing, and other non-pitch data. Suppose, for example, that we wanted to determine the proportion of successively repeated notes in a vocal melody: how often is a pitch followed immediately by the same pitch? We might begin by first determining the total number of notes in the melody using **census** with the **-k** option.

```
census -k melody.krn
```

We can use the **uniq** command to eliminate successive repeated pitches — but only if the note tokens are identical. First we can use **kern -x** to translate "from `**kern` to `**kern`" while eliminating non-pitch-related data. Then we need to remove barlines so they don't interfere with pitches that are repeated across the measure. Using **uniq** will then eliminate all of the successively duplicated records, so a sequence of six G's will be reduced to a single G. Finally, we pipe the output to **census -k** to count the total number of notes.

```
kern -x melody.krn | uniq | census -k
```

A variation on this approach would entail translating to a representation that does not distinguish enharmonic pitches. For example, translating our melody to `**semits` and then back to `**kern` will standarize all of the enharmonic spellings. If our melody contains a G-sharp that undergoes an enharmonic shift to A-flat, then the pitches will be deemed identical. The following command carries out the same task as above, but ignores possible enharmonic spellings:

```
semits melody.krn | kern | uniq | census -k
```

Incidentally, given `**semits` input, the **kern** command will spell pitches according to any key or key signatures it encounters. For example, if the key signature contains sharps, then G-sharp will be output; if the key or key signature contains flats, then A-flat will be output.

Reprise

In this chapter we have introduced a number of pre-defined pitch-related representations. Simple commands can be used to translate from one representation to another. Which representation is most appropriate depends on the user's goal.

There is a wealth of other representation formats related to pitch distances, tablatures, timing, and other types of musical information. These representations will be explored in later chapters. In addition, we'll describe how to design your own representations — representations that may be better tailored to a specific application. However, before we continue discussing further representations, this is an appropriate point to present a more formal description of the general Humdrum representation syntax.

Chapter 5

The Humdrum Syntax

In the previous chapters we have seen several examples of pre-defined Humdrum representations, such as `**kern`, `**solfa` and `**MIDI`. These representations exhibit a number of common properties, including the manner in which the data are organized in spines. In this chapter, we provide a complete description of the Humdrum representation syntax. This chapter will help you better understand how Humdrum representations are organized, and will provide essential foundations for designing your own Humdrum representations.

The Humdrum syntax provides a framework within which representation schemes can be defined. Each scheme consists of a mapping between the concepts we wish to represent (called *signifieds*) and how we wish to represent them (called *signifiers*). The signifieds can be any music-related concept determined by the user. The signifiers consist of the text characters commonly available on computers.

Humdrum regards each file as a two-dimensional plane, much like a sheet of paper. *Successions* of events proceed vertically down the page, whereas *concurrent* events extend horizontally across the page. Two signifiers that occupy the same horizontal line represent concurrent (or overlapping) events. The basic organization of Humdrum files may be schematically illustrated as follows:

	successive events ↓			
concurrent events →	A	J	V	etc.
	B	K	W	etc.
	C	L	X	etc.
	D	M	Y	etc.
	etc.	etc.	etc.	

Types of Records

Humdrum encodings consist of a set of one or more lines or *records.* There are three types of Humdrum records:

1. comment records,

2. interpretation records, and
3. data records.

These three record types are mutually exclusive, so it is not possible to mix comments, interpretations, or data records on the same line.

Comment Records

As we noted in Chapter 2, there are two kinds of comments: global comments and local comments. *Global comments* pertain to all concurrent parts, whereas *local comments* pertain to some specific column of data (such as a particular staff, instrument, note, finger, etc.). Comments are lines that contain an exclamation mark (!) at the beginning of the record (in the left-most position); subsequent characters up to and including the occurrence of a carriage return or newline character constitute the comment record. Recall that global comments are denoted by two exclamation marks (!!) at the beginning of the record. Global comments may contain any sequence of printable characters, including 'blank space' such as tabs and spaces. Local comments may contain any sequence of printable ASCII characters, with the important exception of the tab character which is used to separate spines. Comments may be used to insert free-format commentaries in Humdrum encodings.

Interpretation Records

Interpretations are lines that begin with the asterisk character (*). Interpretations are used to identify more precisely the state of the representation — for example, to indicate that an encoded part is for a transposing instrument in E-flat, or to indicate that the representation is for a given Balinese tuning, or that the representation encodes a conductor's physical gestures. Humdrum requires that at least one interpretation must be specified before any data records are encountered. The difference between a comment and an interpretation is that interpretations are formal, potentially *executable* statements; interpretations pass information to programs that process the Humdrum encoding.

As in the case of comments, there are two types of interpretations: *exclusive* interpretations begin with a double asterisk (**) whereas *tandem* interpretations begin with a single asterisk (*). Exclusive interpretations are mutually exclusive: only one such interpretation can be active at a given time for a given string of data. No set of data is complete without the presence of an exclusive interpretation. Tandem interpretations, by contrast, provide supplementary information about how a set of data is to be interpreted. Several tandem interpretations may pertain to a given set of data; unlike exclusive interpretations, tandem interpretations are not necessarily mutually exclusive.

Data Records

Lines that do not contain either an exclamation mark or an asterisk in the first column are *data records.* Blank lines (i.e., lines which are either empty, or contain only blank space, such as tabs and spaces) are forbidden in Humdrum. Thus data records may be formally defined as non-empty lines that do not begin with either an exclamation mark or an asterisk.

In Humdrum, each data record encodes information pertaining either to a particular moment in time or to a particular time window or duration. (Whether a record represents a precise moment or an expanse of time depends on the accompanying interpretation.) Each data record may contain one or more data *tokens.* When more than one token is present, tokens are separated from each other by tabs. When several data records are present, multiple tokens are aligned in columns through the file. As we noted earlier, these columns are referred to as *spines.* By itself, a spine has no particular meaning; it is simply a way of linking together related tokens through time. Spines become meaningful only when they are labelled by adding an interpretation.

By itself, Humdrum recognizes only six ASCII characters. Two of these characters — the exclamation mark (!) and the asterisk (*) — have a special meaning *only* when they appear in the first column of a record (or are preceded by a tab; see below). The remaining special Humdrum characters are the period (.), the space, the tab character, and the carriage return (= newline character). As we have seen, the exclamation mark and the asterisk are used to identify comments and interpretations, respectively. The tab and carriage return characters are used to format the data into *spines* and *records,* respectively.

Data Tokens and Null Tokens

As we noted above, the data in the data records are conceptually divided into tokens. In Humdrum, there are two possible types of tokens:

1. *data* tokens, and
2. *null* tokens (.).

Consider, for example, the following file:

```
X       .       X
X       X       X
.       X       X
X       .       X
```

This file consists of three vertical spines and four horizontal records. The first and third spines begin with data tokens, while the second spine begins with a null token. Without the presence of interpretations, the meaning of this file is indeterminate. The file below contains two spines that have been labelled using Humdrum interpretations:

```
**left  **right
X       .
.       X
X       .
.       X
X       .
*-      *-
```

The user has defined two interpretations: "left" and "right." The intention is to represent the footfalls of a person's left and right feet. The representation simply encodes that the left and right feet have alternating events, such as might be produced by walking or running. Notice that null tokens (.) indicate nothing at all and merely act as place-holders to maintain the format of the two spines. Notice also that interpretations must be defined for each spine, and that each interpretation consists

of some keyword appended to the double asterisks (e.g. `**left`). No intervening spaces are permitted between the interpretation *keyword* (`left`) and the asterisks; however, spaces may appear as part of the keyword itself. In addition, when more than one spine is present, both the data tokens and the associated interpretations must be separated by a tab character; spaces cannot be used to separate spines. Finally, note that each spine is formally terminated by a *spine-path terminator* — an asterisk followed by a minus sign.

Interpretations can be cascaded so that a single spine has more than one interpretation associated with it. This is done through the addition of tandem interpretations. Consider the following example:

```
**foot   **foot   **arm   **arm
*left    *right   *left   *right
X        .        .       X
.        X        X       .
X        .        .       X
.        X        X       .
X        .        .       X
*-       *-       *-      *-
```

In this case the categories "left" and "right" have been transformed to tandem interpretations. The first spine is interpreted both as "left" and as "foot." The exclusive interpretation (double asterisks) takes conceptual precedence over the tandem interpretation (single asterisk). That is, tandem interpretations merely modify or supplement the exclusive interpretation. Hence, given the above representation, we could say that "left" is an attribute of "foot" or "arm," but we could not say that "foot" is an attribute of "left."

Users are free to define as many different exclusive and tandem interpretations as they wish. For example, a user might define the interpretation `**bowing` that would be suitable for encoding detailed bowing information in works for strings. For each exclusive interpretation, the Humdrum user can re-define the meaning of all of the text characters, with the exception of the tab and the carriage return, which always retain their functions as 'token/spine separator' and 'record separator' respectively. The characters ! . * can also be re-defined, although there are some restrictions as to how they can be used. Specifically, the exclamation mark cannot occur in the first column of the record unless it is used to indicate a comment. Similarly, the asterisk cannot occur in the first column of a record unless it is used to indicate a Humdrum interpretation. The period cannot appear in the first column unless it is used to indicate a null data token. In addition, the exclamation mark, asterisk, and period cannot appear following a tab unless they are used to indicate a comment, interpretation, or null token, respectively.

Data Sub-Tokens

Data tokens can be split into sub-tokens via the space character. In the first data record of the following example, the first spine contains two sub-tokens whereas the third spine contains three sub-tokens. Sub-tokens do not have their own spine organization and can appear and disappear as necessary:

```
**spine1    **spine2  **spine3
A B         J         X Y Z
AB          J         XYZ
A B C       .         X Z
*-          *-        *-
```

Data sub-tokens are useful in a variety of circumstances. An appropriate use of sub-tokens might be to encode double- and triple-stops in string parts.

In the Humdrum data records, the space character is reserved solely for use as a sub-token delimiter. Note that consecutive spaces are illegal, and that data tokens cannot begin or end with a space character. Of course spaces can be used freely in comments and in interpretations.

Spine Paths

Humdrum representations often consist of a fixed number of spines that continue throughout the course of an encoded file. As we have seen in the preceding chapters, a typical use of spines is to encode different voices or parts in a musical work. However, there is no reason to equate spines with voices; spines are used for many other purposes as well.

In encoding Humdrum representations it is occasionally useful to be able to vary the number of spines. However, files with varying numbers of spines can pose significant questions of interpretation. Consider, for example, the following sequence of Humdrum-like data records:

```
1       2       3
1       2       3
1       2       3
A       B
A       B
A       B
```

At the point where three spines are reduced to two spines the continuity is ambiguous: Has spine '3' been discontinued? Or is spine 'B' a continuation of spine '3' with spine 'A' a continuation of spine '1' or '2'? For some representations such questions will be of little concern; however, in other circumstances, the manner in which the spines continue will be of critical importance. For example, if all of the above spines encoded pitch information for various musical parts, a study of melodic intervals would need to resolve the specific melodic paths as the representation moves from three to two spines. Failure to clarify the pitch paths would make it difficult to determine or search for specific successions of melodic intervals.

The Humdrum syntax provides special *spine path indicators* that make it possible to resolve such ambiguities and to ensure that the continuity (or lack of continuity) is made clear. Humdrum provides five special path indicators, one of which we have already encountered:

- a new spine may be introduced
- an existing spine may terminate (without continuing further)
- a previous spine may be split into two spines

- two or more spines may be amalgamated into a single spine
- the positions of two spines may be exchanged

Spine path indicators use the following signifiers: the plus sign (add a spine), the minus sign (terminate a spine), the caret (split a spine), the lower-case letter 'v' (join spines), and the lower-case letter 'x' (exchange spines). In addition to these, a *null interpretation* exists whose purpose is merely to act as a place-holder in interpretation records:

`*+`	add a new spine (to the right of the current spine)
`*-`	terminate a current spine
`*^`	split a spine (into two)
`*v`	join (two or more) spines into one
`*x`	exchange the position of two spines
`*`	null interpretation (place holder)

Spine Path Interpretations

Spine paths are types of interpretations, so the spine path indicators are encoded as Humdrum interpretations, using the asterisk signifier (*). The following examples illustrate a few possible path changes:

```
1   2     3
*   *-    *          (elimination of spine #2)
1   3

1   2     3
*   *x    *x         (exchange spines #2 and #3)
1   3     2

1   2     3
*   *^    *          (splitting of spine #2)
1   2a    2b   3

1   2     3
*   *v    *v         (amalgamation of spines #2 and #3)
1   2&3
```

Notice that in cases where two or more spines are amalgamated, the spines must be adjacent neighbors. For example, the arrangement below is forbidden by the Humdrum syntax since it is not clear whether spines #1 and #3 amalgamate into spine 'A' or spine 'B'.

```
1    2   3
*v   *   *v        (syntactically illegal)
A    B
```

In such cases, amalgamating the two outer spines can be accomplished by first using the exchange path signifier. Here we exchange spines #2 and #3 before amalgamating the original first and third spines:

```
1     2    3
*     *x   *x
*v    *v   *
1&3   2
```

In cases where the user wishes to amalgamate several spines, a number of interpretation records may be necessary. In the following example, spines #1 and #2 are first joined together (momentarily defining three spines: 1&2, 3, 4). In the subsequent interpretation record, spine #2 (previous spine #3) and spine #3 (previous spine #4) are then joined:

```
1     2     3    4
*v    *v    *    *
*     *v    *v
1&2   3&4
```

In addition, it is possible to join more than two spines at the same time:

```
1           2       3       4
*v          *v      *v      *v
1&2&3&4
```

In cases where a new spine is introduced, it is essential to indicate the exclusive interpretation that applies to the new data. Thus an 'add spine' indication must be followed by a second interpretation record:

```
1   2    3
*   *+   *         *     (add a new spine.)
*   *    **inter  *     (define exclusive interpretation for the
1   2    new      3      new spine.)
```

Failing to follow the introduction of a new spine by a subsequent exclusive interpretation is illegal.

The following examples illustrate a variety of more complex path redefinitions:

```
1     2    3    4
*v    *v   *^   *^
1&2   3a   3b   4a   4b
```

```
1       2    3    4    5
*       *-   *    *-   *
*v      *v   *v
1&3&5
```

```
1     2    3    4    5
*     *-   *    *^   *+
*     *    *    *    *      **new
*v    *v   *    *    *      *
1&3   4a   4b   5    new
```

```
1    2    3    4
*x   *x   *    *
*    *x   *x   *
*    *    *x   *x
2    3    4    1
```

Note that with judicious planning, the user can completely reconfigure all spines within a Humdrum file.

Syntactically, some path constructions are illegal; here are some examples of illegal constructions:

```
1    2        3
*v   *        *v
```
(The join-spine indication in spine #1 does not adjoin spine #3.)

```
1    2        3
*x   *x       *x
```
(No more than two exchange interpretations at a time.)

```
1    2        3
*x   *        *
```
(Must have two exchange interpretations together.)

```
1    2        3
*v   *        *
```
(Must have two or more join interpretations at a time.)

```
1    2        3
*    *
1    2
```
(Spine eliminated without using a termination interpretation.)

```
1    2        3
*    *        *+
1    2        3
```
(Adding a new spine should result in 4 interpretations.)

```
1    2
*    *        *-
```
(Cannot eliminate non-existent spine.)

```
1    2
*+   *
1    new      2
```
(New spine started without specifying new interpretation.)

```
1    2
*    *+
*    **inter  *
A    B        C
```
(Interpretation labels the wrong spine.)

The Humdrum Syntax: A Formal Definition

With the preceding background it is now possible to define formally a Humdrum representation. First we can define a Humdrum file. A Humdrum file must conform to one of the following:

1. A file containing *comments, data records* and *interpretations* with the restriction that no data record or local comment appears before the first *exclusive interpretation.*
2. A file containing *data records* preceded by at least one *exclusive interpretation.*
3. A file containing only *comments* and *interpretations* with the restriction that no local comments appear before the first interpretation.
4. A file containing only *interpretations* beginning with an exclusive interpretation.
5. A file containing only global *comments.*
6. A totally empty file (i.e. a file containing no records).

In addition, each spine in a Humdrum file must ultimately end with a path terminator (*–). Only global comments (or new exclusive interpretations) may occur following the termination of all spines. A property of Humdrum files is that the concatenation of two or more Humdrum files will always result in a Humdrum file.

Additional interpretations may be added throughout the file. Global comments may appear anywhere in the file. However, local comments are much more restricted: (1) Local comments may not appear until after the first interpretation record, and (2) The number of sub-comments in a local comment record must be equivalent to the number of currently active spines.

Comment	Either a global or local comment. Any record beginning with an exclamation mark.
Global comment	Any record beginning with two exclamation marks (!!).
Local comment	Any record beginning with one and only one exclamation mark (!). Every spine in that record must also begin with an exclamation mark.
Null comment	A comment record containing no commentary; only the appropriate exclamation mark(s) are present.
Interpretation	Either an exclusive or tandem interpretation. Any record beginning with an asterisk (*).
Exclusive interpretation	Any record beginning with one or more asterisks (*), where at least one spine begins with two asterisks.
Tandem interpretation	Any record beginning with a single asterisk (*) where none of the spines begin with two asterisks.
Path indicator	One of five special tandem interpretations *+ *- *v *^ *x found only in tandem interpretation records.
Null interpretation	An interpretation for a given spine or spines consisting of just the interpretation signifier (i.e., a single asterisk).
Data record	Any record that is not a comment or interpretation. Must contain the same number of tokens as the number of current spines.
Null token	The period (.) either alone on a single record or separated from other characters by a tab. Appears only in data records.
Null data record	A data record consisting only of null tokens.
Spine	A column-like "path" of information — including data records, local comments, and interpretations.

Humdrum Terminology

As a supplement to the above "positive" definition of the Humdrum syntax, we can also describe various inputs that do *not* conform to the Humdrum syntax:

An empty record.
A record containing only tabs.
A record beginning with a tab.
A record ending with a tab.
Any record containing two successive tab characters.
Any data record having fewer or more spines than the immediately preceding data record.
A record having only one join-spine indication.
A record having only one exchange-spine indication.
A record having more than two exchange-spine indications.

Some Illegal Humdrum Constructions

The *humdrum* Command

One of the most important commands in the Humdrum Toolkit is the **humdrum** command itself. This command is used to identify whether a file or other input stream conforms to the above Humdrum syntax. Where appropriate, the **humdrum** command issues error messages identifying the type and location of any syntactic transgressions. If no infractions are found, **humdrum** produces no output (i.e., in UNIX parlance "silence is golden"). All of the commands in the Humdrum toolkit assume that the inputs given to them conform to the Humdrum syntax. Whenever you encounter a problem, you should always test the input to assure that it is in the proper Humdrum format.

The examples given below provide further illustrations of Humdrum representations:

```
**form
Introduction
Exposition
Development
Recapitulation
Coda
*-

**American    **British
quarter       crotchet
eighth        quaver
dotted half   dotted minim
*-            *-

**Opus/No     **Year
23/1          1821
23/2          1821
23/3          1822?
24            1822
*-            *-
```

```
**recip   **diaton **accidental**stem-dir **kern
4         c        #           /          4c#/
8         d        .           /          8d/
8         e        .           /          8e/
2         f        #           /          8f#/
*-        *-       *-          *-         *-

**heart-rate
74
73
74
77
78
*-

**foreground
flute
*^
flute          violin1
*-             *
violin1
*^
violin1        bassoon
*              *^
violin1        bassoon  'cello
*              *        *^
violin1        bassoon  'cello trombone
*-             *-       *-     *
trombone
*^
trombone       trumpet
*-             *-
```

Reprise

This chapter has identified the formal structural and organizational features of the Humdrum syntax. The syntax provides a framework within which sequential symbolic data can be represented. Individual representation schemes map the ASCII character set (signifiers) to various music-related concepts (signifieds).

Each representation is designated by an exclusive interpretation. The corresponding data are organized in spines that may meander throughout the file. New spines may be added, spines joined together, exchanged, split, or terminated. Data are organized as tokens, although tokens can consist of multiple subtokens separated by single spaces. Null tokens can appear as place-holders where no specific data exists.

Free-form comments may be interspersed throughout the file. Global comments pertain to all spines whereas local comments pertain to individual spines. Additional interpretive information may be encoded using tandem interpretations. Both local comments and tandem interpretations may occur anywhere, but must be preceded in the spine by some exclusive interpretation.

Chapter 6

Representing Music Using ***kern* (II)

In this chapter we return to consider several more advanced topics related to the `**kern` representation.

Tuplets

As we saw in Chapter 2, `**kern` durations are represented using a reciprocal number notation. With the exception of the value zero, durations are represented by reciprocal numerical values corresponding to the American duration names: "1" for whole note, "8" for eighth, "32" for thirty-second, etc. The number zero (0) is reserved for the breve duration (a duration of twice the length of a whole note).

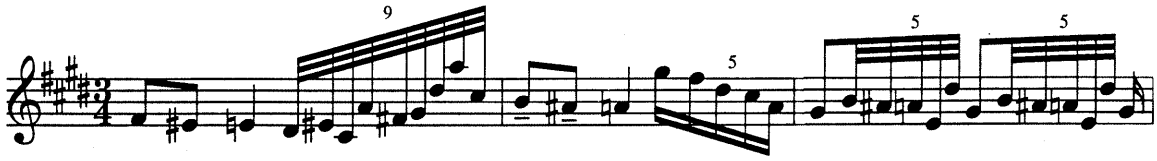
Dotted durations are indicated by adding the period character (.) immediately following the numerical value, hence "`8.`" signifies a dotted eighth-note and "`2..`" signifies a doubly dotted half-note. Any number of augmentation dots may follow the duration integer.

Triplet and other irregular durations are represented using the same reciprocal logic. Consider, for example, the quarter-note triplet duration. Three quarter triplets occur in the time of four quarters or one whole duration. If we divide a whole duration ("`1`") into three equal parts, each part has a duration of one-third. The corresponding reciprocal integer for 1/3 is 3, hence `**kern` represents a quarter-note triplet as a "third-note", `3`. Similarly, eighth-note triplets are represented by the integer 6 and sixteenth-note triplets are represented by the integer 12. Eighth-note quintuplets (5 in the time of 4) will be represented by the value 10 (a half duration divided by 5).

In general, the way to determine the `**kern` equivalent of an arbitrary "tuplet" duration is to multiply the number of tuplets by the total duration which they occupy. If 7 notes of equal duration occupy the duration of a whole-note ("`1`"), then each septuplet is represented by the value 7 (i.e. 1×7). A more extreme example is 23 notes in the time of a doubly dotted quarter. The appropriate `**kern` duration can be found by multiplying 4 by 23 (equals 92) and adding the appropriate augmentation dots. Thus "92.." is the correct `**kern` encoding for a note whose duration is 23 notes in the time of a doubly-dotted quarter.

By way of illustration, example 6.1 shows an excerpt from Sigfrid Karg-Elert's *Caprices* Op. 107, No. 23 for solo flute. The work is in 3/4 meter. The last beat of the first measure has 9 notes in the time of a quarter duration. The reciprocal encoding is 9×4 or 36th notes.

Example 6.1. Karg-Elert, *Caprices* Op. 107, No. 23.




```
**kern
*clefG2
*k[f#c#g#d#]
*M3/4
=
(8f#
8e#
4en)
(36d#
36e#
36c#
36a
36f##
36g#
36dd#
36aa
36cc#
=
8b~
8a#~
4an)
(20gg#
20ff#
20dd#
20cc#
20a
=
8g#)
(40b
40a#
40an
40e
40dd#
8g#)
(40b
40a#
40an
40e
40dd#
20g#)
*-
```

Another set of tuplets appears in the last beat of the second measure. In this case, five notes are played in the time of a quarter, hence the reciprocal durations are 5×4 or 20th notes.

In the last measure of the example, five notes are played in the time of an eighth-note: 5×8 or 40th notes. Note that the 40th notes are half the duration of the 20th notes in the previous measure.

Other noteworthy observations regarding example 6.1: notice the tenuto markings (`~`) on the first two notes of the second measure. Also notice that the duration of the last note in the example is indeterminate. It could be a sixteenth note, a quintuplet sixteenth, or some other duration.

Grace Notes, Gruppettos and Appoggiaturas

The `**kern` representation also allows for the encoding of grace notes (acciaccaturas, non-canonical gruppettos, and appoggiaturas.

Depending on the expected analytic application, one way to handle these notational devices is to encode the notes according to the manner in which they are typically performed. Alternatively, since the component notes of an expanded ornament are viewed as embellishments that hold potentially less analytic status, a special designation for these notes can be useful for certain types of studies.

Grace notes (acciaccaturas) are visually rendered as miniature notes with a slash drawn through the stem. In the `**kern` representation these notes are treated as "durationless" notes and are designated by the lower-case letter "`q`". Hence, the token "`G#q`" denotes a G#3 grace note with an indeterminate duration.

Non-canonical *gruppettos* are miniature (non-cue) notes (typically appearing in groups) whose stems do not contain a slash, and whose notated durations cause the total notated duration for the measure to exceed the prevailing meter. These gruppetto notes are encoded as notes retaining their notated durations, but all such notes are also designated by the upper-case letter "`Q`". Hence, a miniature sixteenth-note middle C would be encoded as "`16cQ`".

When processed by various tools in the Humdrum Toolkit, these notes may be treated as equivalent to their notated durations. Alternatively, in some types of processing these notes may be discarded. For example, the Humdrum **timebase** command (described in Chapter 13) eliminates acciaccaturas and gruppetto notes. Note that data records containing acciaccaturas or gruppetto notes must *not* also include normal notes.

In the case of *appoggiaturas,* `**kern` requires that they be encoded as performed. An appropriate duration is assigned to the appoggiatura according to common performance practice. The duration of the subsequent note is reduced by a corresponding amount. The status of the two notes forming the appoggiatura is nevertheless marked. The appoggiatura note itself is designated by the upper-case letter "`P`", whereas the subsequent note (whose notated duration has been shortened) is designated by the lower-case letter "`p`".

Multiple Stops

In the **kern representation, spines typically represent individual musical parts or voices. Occasionally, a nominally single "part" contains more than one concurrent note. A good example of such a situation occurs when a violin plays a double stop.

The **kern representation provides three different ways of representing such situations: (1) multiple stops, (2) spine splitting and rejoining, and (3) introduction and retiring of a momentary "part". Each of these representation methods captures a different way of interpreting the music. The best representation will depend on the editorial or processing goal.

First we will consider the generic Humdrum *subtoken.* A data token becomes a multiple stop when two or more subtokens are present, separated by single spaces. By way of illustration, the following **kern data represents a scale played in ascending thirds:

```
**kern
4c 4e
8d 8f
8e 8g
8f 8a
8g 8b
8a 8cc
8b 8dd
4cc 4ee
*-
```

An important restriction for multiple stops in the **kern representation is that they must encode notes of the same duration. In the above example, the left and right components of the multiple stop always share the same duration. (Note that this is a restriction of multiple stops in the **kern representation only, and does not necessarily apply to other Humdrum representations.)

Notice that multiple stops are represented with a single spine (that is, there are no tabs present).

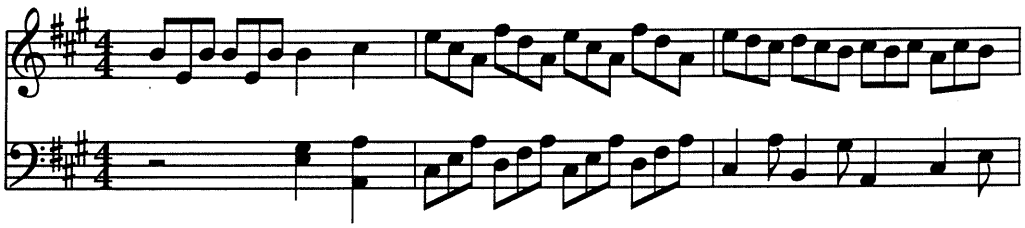
Multiple stops may occur at any point in a **kern spine. For example, the following **kern data represents a scale that begins and ends with chords:

```
**kern
4c 4e 4g
8d
8e
8f
8g
8a
8b
4cc 4ee 4gg 4ccc
*-
```

The first chord has been encoded as a triple stop, whereas the last chord has been encoded as a quadruple stop. Notice once again that all of the notes within a multiple stop must have the same duration. If the durations of the concurrent notes differ, then one must use Humdrum spine path indicators (see below).

Example 6.2 illustrates a musical context where multiple stops may be appropriate. The sample passage is from a keyboard work by Telemann. The work is almost entirely in two parts with only occasional chords. Since the chords always contain notes of equal duration, they can be encoded as double-stops within a single part.

Example 6.2. Telemann, *Kleine Fantasien für Klavier* No. 7.



```
**kern      **kern
*clefF4     *clefG2
*k[f#c#g#]*k[f#c#g#]
*M4/4       *M4/4
!! Allegro
=2          =2
2r          12b
.           12e
.           12b
.           12b
.           12e
.           12b
4E 4G#      4b
4AA 4A      4cc#
=3          =3
12C#        12ee
12E         12cc#
12A         12a
12D         12ff#
12F#        12dd
12A         12a
12C#        12ee
12E         12cc#
12A         12a
12D         12ff#
12F#        12dd
12A         12a
=4          =4
6C#         12ee
.           12dd
12A         12cc#
6BB         12dd
.           12cc#
12G#        12b
```

```
4AA         12cc#
.           12b
.           12cc#
6C#         12a
.           12cc#
12E         12b
=5          =5
*-          *-
```

When encoding multiple stops in the `**kern` representation, some note attributes should be encoded for each note in the multiple stop, whereas other markings should be encoded only once for the entire multiple stop. Note-related attributes such as articulation marks, stem directions, and ties should be encoded for each note in the multiple stop. By contrast, phrase marks, slurs, and beamings should be encoded once for the entire multiple stop. Example 6.3 provides a contrived example that illustrates these conventions. Notice that the first double stop encodes a single open phrase (i.e. '`{`') and the last double stop encodes a single close phrase ('`}`'). Similarly, the slur in the middle of the phrase has been encoded once. However, the staccato markings have been encoded for both notes in each of the double stops. Similarly, separate ties have been encoded for both notes in the double stop.

Example 6.3. Attribute encoding for `**kern` multiple stops.

```
**kern
{4c 4e
8dL 8f
8eJ 8g
(8f' 8a'
8g' 8b')
8.a 8.ccL
16b 16ddk
[4cc [4ee
4cc] 4ee]}
*-
```

Further Examples

Example 6.4 shows an excerpt from a Chopin *Etude* that illustrates a number of subtle features in the `**kern` representation.

The opening measure consists solely of gruppetto notes; each has been designated by the upper-case letter 'Q'. The double-barline has been indicated with two thin lines. Two tandem interpretations encode the meter signature (`*M3/4`) and the tempo (`*MM66`). Note that metronome markings using the `*MM` interpretation are always given in quarter-durations per minute. If the metronome marking had been given as half-note equals 48, then the tempo interpretation would be given as `*MM96` — i.e. quarter-note equals 96 beats per minute.

Two grace-notes are evident in the passage. The first occurs just after the double-bar, and the second occurs three measures later. These notes have been encoded with duration values, but are designated by the lower-case letter '`q`'. Notice that grace-notes are always treated as "durationless" notes. Grace notes must always be encoded on a separate data record. Concurrent grace notes can be encoded on the same data record, but grace notes must never share the same data record with a

regular note. Another peculiarity of grace notes is that they are always assumed to be slurred to the ensuing note (if there is a subsequent note). Thus slurs should not be encoded as part of the grace note.

The grace-note E-natural coincides with the end of the first phrase and the beginning of the second phrase. The phrases are said to be "elided" (overlapping). Since `**kern` phrases are represented by open and close braces, the ampersand character is used to indicate elisions.

Notice that the 'inner' accompaniment chords have been encoded as double stops. This is possible because the notes in these double stops are all the same duration.

Finally, notice how the triplet eighth-notes (encoded as duration '12') have been interleaved with the concurrent eighth-note figures so that the onsets are ordered in the correct temporal sequence.

Example 6.4. Chopin, *Etude* Op. 27, No. 7.

```
!! Lento
**kern    **kern    **kern
*staff2   *staff1   *staff1
*clefF4   *clefG2   *clefG2
{4GG#Q    .         .
4.D#Q     .         .
8C#Q      .         .
8C#Q      .         .
8BB#Q     .         .
4.AQ      .         .
8G#Q      .         .
16F##Q    .         .
16G#Q     .         .
16AQ      .         .
16G#Q     .         .
16BnQ     .         .
```

```
16AQ      .         .
16EQ      .         .
16E#Q     .         .
16F#Q     .         .
16E#Q     .         .
16G#Q     .         .
16F#Q     .         .
16AQ      .         .
16G#Q     .         .
16C#Q     .         .
16D#Q     .         .
=||       =||       =||
*MM66     *MM66     *MM66
*M3/4     *M3/4     *M3/4
&{(Enq}   .         .
2e)       8r        2.r
.         8g# 8cc#  .
.         8g# 8cc#  .
.         8g# 8cc#  .
8.d#      8g# 8cc#  .
.         8g# 8cc#  .
16c#      .         .
=3        =3        =3
4..cc#    8e 8a     {2ee
.         8e 8a     .
.         8e 8a     .
.         8e 8a     .
32d#      .         .
32c#      .         .
.         8e 8a     8.dd#
.         8e 8a     .
.         .         16cc#
=4        =4        =4
[4A       8d# 8f#   4cc#
.         8d# 8f#   .
16A]      8d# 8f#   4b#
16G#      .         .
16F##     8d# 8f#   .
16G#      .         .
16Bn      8d# 8f#   4cc#
16A       .         .
16F#      8d# 8f#   .
16D#      .         .
=5        =5        =5
4BB#}     8d# 8f#   4dd#
.         8d# 8f#   .
8r        8d# 8f#   4..g#}
{16FF##^  8d# 8f#   .
16GG#     .         .
```

```
BBnq        .          .
16AA        8B# 8f#    .
16GG#       .          .
16C#        8B# 8f#    .
16D#        .          {16g#
=6          =6         =6
4E          8r         4g#
.           8c#        .
4CC#}       8c# 8e     4cc#
.           8c# 8e     .
(8.G#       8c# 8e     4bn
.           8c# 8e     .
16.G#
=7          =7         =7
4G#)        8c# 8e     4b
.           8c# 8e     .
(12FF#      8c# 8e     4a
12C#        .          .
.           8c# 8e     .
12F#        .          .
12G#        8c# 8e     4g#
12B         .          .
.           8c# 8e     .
12A         .          .
=8          =8         =8
*-          *-         *-
```

Reprise

This chapter completes our survey of the `**kern` representation. We have noted an number of subtleties related to encoding tuplets, multiple stops, gruppettos, acciaccaturas, elided phrases, and spine path changes.

A more complete description of the `**kern` representation may be found in the *Humdrum Reference Manual.*

Chapter 7

MIDI Output Tools

MIDI is a standard method for exchanging information between sound synthesizers and controlling computers. Humdrum provides a number of MIDI-related tools that allow MIDI data to be input and output. In this chapter we will discuss the output-related tools: **midi**, **perform**, **smf** and **tacet**. MIDI input tools are discussed in Chapter 30.

The ***MIDI* Representation

Humdrum provides a `**MIDI` representation that closely parallels the commercial MIDI specification but conforms to the Humdrum data format. This representation provides an intermediate format; using Humdrum commands such as **perform** and **smf**, true MIDI data can be generated. However, since the `**MIDI` representation conforms to the Humdrum syntax, the data can be manipulated, modified and searched in the same way as other Humdrum data. For example, we can use **grep** to search MIDI data, etc.

MIDI is a type of "tablature" notation. It describes a set of performance actions rather than specifying either the sounded result or the analytic notation. MIDI represents note-related events for various "channels." MIDI events include note-on, note-off, key-velocity, after-touch, control codes, and system-exclusive codes. The original commercial MIDI standard is unable to represent many other musically pertinent signifiers such as pitch spelling (e.g., F-sharp versus G-flat), stem directions, ties, slurs, phrasing, etc. In addition, MIDI does not represent rests.

A simple `**MIDI` example is given below. It consists just a single note (middle C):

```
!! A single MIDI note.
**MIDI
*Ch1
54/60/64
80/-60/64
*-
```

Notice that there are two `**MIDI` data tokens: one to specify note-on and one to specify note-off events. Each `**MIDI` data token consists of three elements or components, delimited by a slash

character (/). The first element in the data token represents the number of clock ticks before the current event is to occur. The absolute duration of a single clock tick is determined by the `**MIDI` clock speed, which is variable. In our example, 54 clock ticks elapse before the note is turned on, and another 80 clock ticks elapse before the note is turned off. The initial pause before the first note begins is not necessary; however, many MIDI cards introduce a brief delay before beginning to send the first data in a stream. The leading pause prevents a "rushed" burst of initial notes.

The second element in a data token represents the `**MIDI` key number. Successive numbers refer to semitone pitches where middle C is designated key `60`. Key-off events are represented by negative integers, so `-60` means turn off key `60`. Key numbers range between 1 and 127.

The third element in a data token represents the `**MIDI` key velocity. MIDI instruments normally interpret key velocity as dynamic or accent information. Higher key-velocity values are associated with accented notes. In the case of key-off events, the key-velocity component represents the key-up velocity.

Unlike most other Humdrum representations, the `**MIDI` format requires two data tokens for each note. The following example shows a three note C-major triad. The `**kern` data is shown in the left spine with the corresponding `**MIDI` data in the right spine. Notice that each data token consists of three subtokens, one for each note:

```
!! A C-major triad.
**kern      **MIDI
*           *Ch1
2c 2e 2g    72/60/64 72/64/64 72/67/64
.           144/-60/64 144/-64/64 144/-67/64
*-          *-
```

Middle C corresponds to MIDI key 60; The pitches E4 and G4 correspond to MIDI keys 64 and 67 respectively. The half-note durations result in a delay of 144 clock ticks before the key-off commands are executed. Notice that all key-off events occur simultaneously.

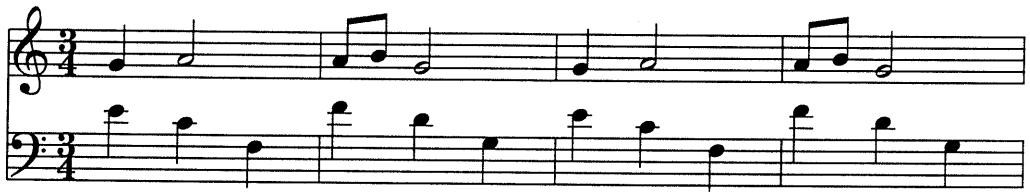
Suppose we add a second chord consisting of the dyad D4 and F4. The dyad begins at the same time that the C major triad ends. This results in five subtokens in the corresponding `**MIDI` data record. Three key-off events are synchronous with two key-on events:

```
!! Two chords.
**kern      **MIDI
*           *Ch1
2c 2e 2g    72/60/64 72/64/64 72/67/64
4d 4f       144/-60/64 144/-64/64 144/-67/64 144/62/64 144/65/64
.           72/-62/64 72/-65/64
*-          *-
```

Notice that the difference in duration between the half-notes and quarter-notes is reflected when the notes are turned *off* rather than when the notes are turned *on*.

Example 7.1 illustrates a slightly more complex excerpt from the beginning of Darius Milhaud's *Touches Blanches*.

Example 7.1 Excerpt from Darius Milhaud's *Touches Blanches*



```
!!!: Milhaud, D.
!!!OTL: Touches Blanches
**kern      **kern      **MIDI              **MIDI
*staff2     *staff1     *Ch1                *Ch1
*clefF4     *clefG2     *clefF4             *clefG2
*k[]        *k[]        *k[]                *k[]
*M3/4       *M3/4       *M3/4               *M3/4
=1-         =1-         =1-                 =1-
4e          (4g         72/64/64            72/67/64
4c          2a)         72/-64/64 72/60/64  72/-67/64 72/69/64
4F          .           72/-60/64 72/53/64  .
=2          =2          =2                  =2
4f          (8a         72/-53/64 72/65/64  72/-69/64 72/69/64
.           8b          .                   36/-69/64 36/71/64
4d          2g)         36/-65/64 36/62/64  36/-71/64 36/67/64
4G          .           72/-62/64 72/55/64  .
=3          =3          =3                  =3
4e          (4g         72/-55/64 72/64/64  72/-67/64 72/67/64
4c          2a)         72/-64/64 72/60/64  72/-67/64 72/69/64
4F          .           72/-60/64 72/53/64  .
=4          =4          =4                  =4
4f          (8a         72/-53/64 72/65/64  72/-69/64 72/69/64
.           8b          .                   36/-69/64 36/71/64
4d          2g)         36/-65/64 36/62/64  36/-71/64 36/67/64
4G          .           72/-62/64 72/55/64  .
.           .           72/-55/64           72/-67/64
*-          *-          *-                  *-
```

The `**MIDI` representation always expects a tandem interpretation indicating the MIDI channel assignment. In Example 7.1 both parts have been assigned to channel 1. Once again, simultaneous key-on and key-off events often appear as double-stops. Also notice that an additional data record is required at the end of the passage in order to turn off the final notes.

The *midi* Command

The **midi** command converts Humdrum `**kern` data into Humdrum `**MIDI` data. By way of example, the above `**MIDI` data can be generated as follows:

```
midi inven05.krn > inven05.hmd
```

The `.hmd` filename extension is a common way of designating Humdrum `MIDI` data.

Since the `**kern` representation does not encode key-velocity information, the **midi** command assumes a default key velocity of 64 (from a range of 1 to 127). If the input is monophonic, **midi** will also allow the user to set a fixed note duration using the **-d** option. This is useful for auditing notes that do not have duration values. For example, a Gregorian chant might be represented without durations. The following command takes a file containing a 12-tone row (pitch information only) and produces a `**MIDI` output where all notes assigned to a quarter duration:

```
midi -d 4 tonerow > tonerow.hmd
```

The most common use of `**MIDI` data is to create a standard MIDI file using the **smf** command, or to listen to the output using the **perform** command. In some cases, it is useful to carry out processing of `**MIDI` data itself.

The *perform* Command

The **perform** command allows the user to listen to synthesized performances of `**MIDI`-format input. When invoked, **perform** provides a simple interactive environment suitable for proof-listening and other audition tasks.

The **perform** command accepts any Humdrum input; however, only `**MIDI` spines present in the input stream are performed. Non-MIDI spines are simply ignored and do not affect the sound output. The **perform** command generates serial MIDI data which are sent directly to a MIDI controller card or on-board sound-card.

The **perform** command is typically the last command in a pipe preceded by the **midi** command. For example, a `**kern`-format score can be heard using the following command:

```
midi clara.krn | perform
```

When invoked, the **perform** command reads in the entire input into memory. This allows the user to move freely both forward and backward through the MIDI score.

The **perform** command provides a set of interactive commands that allow the user to pause and resume playback, to change tempo, to move to any measure by absolute or relative reference, and to search forward or backward for commented markers. The **perform** command remains active until either the end of the score is reached or the user terminates performance by typing the letter 'q' or the escape key (ESC).

Playback can be paused by typing the space-bar and resumed by typing any key. Typing the carriage return by itself will return to the beginning of the score and re-initiate playback. If a number is typed before pressing the carriage return then **perform** will search for a corresponding measure number and initiate playback from that measure. Other commands are provided that allow moving forward or backward a specified number of measures.

In the default operation, **perform** echoes all global comments on the screen as the comments are encountered in the input. For inputs containing appropriate annotations, the echoing of comments

can provide useful visual markers or reminders of particular moments in the sound output. Whether or not global comments are echoed on the standard output, users can use the **perform** forward-search (/) or backward-search (?) commands to move directly to a particular commented point in the score. For example, if an input contains a global comment containing the character string "`Second theme`," then the user can move immediately to this position in the input by entering the following command:

```
/Second theme
```

Similarly, backward searches can be carried out by typing the question mark (?) rather than the slash. If the search is successful, playback continues immediately from the new score position.

Data Scrolling During Playback

The **midi** command provides a useful **-c** option that causes each data record to be repeated as a comment. For example, when the **-c** option is used a sequence of data records such as the following:

```
4C    4E   4G   4c
4D    4F   4G   4B
4AA   4E   4A   4c
```

is transformed to:

```
4C      4E   4G   4c
!!4C    4E   4G   4c
4D      4F   4G   4B
!!4D    4F   4G   4B
4AA     4E   4A   4c
!!4AA   4E   4A   4c
```

Since, by default, the **perform** command echoes all global comments on the screen during playback, this means that the Humdrum data will also appear on the screen as it is being played. In addition, the commented data records are accessible to the forward- and backward-search commands. For example, in the `**kern` representation, pauses are indicated by a semicolon; hence the user might search for the next pause symbol by typing:

```
/;
```

Similarly, the user could search for a particular pitch, e.g.

```
/gg#
```

Since the **perform** command accepts any Humdrum input, other Humdrum data may be used for searching. For example, the input data might contain melodic interval data (see Chapter 11), allowing the user to search for a particular interval such as a diminished octave:

```
/d8
```

If the string pattern is found in the input, **perform** will move immediately forward (or backward) to the next occurrence and begin playing from that point.

Changing Tempo

During playback, the tempo can be modified by typing the greater-than (>) and less-than (<) signs to increase or decrease the tempo respectively. In addition to modifying the tempo interactively, the performance tempo may be specified either in the command line or in the input Humdrum representation. The tempo may be specified on the command line by using the **-t** option. For example, the following command causes the file `Andean` to be performed at half tempo:

```
midi Andean | perform -t 0.5
```

Performing at fast speeds can often be useful when scanning for a particular passage.

Tempo specifications may be present in the input data via the tandem interpretation for metronome marking (e.g. `*MM96`). If no tempo information is available, **perform** uses a default tempo of 66 quarter-notes per minute.

The *tacet* Command

In rare circumstances, ciphers (stuck notes) can occur during MIDI performances; for instance, an intermittently functioning MIDI cable may fail to convey a "note-off" instruction to an active synthesizer. The **p** command ("panic") turns off all active notes. Should a cipher remain after terminating the **perform** command, the Humdrum **tacet** command can be used to send "all-notes-off" commands on all MIDI channels.

In Chapter 12, we will see how **perform** can be used in conjunction with other commands (such as **extract** and **yank**) to listen selectively to specific parts or passages. In Chapter 21 we will use **perform** in conjunction with the **patt** command to listen to patterns (such as harmonic, rhythmic and melodic patterns) found in some repertory.

The *smf* Command

Another MIDI-related tool is the **smf** command. This command allows the user to create "standard MIDI files" from Humdrum `**MIDI`-format files. Standard MIDI files are industry-standard binary files that can be imported by a variety of MIDI applications software packages on many different platforms, including sequencer programs and most music notation packages.

The **smf** command translates only `**MIDI` input spines; all non-`**MIDI` spines are simply ignored. Suppose we begin with a `**kern`-format file named `joplin`. We can create a standard MIDI file as follows:

```
midi joplin | smf > joplin.smf
```

The **smf** command provides two options. The **-t** option allows the user to set the tempo, whereas

the **-v** option allows the user to specify a default MIDI key velocity. See the *Humdrum Reference Manual* for details.

Reprise

In this chapter we have learned how Humdrum data can be output as MIDI data. Humdrum provides a `**MIDI` representation that closely parallels MIDI but remains in conformity with the Humdrum syntax. This means that the data can still be processed with other Humdrum tools (as we will see in later chapters).

The **midi** command can translate `**kern` data to `**MIDI` and the **perform** and **smf** commands can be used to generate true MIDI data for listening. The **perform** command provides a simple interactive command-line sequencer for playing whatever input is provided. The **smf** command generates standard MIDI files that can be used to transport MIDI data to a vast array of commercial and non-commerical applications software. In Chapter 30 we will explore some of the Humdrum tools for inputting MIDI data into Humdrum.

Chapter 8

The Shell (I)

When you type commands, they are interpreted by a command *shell.* The shell is a program that interprets user commands before passing them along to be executed. Command shells are quite sophisticated and provide a number of useful features. Although there is a lot to learn about shells, we will explore only those features that facilitate use of Humdrum. This chapter is the first of four chapters scattered throughout this book where we will pause and examine some of the more pertinent and valuable features of the shell.

In UNIX environments, many different shells have been developed over the years. The original UNIX shell was the *C-shell* — a shell whose syntax is similar to the C programming language. A later shell was developed by Stephen Bourne and is known as the *Bourne shell.* Subsequent improvements by David Korn resulted in the *Korn shell.* The Bourne shell was improved in light of many features introduced in the Korn shell, and resulted in the *Bourne Again Shell* — known as *Bash.* The Korn and Bash shells are the most popular and powerful of the current generation of shells. Although they were originally developed for the UNIX operating system, these shells are also available for DOS, Macintosh, Windows, Windows 98 and many other operating systems.

Shells themselves are advanced programming languages that provide complex control structures. When you type a command, you are already writing a program — although most of your programs are just one line in length.

Shell Special Characters

The shell interprets a number of characters in a special manner. When you type a command, you should know that most shells treat the following characters as having a special meaning: the octothorpe (#), the dollar-sign ($), the semicolon (;), the ampersand (&), the verticule (|), the asterisk (*), the apostrophe ('), the greve (`), the greater-than sign (>), the less-than sign (<), the question-mark (?), the double-quote ("), and the backslash (\). We'll consider the function of each of these characters one at a time.

File Redirection (>)

Some of the special shell characters have already been discussed. The greater-than-sign (>) is a *file redirection operator.* It must be followed by a user-specified filename; any output from the preceding command is placed in the specified file. For example, the following command sorts the file `inputfile` and places the sorted result in the file named `outputfile`:

```
sort inputfile > outputfile
```

If the file `outputfile` already existed, its contents will be destroyed and over-written with the new output. Be careful not to assign the output to the same file as the input, since this will destroy the original input file.

Sometimes it is useful to *add* the results of an operation to some already existing file. The double greater-than-sign (>>) causes the new output to be appended to any data already in the named file. For example, the following command sorts the file `inputfile` and adds the sorted lines to the end of the file named `outputfile`. If the `outputfile` does not already exist, the command will create it.

```
sort inputfile >> outputfile
```

Pipe (|)

The vertical bar (|) is interpreted by the shell as a 'pipe.' Pipes are used to join the output of one command to the input of a subsequent command. For example, in the following construction, the output of `command1` is routed as the input to `command2`:

```
command1 | command2
```

There is no practical limit to the length of a pipeline. Several pipes can be used to connect successive outputs to ensuing commands:

```
command1 | command2 | command3 | command4
```

Shell Wildcard (*)

The asterisk is interpreted by the shell as a "filename wildcard." When it appears by itself, the asterisk is 'expanded' by the shell to a list of all files in the current directory (in alphabetical order). For example, if the current directory contained just three files: `alice`, `barry` and `chris` — then the following command would be applied to all three files in consecutive order:

```
command * > people
```

The file expansion occurs at the moment when the command is invoked. So although the file `people` is added to the current directory, it is not included as its own input. However, if the above command was executed a second time, then the file expansion would include `people` — even as the file itself is over-written to receive the output. Including the output file as input is

never a good idea.

Comment (#)

The octothorpe character (#) indicates a shell *comment*. Any characters following the # (up to the end of the line) are simply ignored by the shell. The following is not a command:

```
#grep OTL: filename
```

The comment can begin anywhere in the line. Here the comment begins after the filename:

```
grep OTL: filename    # (Search for Humdrum titles.)
```

Escape Character (\)

Sometimes we would like to have a special character treated literally. For example, suppose we wanted to search for records containing sharps in a `**kern` file. The following command will not work because the shell will insist on interpreting the octothorpe as beginning a comment:

```
grep # filename
```

There are several ways to "turn off" the special meaning of a character. The simplest way is to precede the character by a backslash (\) as in the following command:

```
grep \# filename
```

The backslash character itself can be treated literally by preceding it with another backslash. For example, the following command searches for down-stems in a `**kern` file:

```
grep \\ filename
```

Escape Quotations ('...')

Another way of escaping the special meaning of shell characters is to place the material in single quotes. For example, we can escape the meaning of the octothorpe (#) by preceding and following it by single quotes:

```
grep '#' filename
```

Single quotes are especially useful for binding spaces. For example, the following command searches for the phrase "Lennon and McCartney" in a file named `beatles`:

```
grep 'Lennon and McCartney' beatles
```

If the single quotes are omitted, the command means something completely different. The following command searches for the string "Lennon" in three files named `and`, `McCartney` and `bea-`

```
tles:

    grep Lennon and McCartney beatles
```

A common mistake is to fail to match quotation marks in a command. The shell will assume that the command is incomplete until all quotation marks are matched (both single quotes and double quotes). In the following example, we have failed to match the quotation mark. When we press the return key, the shell responds with a change of prompt indicating that it is waiting for us to complete the command.

```
grep '# inputfile > outputfile
>
```

Command Delimiter (;)

The semicolon (;) indicates the end of a command. Its presence allows more than one command to be typed on a single line. For example, the following line:

```
command1 ; command2
```

is logically identical to:

```
command1
command2
```

When both commands appear on the same line, they are still executed sequentially, so the second command doesn't begin until the first is completed. Although the ability to place two or more commands on a single line may seem redundant, there are a number of circumstances where this feature proves useful.

Background Command (&)

After typing a command, the command begins executing as soon as you type the carriage return or "enter" key. When the command has finished executing, the shell will display a new command prompt. Sometimes a command can take a long time to execute so it will be awhile before the prompt is displayed again. Unfortunately, you must wait for the prompt before you can type a new command. On multitasking systems it is possible for the computer to execute more than one command concurrently. The ampersand (&) can be used to execute a command as a *background process*. When a command is ended by an ampersand, the shell creates an independent process to handle the command, and the shell immediately returns with a prompt for a new command from the user. UNIX systems provide sophisticated mechanisms for controlling concurrent processing of commands. For further information concerning these features, refer to a UNIX reference book.

Shell Command Syntax

Shell commands follow a special syntax. There are six possible components to a common command:

1. the command name,
2. one or more options,
3. one or more option parameters,
4. a command argument,
5. one or more input file names,
6. output redirection.

Each of these components is separated by 'blank space' (tabs or spaces). A command begins with the command name — such as **uniq**, **sort**, or **pitch**. A command argument is a special requirement of only some commands. A good example of a command argument is the search pattern given to the **grep** command. In the following command, **grep** is the command name, "Lennon" is the command argument and `beatles` is the input file name:

```
grep Lennon beatles
```

For most commands, it is possible to process more than one input file. These files are simply listed at the end of the command. For example, the following **grep** command searches for the string "McCartney" in the file `beatles` and in the file `wings`:

```
grep McCartney beatles wings
```

Most commands provide *options* that modify the behavior of the command in some way. Command options are designated by a leading dash character. The specific option is usually indicated by a single alphabetic letter, such as the **-b** option (spoken: "dash-B" option). In the **uniq** command, the **-c** option causes a count to be prepended to each output line. In the following command, **uniq** is the command name, **-c** is the option, and `ghana32` is the name of the input file:

```
uniq -c ghana32
```

In many cases, the option is followed by a *parameter* that specifies further information pertaining to the invoked option. In the following command, **recode** is the command name, **-f** is the option, **reassign** is the parameter used by the **-f** option, and **gagaku** is the name of the input file:

```
recode -f reassign gagaku
```

Options and their accompanying parameters must be separated by blank space (i.e. one or more spaces and/or tabs). If more than one option is invoked, and none of the invoked options require a parameter, then the option-letters may be combined. For example, the **-a** and **-b** options might be invoked as **-ab** (or as **-ba**) — provided neither option requires a parameter.

Whenever an option requires a parameter, the option must be specified alone and followed immediately by the appropriate parameter. For example, in the following command, the command name is **trans**, the **-d** option is followed by the numerical parameter **3**; the parameter for the **-c** option is the number **4** and the input file is named **gambia21**.

```
trans -d 3 -c 4 gambia21
```

Since numerical parameters can sometimes be negative, it can be difficult to discern whether a negative number is a parameter or another option. In the following example, the **–3** is a parameter to the **-d** option rather than an option by itself.

```
trans -d -3 -c 2 gambia21
```

Output Redirection

Most commands support several input and output modes. Input to a command may come from three sources. In many cases the input will come from one or more existing files. Apart from existing files, input may also come from text typed manually at the terminal, or from the output of preceding commands. When input text is entered manually it must be terminated with an end-of-file character (control-D) on a separate line. (On Microsoft operating systems the end-of-file character is control-Z.) When input is received from preceding commands, the output is sent via a UNIX pipe ('|') as discussed above.

The different ways of providing input to a command are illustrated in the following examples. In the first example, the input (if any) is taken from the terminal (keyboard). In the second example, the input is *explicitly* taken from a file named `input`. In the third example, the input is *implicitly* taken from a file named `input`. In the fourth example, the input to **command2** comes from the output of **command1**.

```
command
command < input
command input
command1 | command2
```

Outputs produced by commands may similarly be directed to a variety of locations. The default output from most commands is sent to the terminal screen. Alternatively, the output can be sent to another process (i.e. another command) using a pipe (|). Output can also be stored in a file using file redirection operator ('>') or *added* to the end of a (potentially) existing file using the file-append operator ('>>'). In the first example below, the output is sent to the screen. In the second example, the output is sent to the file `outfile`; if the file `outfile` already exists, its contents will be overwritten. In the third example, the output is appended to the end of the file `outfile`; if the file `outfile` does not already exist, it will be created. In the fourth example, the output is sent as input to the command **command2**.

```
command
command > outfile
command >> outfile
command1 | command2
```

When two or more commands have their inputs and outputs linked together using the pipe operator (|), the entire command line is known as a *pipeline.* Pipelines occur frequently in Humdrum applications.

Tee

A special shell command known as **tee** can be used to clone a copy of some output, so that two identical output streams are generated. In the first example below, the output is piped to **tee** which writes one copy of the output to the file `outfile` and the second copy appears on the screen. In the second example, the output from **command1** is split: one copy is piped to **command2** for further processing, while an identical copy is stored in the file `outfile1`; if the file `outfile1` already exists, its contents will be overwritten. In the third example, the append option (**-a**) for **tee** has been invoked — meaning that the output from `command` will be added to the end of any existing data in the file `outfile`. If the file `outfile` does not already exist, it will be created.

```
command | tee outfile
command1 | tee outfile1 | command2 > outfile2
command | tee -a outfile
```

The **tee** command is a useful way of recording or diverting some intermediate data in the middle of a pipeline.

Reprise

In this chapter we have noted that the shell interprets certain characters in a special way. We learned about the octothorpe (#), the ampersand (&), the verticule (|), the asterisk (*), the apostrophe ('), the greater-than sign (>), the semicolon (;), and the backslash (\). In a later chapter we'll discuss the remaining special characters: the dollar-sign ($), the apostrophe ('), the less-than sign (<), the question-mark (?), and the double-quote ("),

We have also reviewed the syntax for UNIX commands. Commands can include components such as the *command name*, *options*, *parameters*, *command arguments*, *input files* and *output redirection*.

Chapter 9

Searching with Regular Expressions

A common task in computing environments is searching through some set of data for occurrences of a given pattern. When a pattern is found, various courses of action may be taken. The pattern may be copied, counted, deleted, replaced, isolated, modified, or expanded. A successful pattern match might even be used to initiate further pattern searches.

In Chapter 3 we introduced simple searching using the **grep** command. We used **grep** to search for strings of characters that match a particular pre-defined string. This chapter describes the full power of *regular expressions* for defining complex patterns of characters. Becoming skilled with regular expressions is perhaps the principal foundation for productive use of Humdrum. Regular expressions can be used to define patterns in any representation, and are widely used in many UNIX and Humdrum tools.

Regular expression syntax provides a standardized method for defining patterns of characters. Regular expressions are restricted to common text characters including the upper- and lower-case letters of the alphabet, the digits zero to nine, and other characters typically found on computer keyboards.

Regular expressions will not allow users to define every possible musical pattern of potential interest. In particular, regular expressions cannot be used directly to identify deep-structure patterns from surface-level representations. However, regular expressions are quite powerful — much more powerful than they appear to the novice user. Not all users will be equally adept at formulating an appropriate regular expression to search for a given pattern. As with the study of a musical instrument, practise is advised.

Literals

The simplest regular expressions are merely literal sequences of characters forming a character **string,** as in the pattern:

```
car
```

This pattern will match any data string containing the sequence of letters c-a-r. The letters must be contiguous, so no character (including spaces) can be interposed between any of the letters. The above pattern is present in strings such as "`carillon`" and "`ricercare`" but not in strings such

as "Caruso" or "clarinet". The above pattern is called a *literal* since the matching pattern must be literally identical to the regular expression (including the correct use of upper- or lower-case).

When a pattern is found, a starting point and an ending point are identified in the input string, corresponding to the defined regular expression. The specific sequence of characters found in the input string is referred to as the *matched string* or *matched pattern.*

Wild-Card

Regular expressions that are not literal involve so-called *metacharacters.* Metacharacters are used to specify various operations, and so are not interpreted as their literal selves. The simplest regular expression metacharacter is the period (.). The period matches *any single character* — including spaces, tabs, and other ASCII characters. For example, the pattern:

```
c.u
```

will match any input string containing three characters, the first of which is the lower-case 'c' and the third of which is the lower-case 'u'. The pattern is present in strings such as "counterpoint" and "acoustic" but not in "cuivre" or "Crumhorn". Any character can be interposed between the 'c' and the 'u' provided there is precisely one such character.

Escape Character

A problem with metacharacters such as the period is that sometimes the user wants to use them as literals. The special meaning of metacharacters can be "turned-off" by preceding the metacharacter with the backslash character (\). The backslash is said to be an *escape* character since it is used to release the metacharacter from its special function. For example, the regular expression

```
\.
```

will match the period character. The backslash itself may be escaped by preceding it by an additional backslash (i.e. \\).

Repetition Operators

Another metacharacter is the plus sign (+). The plus sign means "one or more consecutive instances of the previous expression." For example,

```
fo+
```

specifies any character string beginning with a lower-case 'f' followed by one or more consecutive instances of the small letter 'o'. This pattern is present in such strings as "food" and "folly," but not in "front" or "flood." The length of the matched string is variable. In the case of "food" the matched string consists of three characters, whereas in "folly" the matched string consists of just two characters.

The plus sign in our example modifies only the preceding letter 'o' — that is, the single letter 'o' is deemed to be the *previous expression* which is affected by the +. However, the affected expression need not consist of just a single character. In regular expressions, parentheses ( ) are metacharacters that can be used to bind several characters into a single unit or sub-expression. Consider, by way of example, the following regular expression:

```
(fo)+
```

The parentheses now bind the letters 'f' and 'o' into a single two-character expression, and it is this expression that is now modified by the plus sign. The regular expression may be read as "one or more consecutive instances of the string 'fo'." This pattern is present in strings like "food" (one instance) and "fofoe" (two instances).

Of course we can mix metacharacters together. The expression:

```
(.o)+
```

will match strings such as "polo" and the first four letters of "tomorrow."

Several sub-expressions may occur within a single regular expression. For example, the following regular expression means "one or more instances of the letter 'a', followed by one or more instances of the string 'go'."

```
(a)+(go)+
```

This would match character strings in inputs such as "ago" and "agogic," but not in "largo" (intervening 'r') or "gogo" (no leading 'a'). Note that the parentheses around the letter 'a' can be omitted without changing the sense of the expression. The following expression mixes the + repetition operator with the wild-card (.):

```
c+.m+
```

This pattern is present in strings such as "accompany," "accommodate," and "cymbal." This pattern will also match strings such as "ccm" since the second 'c' can be understood to match the period metacharacter.

A second repetition operator is the asterisk (*). The asterisk means "zero or more consecutive instances of the previous expression." For example,

```
Do*r
```

specifies any character string beginning with an upper-case 'D' followed by zero or more instances of the letter 'o' followed by the letter 'r'. This pattern is present in such strings as "Dorian," "Doors" as well as "Drum," and "Drone." As in the case of the plus sign, the asterisk modifies only the preceding expression — in this case the letter 'o'. Multi-character expressions may be modified by the asterisk repetition operator by placing the expression in parentheses. Thus, the regular expression:

```
ba(na)*
```

will match strings such as "ba," "bana," "banana," "bananana," etc.

Incidentally, notice that the asterisk metacharacter can be used to replace the plus sign (+) metacharacter. For example, the regular expression `X+` is the same as `XX*`. Similarly, `(abc)+` is equivalent to `(abc)(abc)*`.

A frequent construction used in regular expressions joins the wild-card (`.`) with the asterisk repetition character (`*`). The regular expression:

```
.*
```

means "zero or more instances of any characters." (Notice the plural "characters;" this means the repetition need not be of one specific character.) This expression will match *any string*, including nothing at all (the *null string*). By itself, this expression is not very useful. However it proves invaluable in combination with other expressions. For example, the expression:

```
{.*}
```

will match any string beginning with a left curly brace and ending with a right curly brace. If we replaced the curly braces by the space character, then the resulting regular expression would match any string of characters separated by spaces — such as printed words.

A third repetition operator is the question mark (`?`), which means "zero or one instance of the preceding expression." This metacharacter is frequently useful when you want to specify the presence or absence of a single expression. For example, the pattern:

```
Ch?o
```

is present in such strings as "`Chopin`" and "`Corelli`" but not "`Chinese`" or "`cornet`."

Once again, parentheses can be used to specify more complex expressions. The pattern:

```
Ch?(o)+
```

is present in such strings as "`Chorale`," "`Couperin`," and "`Cooper`," but not in "`Chloe`" or "`Chant`."

In summary, we've identified three metacharacters pertaining to the number of occurrences of some character or string. The plus sign means "one or more," the asterisk means "zero or more," and the question mark means "zero or one." Collectively, these metacharacters are known as *repetition operators* since they indicate the number of times an expression can occur in order to match.

Context Anchors

Often it is helpful to limit the number of occurrences matched by a given pattern. You may want to match patterns in a more restricted context. One way of restricting regular expression pattern-matches is by using so-called *anchors*. There are two regular expression anchors. The caret (`^`) anchors the expression to the beginning of the line. The dollar sign (`$`) anchors the expression to

the end of the line. For example,

```
^A
```

matches the upper-case letter 'A' only if it occurs at the beginning of a line. Conversely,

```
A$
```

will match the upper-case letter 'A' only if it is the last character in a line. Both anchors may be used together, hence the following regular expression matches only those lines containing just the letter 'A':

```
^A$
```

Of course anchors can be used in conjunction with the other regular expressions we have discussed. For example, the regular expression:

```
^a.*z$
```

matches any line that begins with 'a' and ends with 'z'.

OR Logical Operator

One of several possibilities may be matched by making use of the logical *OR* operator, represented by the vertical bar (|). For example, the following regular expression matches either the letter 'x' or the letter 'y' or the letter 'z':

```
x|y|z
```

Expressions may consist of multiple characters, as in the following expression which matches the string 'sharp' or 'flat' or 'natural'.

```
sharp|flat|natural
```

More complicated expressions may be created by using parentheses. For example, the regular expression:

```
(simple|compound) (duple|triple|quadruple|irregular) meter
```

will match eight different strings, including `simple triple meter` and `compound quadruple meter`.

Character Classes

In the case of single characters, a convenient way of identifying or listing a set of possibilities is to use the *character class*. For example, rather than writing the expression:

```
a|b|c|d|e|f|g
```

the expression may be simplified to:

```
[abcdefg]
```

Any character within the square brackets (a "character class") will match. Spaces, tabs, and other characters can be included within the class. When metacharacters like the period (`.`), the asterisk (`*`), the plus sign (`+`), and the dollar sign (`$`) appear within a character class, they lose their special meaning, and become simple literals. Thus the regular expression:

```
[xyz.+*$]
```

matches any one of the characters 'x,' 'y,' 'z,' the period, plus sign, asterisk, or the dollar sign.

Some other characters take on special meanings within character classes. One of these is the dash (-). The dash acts as a *range* operator. For example,

```
[A-Z]
```

represents the class of all upper-case letters from A to Z. Similarly,

```
[0-9]
```

represents the class of digits from zero to nine. The expression given earlier — `[abcdefg]` — can be simplified further to: `[a-g]`. Several ranges can be mixed within a single character class:

```
[a-gA-G0-9#]
```

This regular expression matches any one of the lower- or upper-case characters from A to G, or any digit, or the octothorpe (#). If the dash appears at the beginning or end of the character class, it loses its special meaning and becomes a literal dash, as in:

```
[a-gA-G0-9#-]
```

This regular expression adds the dash character to the list of possible matching characters.

Another useful metacharacter within character classes is the caret (^). When the caret appears at the beginning of a character-class list, it signifies a *complementary character class.* That is, only those characters *not* in the list are matched. For example,

```
[^0-9]
```

matches any character other than a digit. If the caret appears in any position other than at the beginning of the character class, it loses its special meaning and is treated as a literal. Note that if a character-class range is not specified in numerically ascending order or alphabetic order, the regular expression is considered ungrammatical and will result in an error.

Examples of Regular Expressions

The following table lists some examples of regular expressions and provides a summary description of the effect of each expression:

A	match letter 'A'
^A	match letter 'A' at the beginning of a line
A$	match letter 'A' at the end of a line
.	match any character (including space or tab)
A+	match one or more instances of letter 'A'
A?	match a single instance of 'A' or the null string
A*	match one or more instances of 'A' or the null string
.*	match any string, including the null string
A.*B	match any string starting with 'A' up to and including 'B'
A|B	match 'A' or 'B'
(A)|(B)	match 'A' or 'B'
[AB]	match 'A' or 'B'
[^AB]	match any character other than 'A' or 'B'
AB	match 'A' followed by 'B'
AB+	match 'A' followed by one or more 'B's
(AB)+	match one or more instances of 'AB', e.g. ABAB
(AB)|(BA)	match 'AB' or 'BA'
[^A]AA[^A]	match two 'A's preceded and followed by characters other than 'A's
^[^^]	match any character at the beginning of a record except the caret

Examples of regular expressions.

Examples of Regular Expressions in Humdrum

The following table provides some examples of regular expressions pertinent to Humdrum-format inputs:

^!!	match any global comment
^!!.*Beethoven	match any global comment containing 'Beethoven'
^!!.*[Rr]ecapitulation	match any global comment containing the word 'Recapitulation' or 'recapitulation'
^!($|[^!])	match only local comments
^**	match any exclusive interpretation
^*[^*]	match only tandem interpretations
^*[-+vx^]$	match spine-path indicators
^[^*!]	match only data records
^[^*!].*$	match entire data records
^(\.<*tab*>)*\.$	match records containing only null tokens (<*tab*> means a tab)
^*f#:	match key interpretation indicating F# minor

Regular expressions suitable for all Humdrum inputs.

By way of illustration, the next table shows examples of regular expressions appropriate for processing `**kern` representations.

^=	match any **kern barline or double barline
^=[^=]	match **kern single barlines but not double barlines
^[^=]	match any token other than a barline or double barline
;	match any **kern note or barline containing a pause
T	match any **kern note containing a whole-tone trill
[Tt]	match any **kern note containing a whole-tone or half-tone trill
–	match any **kern note containing at least one flat
[#]	match any **kern note containing a sharp, double-sharp, etc.
[#n–;]	match any **kern note containing an accidental, including a natural
[A-Ga-g]+	match any diatonic pitch letter-name
[0-9]+\.	match **kern dotted durations
[0-9]+\.\.[^.]	match only doubly-dotted durations
[Gg]+[^#-]	match any **kern pitch 'G' that does not have a sharp or flat
(^|[^g])gg($|[^g#-])	match only the pitch 'gg' (G5)
{.*r|r.*{	match all phrases that start with a rest
^4[^0-9.]|[^0-9]4([^0-9.]|$)	match **kern quarter durations
^(8|16)[^0-9.]|[^0-9](8|16)[^0-9.]	match eighth and sixteenth durations only
(([Ee]+-)|([Gg]+-)|([Bb]+-))($|[^-])	match any note from E-flat minor chord

Regular expressions suitable for **kern *data records.*

Note that the above regular expressions assume that comments and interpretations are not processed in the input. The processing of just data records can be assured by embedding each of the regular expressions given above in the expression

(^[^*!].**regexp*)|(^*regexp*)

For example, the following regular expression can be used to match **kern trills without possibly mistaking comments or interpretations:

```
(^[^*!].*[Tt])|(^[Tt])
```

For Humdrum commands such as **humsed**, **rend**, **yank**, **xdelta**, and **ydelta**, regular expressions are applied only to data records so there is no need to use the more complex expressions. In many circumstances, we will see that it is convenient to use the Humdrum **rid** command to explicitly remove comments and interpretations prior to processing (see Chapter 13).

Basic, Extended, and Humdrum-Extended Regular Expressions

Over the years, new features have been added to regular expression syntax. Some of the early software tools that make use of regular expressions do not support the extended features provided by more recently developed tools. So-called "basic" regular expressions include the following features: the single-character wild-card (.), the repetition operators (*) and (?) but not (+), the context anchors (^) and ($), character classes ([...]), or complementary character classes ([^...]). Parenthesis grouping is supported in basic regular expressions, but the parentheses

must be used in conjunction with the backslash to *enable* this function (i.e. \ (\ )). In Chapter 3 we introduced the **grep** command; **grep** supports only basic regular expressions.

"Extended" regular expressions include the following: the single-character wild-card (.), the repetition operators (*), (?) and (+), the context anchors (^) and ($), character classes ([...]), complementary character classes ([^...]), the logical OR (|), and parenthesis grouping. Extended regular expressions are supported by the **egrep** command; **egrep** operates in the same manner as **grep**, only the search patterns are interpreted according to extended regular expression syntax.

The Humdrum **pattern** command further extends regular expression syntax by providing multi-record repetition operators that prove very useful in musical applications. These Humdrum extensions will be discussed in Chapter 21.

Reprise

Regular expressions provide a powerful method for defining abstract patterns of alphanumeric characters. The wild card (.) matches any character. Repetition operators include "one or more" (+), "zero or more" (*), and "zero or one" (?). Context anchors define the beginning of the line (^) or the end of the line ($). Character classes ([ ]) specify a choice of possible characters. Ranges can be specified within character classes ([-]) and complementary classes may be defined ([^]). The logical OR (|) may be used in conjunction with parentheses to define more complex expressions.

There are many software tools that make use of regular expressions. The UNIX **grep** command supports standard or "basic" regular expressions. The UNIX **egrep** command supports "extended" regular expressions.

In the next chapter we will explore how regular expressions may be used in musical applications.

Chapter 10

Musical Uses of Regular Expressions

Now that you have a better understanding of regular expressions, let's apply them. This chapter provides many examples of how regular expressions may be used to define musically useful patterns. In subsequent chapters, we'll make frequent use of regular expressions.

The *grep* Command (Again)

Although regular expressions are used in a number of Humdrum commands, they are most frequently used in conjunction with the **grep** command encountered in Chapter 3. **grep** is a popular software tool that is available from a number of manufacturers and sources. Many versions of **grep** differ in the options provided. For example, the version of **grep** distributed by the GNU Software Foundation provides no fewer than 19 options. Some of the most common options for **grep** are identified in Table 10.1.

Table 10.1

-c	count the number of lines matching the regular expression
-f *file*	search for patterns that are specified in *file*
-i	ignore differences of upper- and lower-case
-l	just list the names of files containing a matching line
-n	prefix each output line with its line number
-h	suppress file-name prefixes (headers) in output when searching more than one file
-v	display all lines *not* matching the regular expression
-L	list names of files *not* containing the regular expression

*Common options for the **grep** command.*

Many of the predefined Humdrum representations make use of the "common system" for representing barlines. The following command counts the number of barlines in the file `czech37.krn`. Note that the caret anchor (`^`) is used to avoid inadvertent matches of the equals sign that might appear in Humdrum comments or interpretations.

```
grep -c ^= czech37.krn
```

Recall that the dollar sign (`$`) can be used to anchor an expression to the end of the line. The following command determines whether numbered measure 9 is present in the file `france12.krn`; the dollar sign ensures that measure 9 is not mistaken for measure 90, 930, etc.

```
grep ^=9$ france12.krn
```

The asterisk means "zero or more" instances of the preceding expression. For example, the following regular expression will match any reference record or global comment in the file `clara29`:

```
grep '^!!!*' clara29
```

Suppose we want to list all of the global comments for all files in the current directory:

```
grep '^!!!*' *
```

Notice that the two asterisks serve different functions in the above command. The first asterisk means "zero or more instances" and is part of the regular expression passed to **grep**. The second asterisk means "all files in the current directory" and is expanded by the shell. The first asterisk is 'protected' from the shell by the single quotes. Otherwise, the first asterisk might be expanded by the shell to a list of all files in the current directory.

In regular expressions, the period character (`.`) matches any single character. For example, the expression '`A.B`' will match strings such as '`AXB`' and '`AAB`' etc. The following command identifies all eighth-notes containing at least one flat, and whose pitch lies within an octave of middle C.

```
grep 8.- *.krn
```

Frequently it is necessary to turn off the special meanings for metacharacters such as ^, $, and *. Recall that this can be done by inserting a backslash (`\`) immediately prior to the metacharacter. In the `**kern` representation the caret signifies an accent. In a monophonic input, we might count the number of notes that have a notated accent as follows:

```
grep -c '\^' danmark3.krn
```

In the following command we have used the backslash to escape the special meaning of the asterisk. The **-l** option causes **grep** to output only the names of any files that contain a line matching the pattern. Hence, the following command identifies those files in the current directory that encode music in 9/8 meter:

```
grep -l '^\*M9/8' *
```

Recall that square brackets can be used to indicate character classes where any of the characters in the class can be used to match the expression. The following command identifies those files in the current directory that encode music in either 3/8 or 9/8 meter:

```
grep -l '\*M[39]/8' *
```

One of the most frequently used regular expressions consists of the period followed by the asterisk (`.*`). Recall that this expression will match *any* string including the null string (i.e. nothing at all). This expression commonly appears between two other character strings. For example, we can identify all files in the current directory whose instrumentation includes a trumpet:

```
grep -l '!!!AIN.*tromp' *
```

The `.*` expression is needed since we don't know what other instruments might be listed following `AIN` and before `tromp`. Instrumentation reference records require that instrument codes appear in alphabetical order. This makes it easier to conduct searches for combinations of instruments. For example, we can identify all scores in the current directory whose instrumentation includes both trumpet and cornet as follows:

```
grep -l '!!!AIN.*cornt.*tromp' *
```

There are many variants on the use of the `.*` expression. The following command identifies all files that contain a record having the word `Drei` followed by the word "`Koenige`". (Notice the use of the **-i** option in order to ignore the case of the letters.)

```
grep -li 'Drei.*Koenige' *
```

This command will match such strings as: *Die Heiligen Drei Koenige*, *Drei Koenige, Dreikoenigslied,* etc.

The '`!!!AGN`' reference record is used to encode genre-related keywords. The following command lists all files that are ballads.

```
grep -l '!!!AGN.*Ballad' *
```

List all files that have the word `Amour` in the title:

```
grep -li '!!!OLT.*Amour' *
```

List any works that bear a dedication:

```
grep -l '!!!ODE:' *
```

List those works that are in irregular meters:

```
grep -l '!!!AMT.*irregular' *
```

The **-L** option for **grep** causes the output to contain a list of files *not* containing the regular expression. For example, we could identify those works that don't bear any dedication:

```
grep -L '!!!ODE:' *
```

List those works *not* composed by Schumann:

```
grep -L '!!!COM: Schumann' *
```

Identify any works that don't contain any double barlines:

```
grep -L '^==' *
```

How many works in the current directory are in simple-triple meter?

```
grep -c '!!!AMT.*simple.*triple' *
```

When searching for more complex patterns it may be necessary to use **grep** more than once. Consider, for example, the problem of identifying works whose titles contain both the words `Liebe` and `Tod`. The first of the following commands will identify only those titles that contain `Liebe` followed by `Tod`, whereas the second command will identify only those titles that contain `Tod` followed by `Liebe`:

```
grep '!!!OTL.*Liebe.*Tod' *
grep '!!!OTL.*Tod.*Liebe' *
```

A better solution is to pipe the output between two **grep** commands. Recall that the vertical bar ('|') conveyes or "pipes" the output from one command to the input of a subsequent command. The following command passes all opus-title records (`OTL`) containing the word `Liebe` to a second **grep**, which passes only those records also containing the word `Tod`. Since both **grep** commands process the entire input line, it does not matter whether the word `Tod` precedes or follows the word `Liebe`:

```
grep '!!!OTL.*Liebe' * | grep 'Tod'
```

The **-v** option for **grep** causes a "reverse" or "negative" output. Instead of outputting all records that *match* the specified regular expression, the **-v** option causes only those records to be output that do *not* match the given regular expression. For example, the following command eliminates all comments from the file `polska24.krn`:

```
grep -v '^!' polska24.krn
```

Similarly, the following command eliminates all whole-note rests:

```
grep -v 1r *
```

The **-v** option is especially convenient in pipelines. For example, the following command identifies all those files whose instrumentation includes a cornet but not a trumpet:

```
grep '!!!AIN.*cornt' * | grep -v 'tromp'
```

The following command identifies those works in compound meters that are not also quadruple meters:

```
grep '!!!AMT.*compound' * | grep -v 'quadruple'
```

Similarly, the following command identifies those notes that begin a phrase, but are not rests.

```
grep '^{' * | grep -v r
```

German, French, Italian, and Neapolitan Sixths

In conjunction with the **solfa** command, **grep** can be used to search for various types of special chords. Suppose, for example, that we wanted to identify occurrences of augmented sixth chords. An augmented sixth chord is characterized by an augmented sixth interval occurring between the lowered sixth scale-degree and the raised fourth scale-degree. In Chapter 4, we saw that the **solfa** command represents pitches with respect to an encoded tonic pitch. In the `**solfa` representation, the lowered sixth and raised fourth degrees will be represented as `6-` and `4+` respectively. First we translate the input to the `**solfa` representation, and then we search for records matching the appropriate regular expression:

```
solfa input | grep '6-.*4+'
```

Notice that the expression '`6-.*4+`' presumes that the lowered sixth degree is lower in pitch than the raised fourth degree. For augmented sixth chords, this is a reasonable presumption. In the unlikely situation that the raised fourth degree is lower in pitch than the lowered sixth degree, we would need to also search for the expression '`4+.*6-`'. Alternatively, we could use two separate **grep** commands, eliminating the constraint of order:

```
solfa input | grep '6-' | grep '4+'
```

Augmented sixth chords can be further classified as either German, French, or Italian sixths. The German sixth contains the lowered mediant whereas the French sixth contains the supertonic pitch; the Italian sixth contains neither:

```
solfa input | grep '6-.*4+' | grep '3-'          # German sixth
solfa input | grep '6-.*4+' | grep '2'           # French sixth
solfa input | grep '6-.*4+' | grep -v '[23]'     # Italian sixth
```

A similar approach can be used to identify Neapolitan sixth chords. These chords are based on the lowered supertonic with the third of the chord (unaltered subdominant) in the bass.

```
solfa input | grep '4[^-+].*2-' | grep '6-'      # Neapolitan sixth
```

Depending on the key, Neopolitan chords are sometimes notated enharmonically as a raised tonic chord. Suppose we were looking for such enharmonically spelled Neopolitan chords:

```
solfa input | grep '3+.*1+' | grep '5+'
```

Occassionally, Neapolitan chords are missing the fifth of the chord (the lowered sixth degree of the scale). We might search for an example of such a chord:

```
solfa input | grep '2-' | grep '4' | grep -v '6-'
```

AND-Searches Using the *xargs* Command

In some cases, we want to identify those files that match two entirely different patterns (in different records). Recall that the **-l** option causes **grep** to output the *filename* rather than the matching

record. If we could pass along these file names to another **grep** command, we could search those same files for yet another pattern.

The UNIX **xargs** command provides a useful way of transferring the output from one command to be used as final arguments for a subsequent command. For example, the following command takes each file whose opus title contains the word `Liebe` and counts the number of phrases.

```
grep -l '!!!OTL:.*Liebe' * | xargs grep -c '^{'
```

In this case the **grep -l** command outputs a list of names of files containing the string `Liebe` in an OTL reference record. The **xargs** command causes these filenames to be appended to the end of the following **grep** command. The **grep -c** command will thus be applied only to those files already identified by the previous **grep** as containing `Liebe` in the title.

A set of such pipelines can be used to answer more sophisticated questions. For example, are drinking songs more apt to be in triple meter?

```
grep -l '!!!AMT.*triple'  *   | xargs grep -l '!!!AGN.*Trinklied'
grep -l '!!!AMT.*duple'   *   | xargs grep -l '!!!AGN.*Trinklied'
grep -l '!!!AMT.*quadruple' * | xargs grep -l '!!!AGN.*Trinklied'
```

Similarly, the following commands determine whether files whose titles contain the word *death* are more apt to be in minor keys:

```
grep -li '!!!OTL.*death' * | xargs grep -c '^\*[a-g][#-]*:'
grep -li '!!!OTL.*death' * | xargs grep -c '^\*[A-G][#-]*:'
```

Note that the **xargs** command can be used again and again to continue propagating file names as arguments to subsequent searches. For example, the following command outputs the key signatures for all works originating from Africa that are written in 3/4 meter:

```
grep -l '!!!ARE.*Africa' * | xargs grep -l '^\*M3/4' \
    | xargs grep '^\*k\['
```

Similarly, the following command outputs the names of all files in the current directory that encode 17th century organ works containing passages in 6/8 meter:

```
grep -l '!!!ODT.*16[0-9][0-9]/' | xargs grep -l \
     '!!!AIN.*organ' | xargs grep -l '\*M6/8'
```

Using the **-L** option allows us to form even more complex criteria by excluding certain works. For example, the following variation of the above command outputs the names of all files in the current directory that encode 17th century organ works that do not contain passages in 6/8 meter:

```
grep -l '!!!ODT.*16[0-9][0-9]/' | xargs grep -l \
     '!!!AIN.*organ' | xargs grep -L '\*M6/8'
```

OR-Searches Using the *grep -f* Command

In effect, the above pipelines provide logical **AND** structures: e.g. identify works composed in the 17th century AND written for organ AND containing a passage in 6/8 meter. The **-f** option for **grep** provides a way of creating logical **OR** searches. With the **-f** option, we specify a file containing the patterns being sought. For example, we might create a file called `criteria` containing the following three regular expressions:

```
!!!ODT.*16[0-9][0-9]/
!!!AIN.*organ
\*M6/8
```

We would invoke **grep** as follows:

```
grep -l -f criteria *
```

The **-f** option tells **grep** to fetch the file `criteria` and use the records in this file as regular expressions. A match is made if any of the regular expressions is found.

The output would consist of a list of all files in the current directory that encode works composed in the 17th century OR written for organ OR in 6/8 meter. The **-f** option is more typically used to specify several variations of the same idea. For example, suppose we were searching for D major triads in `**pitch` data. We could use a file containing the following regular expressions:

```
[Dd].*[Ff]#.*[Aa]
[Dd].*[Aa].* [Ff]#
[Ff]#.*[Aa].*[Dd]
[Ff]#.*[Dd].*[Aa]
[Aa].*[Dd].*[Ff]#
[Aa].*[Ff]#.*[Dd]
```

Depending on the application, it may be easier to construct such pattern files than to use a lengthy pipeline. That is:

```
grep -f Dmajor *
```

may be less cumbersome than:

```
grep [Dd] * | grep [Ff]# | grep [Aa]
```

The **-f** option can be combined with **-L**. For example, suppose we wanted to identify all works in the current directory that are not in the keys of C major, G major, B-flat major or D minor. Our regular expression file would contain the following regular expressions:

```
^\*[CGd]:
^\*B-:
```

The corresponding command would be:

```
grep -L -f criteria *
```

Another way of thinking of the **-f** option is that it allows us to define equivalences. Consider, for example, the task of counting all of the notes in a `**kern` melody that belong to a particular whole-tone pitch set. Let's create two files, one called `whole1` and the other called `whole2`. The file `whole1` might contain the following regular expressions:

```
[Cc]([^-#Cc]|$)
[Dd]([^-#Dd]|$)
[Ee]([^-#Ee]|$)
[Ff]#([^#]|$)
[Gg]-([^-]|$)
[Gg]#([^#]|$)
[Aa]-([^-]|$)
[Aa]#([^#]|$)
[Bb]-([^-]|$)
```

Notice that the regular expressions have been carefully defined. The first regular expression defines a pattern consisting of either an upper- or lower-case letter 'C' followed either by a character that is neither a sharp (#) nor a flat (-) nor another letter 'C', nor is followed by the end of the line ($).

Recall that parenthesis grouping (...) is part of the *extended* regular expression syntax. Therefore, we should use the **egrep** rather than the **grep** command with the above expressions. We can count the number of notes in a monophonic `**kern` input that belong to this whole-tone set:

```
egrep -c -f whole1 debussy
```

If the file `whole2` contains regular expressions for the complementary pitch set, we could similarly count the number of pitches that belong to this alternative set:

```
egrep -c -f whole2 debussy
```

Reprise

The **grep** command is usually thought of as a way to find particular patterns in a file or input stream. However, the various options for **grep** (such as -v, -l, and -L) allow **grep** to be used for other purposes. It can be used to isolate data, to count occurrences of patterns, to eliminate unwanted lines, to identify files for processing, and to avoid files that contain certain information.

We have seen how the **xargs** command can be used to carry out **AND**-searches where each work must conform to multiple criteria. We have also seen how the **-f** option for **grep** can be used to permit **OR**-searches where a work needs to conform only to one of a set of possible criteria.

Although this chapter has focussed principally on the **grep** command, the ensuing chapters will show that regular expressions are used by a wide variety of commands. In Chapter 33, many more powerful examples will be discussed in conjunction with the **find** command.

Chapter 11

Melodic Intervals

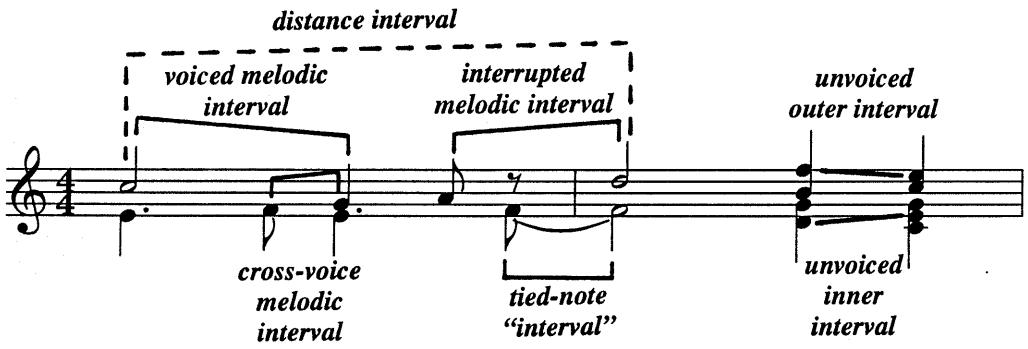
A musical interval is the distance between two pitches. When the pitches are consecutive the distance is referred to as a *melodic interval;* when the pitches are concurrent the distance is referred to as a *harmonic interval.*

The simplicity of these definitions is deceptive. In real music, the determination of pitch intervals can be surprisingly complicated. In this chapter we will discuss Humdrum tools related to melodic pitch intervals — specifically the **mint** (melodic interval) and **xdelta** commands. Discussion of harmonic intervals will be delayed until Chapter 15.

Types of Melodic Intervals

Example 11.1 provides a contrived illustration of seven different types of melodic intervals. (The corresponding `**kern` representation is given on the following page.) The simplest melodic interval is calculated between successive pitches within the same voice or part. We might call this a *voiced melodic interval.* Examples of voiced melodic intervals are the descending perfect fourth between the first two notes of the upper part, and the rising semitone at the beginning of the lower part.

Example 11.1 Types of melodic intervals.



Even within a monophonic score, successive pitches may have one or more rests interposed between them. Depending on the research task, the interval spanning across the rest(s) may or may not be considered important. We might call such intervals *interrupted melodic intervals.* An ex-

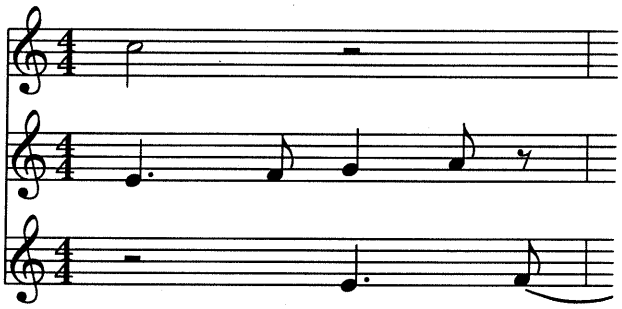
ample of an interrupted melodic interval is the ascending perfect fourth in the upper voice between the last note of the first measure (A4) and the first note of the last measure (D5).

Example 11.1 continued.

```
**kern      **kern
*staff1     *staff1
*M4/4       *M4/4
=1-         =1-
4.e         2cc
8f          .
4.e         4g
.           8a
[8f         8r
=2          =2
2f]         2dd
4d 4g       4b 4ff
4c 4e 4g    4cc 4ee
*-          *-
```

In some cases, implied melodic intervals may arise by the interaction of two or more parts. For example, Example 11.2 shows a plausible reinterpretation of the voicings for the first measure. Here the quarter-note G is presumed to continue from the preceding eighth-note F rather than from the half-note C. In the context of the original `**kern` encoding above, we might call such intervals *cross-voice melodic intervals.*

Example 11.2 Possible re-interpretation of opening measure for Example 11.1.



In the second measure, both parts are encoded using multiple stops. In the upper part, two successive double stops are encoded. In the case of multiple-stops, it is common to perceive the outermost notes as connected. Hence, the B would resolve to the C and the F would resolve to the E. We might call such intervals *unvoiced outer intervals.* We may call them *unvoiced* because they aren't encoded using separate spines. The voicings are implied, principally because the notes form an 'upper' or 'lower' part.

The multiple-stops in the lower voice illustrate even more possibilities. Apart from the unvoiced outer intervals (D→C and G→G), there are other possible melodic intervals. These include D→E and G→E. We might refer to such intervals as *unvoiced inner intervals.*

In many research tasks (such as identifying melodic variations), important interval relationships

may stretch across several intervening notes. In Example 11.1, for example, the two half-notes in the upper voice might be viewed as forming an ascending major second interval (i.e. C→D). We might call such intervals *distance intervals.*

Finally, although in most situations tied notes should be treated as a single note, in some circumstances there is merit to considering each notehead as independent.

By way of summary, we have distinguished no less than seven different types of melodic intervals: voiced melodic intervals, interrupted melodic intervals, cross-voice melodic intervals, unvoiced outer intervals, unvoiced inner intervals, distance intervals, and tied note intervals.

Humdrum commands related to melodic intervals provide users with several alternative ways of interpreting melodic intervals. Users are typically interested in only certain types of intervals and so it is useful to restrict the outputs to specified interval classes.

Apart from the question of types of melodic intervals, melodic intervals can be calculated according to a variety of *units*. Depending on the circumstance, the user may wish to calculate diatonic intervals, semitones, cents, frequency differences, or even differences in cochlear coordinates.

Melodic Intervals Using the *mint* Command

The Humdrum **mint** command calculates melodic intervals for pitch-related representations such as `**kern`, `**pitch`, `**solfg` and `**Tonh`. Output intervals are expressed using the traditional diatonic terms where both interval quality and interval size are specified. Interval qualities include perfect (`P`), major (`M`), minor (`m`), augmented (`A`) and diminished (`d`). Interval qualities may also be doubly augmented (`AA`), triply diminished (`ddd`) and so on. Diatonic interval sizes are indicated by numbers (1=unison, 2=second, ... 8=octave, 9=ninth, etc.). Ascending and descending intervals are distinguished by a leading plus sign (+) or minus sign (−) respectively.

In the default operation, **mint** outputs two of the seven types of melodic intervals. These are *voiced melodic intervals,* and *interrupted melodic intervals;* By way of illustration, Example 11.3 shows the output from the **mint** command for the input shown in Example 11.1.

Example 11.3 Default interval outputs from the **mint** command corresponding to Example 11.1.

```
**mint          **mint
*M4/4           *M4/4
=1-             =1-
.               .
+m2             .
-m2             -P4
.               +M2
+m2             r
=2              =2
.               +P4
-m3  +M2        -m3  +m3
-M2  P1         +m2  -m2
*-              *-
```

Notice that the *interrupted interval* (spanning the rest) has been calculated, and that no unison has appeared for the tied note in the lower voice.

If desired, the unison intervals between successive tied notes can be output via the **-t** option for **mint.**

Sometimes it is useful to maintain the initial starting pitches in the output. The presence of these "offset" pitch values can prove useful in later reconstructing the original pitches from the `**mint` interval data. When the **-o** option is invoked, **mint** outputs the initial starting pitches (placed in square brackets) from which the subsequent melodic intervals have been calculated.

In order to avoid outputting interrupted intervals, the **-b** (break) option can be used. This option requires a subsequent regular expression that defines the contexts where the interval calculation should be suspended and restarted. A common invocation would identify `**kern` rests (`r`) as a suitable place to break melodic interval calculations. For example,

```
mint -b r inputfile
```

would produce the following output when applied to Example 11.1:

```
**mint          **mint
*M4/4           *M4/4
=1-             =1-
.               .
+m2             .
-m2             -P4
.               +M2
+m2             r
=2              =2
P1              .
-m3 +M2         -m3 +m3
-M2 P1          +m2 -m2
*-              *-
```

Notice that the perfect fourth (+P4) has been replaced by a null token at the beginning of measure 2. In addition, the rest token '`r`' has been echoed just prior to the barline.

Depending on the regular expression given, the **-b** option can used for a variety of specialized intervals. For example, suppose that we wanted to avoid calculating intervals between the last note of a phrase and the first note of the next phrase. In the `**kern` representation, the open and closed curly braces are used to indicate the beginnings and ends of phrases. We need to tell **mint** to break interval calculations each time an end-of-phrase signifier is encountered:

```
mint -b '}' inputfile
```

Similarly, the `**kern` representation uses the semicolon (`;`) to represent pauses. We might instruct **mint** to avoid calculating intervals between notes having pauses and the subsequent note:

```
mint -b ';' inputfile
```

Since the **-b** option accepts regular expressions, we can combine patterns. For example, the following command instructs **mint** to calculate melodic intervals, not including intervals spanning phrase boundaries, and not following notes with pauses:

```
mint -b '[;}]' inputfile
```

Unvoiced Inner Intervals

Unvoiced inner intervals can be included in the output by using the **-i** or **-I** options. With the **-I** option, unvoiced inner intervals appear in the output in parentheses. For example, the following output is generated for Example 1.1 with the **-I** option. Notice the addition of `(+M2)` and `(-2)`. The rising major second arises from the pitches D4 and E4; the falling minor third arises from the pitches G4 and E4.

```
**mint                    **mint
*M4/4                     *M4/4
=1-                       =1-
.                         .
+m2                       .
-m2                       -P4
.                         +M2
+m2                       r
=2                        =2
P1                        .
-m3 +M2                   -m3 +m3
-M2 (+M2) (-m3) P1        +m2 -m2
*-                        *-
```

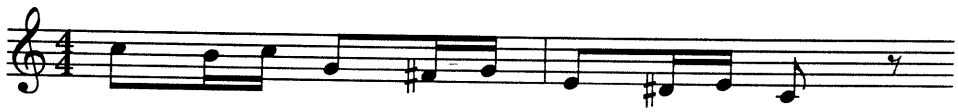
With the **-i** option, the parentheses surrounding the unvoiced inner intervals would be omitted.

Calculating Distance Intervals Using the mint -s Command

Another option provided by **mint** is the **-s** or skip option. Like the **-b** option, this option requires a subsequent regular expression. Any token matching this expression is transformed to a null data token and is ignored when processing. One possible use for this option is to help calculate *distance intervals.* Consider Example 11.4 where all of the durations are either sixteenth notes or eighth notes. Suppose we wanted to calculate the intervals only between the eighth notes.

Example 11.4

```
**kern
*M4/4
=1-
8cc
16b
16cc
8g
16f#
16g
=2
8e
16d#
16e
8c
8r
*-
```

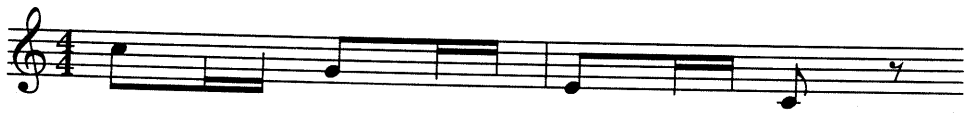


We can use the skip option to instruct **mint** to ignore any note token matching the string '16':

```
mint -s 16 inputfile
```

This command would produce an output that highlights the descending arpeggiated major chord — from C5 to G4 (down a P4), to E4 (down a m3) to C4 (down a M3).

```
**mint
*M4/4
=1-
.
.
.
-P4
.
.
=2
-m3
.
.
-M3
r
*-
```



Using duration information is a somewhat limited technique for calculating distance intervals. Typically, users will want to define much more refined ways of identifying structural tones. More sophisticated methods for calculating distance intervals are discussed in Chapter 31 on "Layers."

Cross-voice melodic intervals can be calculated by amalgamating several spines into a single spine. In Chapter 26 we will learn more about the **cleave** command. But here is a typical use:

```
cleave -d ' ' -i '**kern' -o '**kern' example11a
```

With the `**kern` encoding for Example 11.1 as input, the corresponding output would be:

```
**kern
*M4/4
=1- =1-
4.e 2cc
8f
4.e 4g
8a
[8f 8r
=2 =2
2f] 2dd
4d 4g 4b 4ff
4c 4e 4g 4cc 4ee
*-
```

Note that this output doesn't quite conform to the `**kern` syntax: the barlines have been duplicated as double-stops, and the durations aren't right for multiple-stops. We can clean up the output using **humsed,** but the incoherent durations won't cause problems if our intention is to calculate pitch intervals.

If we pipe the above output through the **mint** command, the appropriate command pipeline becomes:

```
cleave -d ' ' -i '**kern' -o '**kern' example11a \
    | humsed 's/ =.*//' | mint -I
```

The corresponding melodic interval output is:

```
**mint
*M4/4
=1-
.
+m2 -P5
-m2 +M2
+P4 +M2
-M3 r
=2
+P4
-m3 (+M2) (-P5) (+A4) (-m3) +m3
-M2 (-m3) (-P5) (P1) (-M3) (+P4) (+m2) -m2
*-
```

Simple and Compound Melodic Intervals

Of course, some melodic variants alter the octave placement of pitches. The **mint -c** option outputs compound intervals (i.e. intervals of an octave or greater) as non-compound equivalents. For example, the interval of a major tenth (`M10`) will be output as a major third (`M3`).

Diatonic Intervals, Absolute Intervals and Contour

The **mint** command provides three further options of interest. The **-d** option causes **mint** to output only the diatonic interval size without the interval quality information. The **-a** option causes **mint** to output absolute pitch intervals without distinguishing ascending intervals from descending intervals. That is, the leading plus (+) and minus (–) signs are discarded.

Finally, the **-A** option causes **mint** to output just one of three states: a plus sign (+) indicating a rising interval, the minus sign (–) indicating a falling interval, and the number zero (0) indicating no pitch movement (i.e., unison). In short, the **-A** option outputs only gross contour. The **-a** and **-A** options are complementary.

Using the *mint* Command

Consider some of the following uses of the **mint** command.

Are there any major or minor ninth melodic intervals in the file `Sinatra`?

```
mint Sinatra | grep '[Mm][9]'
```

Are there any compound melodic intervals in the file `Piaf`?

```
mint Piaf | egrep '([Mm][9])|([MmPAd][1-9][0-9]''
```

Are descending seconds more common than ascending seconds in melodies by Maurice Chevalier?

```
mint Chevalier* | grep -c '+[Mm]2'
mint Chevalier* | grep -c '-[Mm]2'
```

An alternative way of achieving the same goal might simplify the regular expression to **grep** and use the **-d** (diatonic) option for **mint**:

```
mint -d Chevalier* | grep -c '+2'
mint -d Chevalier* | grep -c '-2'
```

Identify whether there are any tritone melodic intervals in any of the vocal parts of a score:

```
extract -i '*Ivox' Platters | mint -c | egrep '(A4)|(d5)'
```

Here we have used the extended regular expression capabilities of **egrep** to specify an either/or pattern.

Suppose we had a directory containing only files encoding melodies using the `**mint` representation. Does any melody in the current directory contain both an ascending major sixth and a descending major sixth?

```
grep -l '+M6' * | xargs grep '-M6'
```

Do the vocal lines of Louis Jourdain contain successive ascending major thirds (such as forming an augmented triad)?

```
mint Jourdain | grep -v = | uniq -d | grep '+M3'
```

What is the longest run of rising intervals in the vocal lines of Marlene Dietrich?

```
mint -A Dietrich | grep -v = | uniq -cd | grep '+' | sort -n
```

Calculating Melodic Intervals Using the *xdelta* Command

Often it is useful to calculate melodic intervals in purely numerical values, such as the number semitones or the number of cents. The **xdelta** command provides a general tool for calculating numerical differences between successive values within individual spines. In order to use **xdelta** to calculate semitone differences, we first need to transform our representation to `**semits` (discussed in Chapter 4). Recall that in the `**semits` representation, middle C is designated by the value zero, and all other pitches are represented by their (positive or negative) semitone distance. A C-major scale would appear as follows:

```
**semits
0
2
4
5
7
9
11
12
*-
```

We can transform this representation to semitone intervals as follows:

```
xdelta inputfile
```

For the above scale, the output would be:

```
**Xsemits
.
2
2
1
2
2
2
1
*-
```

Notice that the initial numerical value has been replaced by a null token, and all subsequent values

represent the numerical *difference* between successive values. If the scale had been descending in pitch, then the difference values would be negative.

Notice also that the input interpretation (`**semits`) has been modified to `**Xsemits`). The input representation for **xdelta** does not matter. The output is always modified so the letter X is prepended to the representation. This means that **xdelta** could as easily be used to calculate differences in cents (`**Xcents`), frequency (`**Xfreq`) or any other representation that contains numbers.

When **xdelta** encounters multiple-stops, it behaves in a manner similar to the **mint** command by calculating the numerical equivalent of unvoiced inner intervals. Consider the following example:

```
**semits
3
2 5
-1 7 14
12
*-
```

The **xdelta** command produces the following output:

```
**Xsemits
.
-1 2
-3 (5) (2) 9
13 5 -2
*-
```

Once again, the interpretation has been modified to `**Xsemits` and the leading value has been changed to a null token. In going from the value 3 to the values 2 and 5, the output differences are -1 and +2 respectively. In going from the values 2 and 5 to the values -1, 7 and 14, we see the outer differences (-1 – 2 = -3) and (14 – 5 = 9). The inner differences are output in parentheses: (7 – 2 = 5) and (7 – 5 = 2).

Like the **mint** command, **xdelta** provides a **-b** option to break calculations of numerical differences and a **-s** option to skip or completely ignore certain data tokens when processing. An important use of the **-s** option is to ignore barlines. Consider the following example:

```
**cents
100
400
800
=2
600
*-
```

The proper way to calculate differences in cents is to ensure that measure numbers are ignored:

```
xdelta -s ^= inputfile
```

Failure to skip the barline will cause a difference to be calculated between 800 and =2 (i.e., -798) and between =2 and 600 (i.e., 598).

Outputs from **xdelta** can be processed again using **xdelta** in order to calculate the differences of the differences. For example, we can calculate the second derivative of successive cents by using a pipeline containing two **xdelta** commands:

```
xdelta -s ^= inputfile | xdelta -s ^=
```

Reprise

Intervals come in a mind-boggling range of types and sizes. Interval sizes can be measured in a variety of ways. They can be characterized as diatonic qualities such as minor sevenths or augmented sixths. They can be measured in terms of semitone distance — or even in cents or hertz (frequency difference). Only the diatonic size may be of interest (e.g., "a fifth"), and compound intervals (e.g., major tenth) can be expressed by their non-compound equivalents (major third).

Melodic intervals can be described as ascending or descending, or as absolute distances without regard for direction. Types of melodic intervals can be distinguished according to how successive notes are voiced, and according to spans across rests or across less important pitches. We defined seven different types of melodic intervals including *voiced melodic intervals*, *interrupted melodic intervals*, *cross-voice melodic intervals*, *unvoiced outer intervals*, *unvoiced inner intervals*, *distance intervals*, and *tied note intervals*.

In this chapter we have seen how to use the **mint** command to calculate these various kinds of intervals. Specifically, we have illustrated how to calculate voiced intervals, interrupted intervals, unvoiced outer intervals and unvoiced inner intervals. In Chapter 26 we will show how to calculate cross-voice intervals, and in Chapter 31 ("Layers") we will consider how to calculate distance intervals.

We have also seen how **xdelta** can be used to measure purely numerical distances between successive values. As we will see, **xdelta** will prove useful in many other applications apart from calculating pitch distances.

Chapter 12

Selecting Musical Parts and Passages

A Humdrum file may contain an encoding of a full score, or even of a large collection of scores. Often, we would like to isolate or extract particular parts or segments from a file or input stream. For example, we might want to extract a particular instrumental part, or select a group of related instruments; we might want to isolate a particular passage, remove certain measures, or extract a specific phrase; we might want to pull out a labelled section of a score (such as a *Coda* or *Da Capo*), or we might want to select a particular verse in a strophic song. In addition, we might want to isolate specific types of information, such as the figured bass, melodic interval information, or vocal text.

In this chapter we will explore two Humdrum tools for extracting material: **extract** and **yank**.

We know that Humdrum representations are structured like a grid with horizontal data ("records") representing concurrent information, and vertical data ("spines") representing sequentially occuring information. The Humdrum **extract** command can be used to isolate columns or spines of information. The **yank** command can be used to isolate rows or records. The **extract** command can be used to extract musical parts or other types of information that might be represented in individual spines. The **yank** command can be used to isolate passages or segments from an input, such as specified measures, phrases, or sections.

Extracting Spines: The *extract* Command

The **extract** command allows the user to select one or more spines from a Humdrum input. The command is typically used to extract parts (such as a tuba part) from some multi-part score. However, **extract** can also be used to isolate dynamic markings, musical lyrics, or any other stream of information that has been encoded as a separate Humdrum spine.

The **extract** command has several modes of operation. With the **-f** option, the user may specify a given data column (spine) or "field" to extract. Consider the opening of Bach's second Brandenburg Concerto shown in Example 12.1.

Example 12.1. J.S. Bach *Brandenburg Concerto No. 2*, mov. 1.

```
!!!COM: Bach, Johann Sebastian
!!!OPR: Six Concerts Avec plusieurs ... le prince regnant d'Anhalt-Coethen
!!!OTL: Brandenburgische Konzerte F
!!!XEN: Brandenburg Concerto No. 2 in F major.
!!!OMV: Movement 1.
!!!SCT: BWV 1047
!! [Allegro]
**kern  **kern  **kern  **kern  **kern  **kern  **kern  **kern  **kern  **kern
*ICklav *ICstr  *ICstr  *ICstr  *ICstr  *ICstr  *ICstr  *ICww   *ICww   *ICbras
*Icemba *Icello *Icbass *Iviola *Ivioln *Ivioln *Ivioln *Ioboe  *Ifltds *Itromp
*IGcont *IGcont *       *       *       *       *       *       *       *
*IGripn *IGripn *IGripn *IGripn *IGripn *IGripn *IGconc *IGconc *IGconc *IGconc
!cembal !'cello !Bd'rip !Vd'rip !v'lin2 !v'lin1 !v'lino !oboe   !flauto !tromba
*k[b-]  *k[b-]  *k[b-]  *k[b-]  *k[b-]  *k[b-]  *k[b-]  *k[b-]  *k[b-]  *k[]
*F:     *F:     *F:     *F:     *F:     *F:     *F:     *F:     *F:     *F:
*M2/2   *M2/2   *M2/2   *M2/2   *M2/2   *M2/2   *M2/2   *M2/2   *M2/2   *M2/2
*MM54   *MM54   *MM54   *MM54   *MM54   *MM54   *MM54   *MM54   *MM54   *MM54
*clefF4 *clefF4 *clefF4 *clefC3 *clefG2 *clefG2 *clefG2 *clefG2 *clefG2 *clefG2
8FF/    8FF/    8FFF/   8a\     8cc\    8ff\    8ff\    8ff\    8ff\    8f/
=1      =1      =1      =1      =1      =1      =1      =1      =1      =1
16F\LL  16F\LL  16FF\LL 8f\L    8a/L    8cc\L   8cc\L   8cc\L   8cc\L   8a/L
16G\    16G\    16GG\   .       .       .       .       .       .       .
16A\    16A\    16AA\   8c\     16f/LL  16a\LL  16a\LL  16a\LL  16a\LL  8cc/
16G\JJ  16G\JJ  16GG\JJ .       16g/JJ  16b-\JJ 16b-\JJ 16b-\JJ 16b-\JJ .
16F\LL  16F\LL  16FF\LL 8f\     8a/L    8cc\L   8cc\L   8cc\L   8cc\L   8a/
16G\    16G\    16GG\   .       .       .       .       .       .       .
16A\    16A\    16AA\   8c\J    16f/LL  16a\LL  16a\LL  16a\LL  16a\LL  8f/J
16G\JJ  16G\JJ  16GG\JJ .       16g/JJ  16b-\JJ 16b-\JJ 16b-\JJ 16b-\JJ .
16F\LL  16F\LL  16FF\LL 8f\L    8a/L    8cc\L   8cc\L   8cc\L   8cc\L   8a/L
16E\    16E\    16EE\   .       .       .       .       .       .       .
16F\    16F\    16FF\   8a\     8cc/    8ff\    8ff\    8ff\    8ff\    8cc/
16G\JJ  16G\JJ  16GG\JJ .       .       .       .       .       .       .
16A\LL  16A\LL  16AA\LL 8cc\    8f/     8cc\    8cc\    8cc\    8cc\    8ff/
16B-\   16B-\   16BB-\  .       .       .       .       .       .       .
16A\    16A\    16AA\   8c\J    8cc/J   8ff\J   8ff\J   8ff\J   8ff\J   8cc/J
16G\JJ  16G\JJ  16GG\JJ .       .       .       .       .       .       .
=2      =2      =2      =2      =2      =2      =2      =2      =2      =2
*-      *-      *-      *-      *-      *-      *-      *-      *-      *-
```

Suppose we wanted to extract the 'cello part. In the above encoding, the 'cello occupies the second spine (second field) from the left, hence:

```
extract -f 2 brandenburg2.krn
```

The resulting output would begin as follows:

```
!!!COM: Bach, Johann Sebastian
!!!OPR: Six Concerts Avec plusieurs ... le prince regnant d'Anhalt-Coethen
!!!OTL: Brandenburgische Konzerte F
!!!XEN: Brandenburg Concerto No. 2 in F major.
!!!OMV: Movement 1.
!!!SCT: BWV 1047
!! [Allegro]
**kern
*ICstr
*Icello
*IGcont
*IGripn
!'cello
*k[b-]
*F:
*M2/2
*MM54
*clefF4
8FF/
=1
16F\LL
16G\
etc.
```

Notice that the **extract** command outputs all global comments. In the case of local comments, **extract** outputs only those local comments that belong to the output spine.

The oboe and flauto dolce parts are encoded in spines 8 and 9. So we could extract the 'cello, oboe and flauto dolce parts by submitting a list of the corresponding fields. Spine numbers are separated by commas:

```
extract -f 2,8,9 brandenburg2.krn
```

Numerical **ranges** can be specified using the dash. For example, if we wanted to extract all of the string parts (spines 2 through 7):

```
extract -f 2-7 brandenburg2.krn
```

With the **-f** option, field specifications may also be made with respect to the right-most field. The dollars-sign character (`$`) refers to the right-most field in the input. The trumpet part can be extracted as follows:

```
extract -f '$' brandenburg2.krn
```

(Notice the use of the single quotes to ensure that the shell doesn't misinterpret the dollar sign.) Simple arithmetic expressions are also permitted; for example '`$-1`' refers to the right-most field minus one, etc. By way of example, the command

```
extract -f '$-2' brandenburg2.krn
```

will extract the oboe part.

Extraction by Interpretation

Typically, it is inconvenient to have to determine the numerical position of various spines in order to extract them. With the **-i** option, **extract** outputs all spines containing a specified *interpretation.* Suppose we had a file containing a Schubert song, including vocal score, piano accompaniment and vocal text (encoded using `**text`). The vocal text from the file `lieder` can be extracted as follows:

```
extract -i '**text' lieder
```

(Notice again the need for single quotes in order to avoid the asterisk being interpreted by the shell.) Several different types of data can be extracted simultaneously. For example:

```
extract -i '**semits,**MIDI' hildegard
```

will extract all spines in the file `hildegard` containing `**semits` or `**MIDI` data.

An important use of the **-i** option for **extract** is to ensure that a particular input contains only a specified type of information. For example, the lower-case letter 'r' represents a rest in the `**kern` representation. If we wish to determine which sonorities contain rests, we might want to use **grep** to search for this letter. However, the input might contain other Humdrum interpretations (such as `**text`) where the presence of the letter 'r' does not signify a rest. We can ensure that our search is limited to `**kern` data by using the **extract** command:

```
extract -i '**kern' | grep  ...
```

Both exclusive interpretations and tandem interpretations can be specified with the **-i** option. For example, the following command will extract any *transposing* instruments in the score `albeniz`:

```
extract -i '*ITr' albeniz
```

Tandem interpretations are commonly used to designate instrument classes and groups, so different configurations of instruments are easily extracted. The Brandenburg Concerto shown in Example 12.1 illustrates a number of tandem interpretations related to instrumentation classes and groups. For example, the interpretation `*ICww` identifies woodwind instruments; `*ICbras` identifies brass instruments; `*ICstr` identifies string instruments. In addition, `*IGcont` identifies "continuo" instruments; `*IGripn` identifies "ripieno" instruments; and `*IGconc` identifies "concertino" instruments. The following three commands extract (1) the woodwind instruments, (2) the ripieno instruments, and (3) any vocal parts, respectively.

```
extract -i '*ICww' concerto4
extract -i '*IGrip' brandenburg2
extract -i '*ICvox' symphony9
```

Once again, more than one interpretation can be extracted simultaneously. The following command will extract the instrument-class "strings" and the instrument "oboe" from the file `milhaud`.

```
extract -i '*ICstr,*Ioboe' milhaud
```

Similarly, the following command will extract the shamisen and shakuhachi parts from a score:

```
extract -i '*Ishami,*Ishaku' hito.uta
```

The behavior of **extract** is subtly different for tandem interpretations versus exclusive interpretations. Remember that exclusive interpretations are mutually exclusive, whereas tandem interpretations are not. Consider the following Humdrum representation:

```
**foo
a
b
c
**bar
x
y
z
*-
```

The command

```
extract -i '**foo'
```

will result in the output:

```
**foo
a
b
c
*-
```

Whereas the command

```
extract -i '**bar'
```

will result in the output:

```
**bar
x
y
z
*-
```

The `**foo` and `**bar` data are mutually exclusive. Now consider an input file where `foo` and `bar` are tandem interpretations:

```
**foobar
*foo
a
b
c
*bar
```

```
x
y
z
*-
```

The command

```
extract -i '*foo'
```

will result in the output:

```
**foobar
*foo
a
b
c
*bar
x
y
*-
```

Whereas the command

```
extract -i '*bar'
```

will result in the output:

```
**foobar
*foo
*bar
x
y
z
*-
```

When searching for a particular exclusive interpretation, **extract** resets each time a new exclusive interpretation is encountered. By contrast, when **extract** finds a target tandem interpretation, it begins outputting and doesn't stop until the spine is terminated.

Using *extract* in Pipelines

Of course the output from **extract** can be used to generate inputs for other Humdrum tools. Here are a few examples.

Recall that the **census** command tells us basic information about a file. With the **-k** option, **census** will tell us the number of barlines, the number of rests, the number of notes, the highest and lowest notes, and the longest and shortest notes for a **kern input. The following commands can be used to determine this information for (1) a bassoon part, (2) all woodwind parts:

```
extract -i '*Ifagot' ives | census -k
extract -i '*ICww' ives | census -k
```

With the **midi** and **perform** commands, **extract** allows the user to hear particular parts. For example, the following command extracts the bass and soprano voices, translates them to `**MIDI` data, and plays the output:

```
extract -i '*Ibass,*Isopran' lassus | midi | perform
```

We might extract a particular part (such as the trumpet part) and use the **trans** command to transpose it to another key:

```
extract -i '*Itromp' purcell | trans -d +1 -c +2
```

In addition, we might extract a particular instrument or group of instruments for notational display using the **ms** command. The following command will extract the string parts and create a postscript file for displaying or printing.

```
extract -i '*ICstr' brahms | ms > brahms.ps
```

The UNIX **lpr** command can be used to print a file or input stream. Suppose we want to transpose the piano accompaniment for a song by Hugo Wolf up an augmented second, and then print the transposed part:

```
extract -i '*IGacmp' wolf | trans -d +1 -c +3 | ms | lpr
```

Extracting Spines that Meander

As we saw in Chapter 5, spines can move around via various spine-path interpretations. Changes of spine position will cause havoc when extracting by fields (the **-f** option); **extract** will generate an error message and terminate. With the **-i** option, **extract** will follow the material throughout the input.

Consider the following input:

```
**mip   **dip  **dip  **blip
A       a      b      x
A       a      b      x
*       *^     *      *
A       a1     a2     b      x
A       a1     a2     b      x
A       a1     a2     b      x
*-      *-     *-     *-     *-
```

Suppose we want to extract the second spine (the first `**dip`) spine. Using the field option (**-f**) will generate an error message since this spine splits. Similarly, using the interpretation (**-i**) option will fail because the output will contain *all* of the `**dip` spines.

The **extract** command provides a third **-p** option that traces specific spine *paths*. Like the **-f** option, the **-p** option requires one or more numbers indicating the *beginning* field position for the spine. The command

```
extract -p 2 ...
```

will generate the following output:

```
**dip
a
a
*^
a1      a2
a1      a2
a1      a2
*-      *-
```

In *spine-path mode,* the **extract** command follows a given spine starting at the beginning of the file, and traces the course of that spine throughout the input stream. If spine-path changes are encountered in the input (such as spine exchanges, spine merges, or spine splits) the output adapts accordingly. If the "nth" spine is selected, the output consists of the nth spine and follows the path of that spine throughout the input until it is terminated or the end-of-file is encountered. What begins as the nth column, may end up as some other column (or columns) in the input.

There are complex circumstances where the **-p** option will not guarantee an output that conforms to the Humdrum syntax. When using the **-p** option it is prudent to check the output using the **humdrum** command in order to ensure that the output is valid. A full discussion of the **-p** option is given in the *Humdrum Reference Manual.*

Field-Trace Extracting

For circumstances where the input is very complex, **extract** provides a *field-trace mode* (**-t** option) that allows the user to select any combination of data tokens from the input stream. The field-trace option is rarely used when extracting spines. Refer to the *Humdrum Reference Manual* for further information.

Extracting Passages: The *yank* Command

A useful companion to the **extract** command is the Humdrum **yank** command. The **yank** command can be used to selectively extract segments or passages from a Humdrum input. The yanked material can be identified by absolute line numbers, or relative to some marker. In addition, **yank** is able to output logical segments, such as measures, phrases, or labelled sections, and is able to output material according to content. The output always consists of complete records; **yank** never outputs partial contents of a given input record.

The **yank** command provides five different ways of extracting material. The simplest way of yanking material is by specifying ranges of line numbers. In the following command, the **-l** option invokes the line-number operation. The **-r** option is used to specify the range. Ranges are defined by integers separated by commas, or with a dash indicating a range of consecutive values. For example, the following command selects lines, 5, 13, 23, 24, 25 and 26 from the file named `casella`:

```
yank -l -r 5,13,23-26 casella
```

The dollar sign ($) can be used to refer to the last record in the input. For example, the following command yanks the first and last records from the file `mossolov`.

```
yank -l -r '1,$' mossolov
```

Once again note that single quotes are needed here in order to prevent the shell from misinterpreting characters such as the dollar sign or the asterisk. Records close to the end of the input can be specified by subtracting some value from $. For example, the following command yanks the first 20 records from the last 30 records contained in the file `ginastera`. Notice that the dash/minus sign is used both to convey a range and as an arithmetic operator.

```
yank -l -r '$-30-$-10' ginastera
```

If **yank** is given a Humdrum input, it always produces a syntactically correct Humdrum output. All interpretations prior to and within the yanked material are echoed in the output. The **yank** command also appends the appropriate spine-path terminators at the end of the yanked segment. By way of example, if we yanked line 10 (containing 4 spines) and line 100 (containing 5 spines), **yank** will include in the output the appropriate spine-path interpretations that specify how 4 spines became 5 spines.

Yanking by Marker

Alternatively, **yank** can output lines relative to some user-defined *marker.* This mode of operation can be invoked using the **-m** option. Markers are specified using regular expressions. The range option (**-r**) specifies which lines are to be output whenever a marker is encountered. For example, the following command outputs the first and third data records following each occurrence of the string "XXX" in the file `wieck`.

```
yank -m XXX -r 1,3 wieck
```

If the value zero is specified in the range, the record containing the marker is itself output.

Since markers are interpreted by **yank** as regular expressions, complex markers can be defined. For example, the following command yanks the first data record following any record in the file `franck` beginning with a letter and ending with a number:

```
yank -m '^[a-zA-Z].*[0-9]$' -r 1 franck
```

Using **yank -m** with a range defined as zero is an especially useful construction:

```
yank -m regexp -r 0
```

This command is analogous to the familiar **grep** command. However, the output from **yank** will preserve all of the appropriate interpretations. In short, **yank** guarantees that the output conforms to the Humdrum syntax, whereas **grep** does not.

Suppose, for example, that we wanted to calculate the pitch intervals between notes that either begin or end a phrase in a monophonic input. If we use **grep** to search for `**kern` phrase indicators, we will be unable to process the resulting (non-Humdrum) output, since it will typically consist of just data records:

```
grep [{}] sibelius
```

By contrast, the comparable **yank** command preserves the Humdrum syntax and so allows us to pipe the output to the melodic interval command:

```
yank -m [{}] -r 0 sibelius | mint
```

Yanking by Delimiters

It is often convenient to yank material according to logical segments such as measures or phrases. In order to access such segments, the user must specify a segment *delimiter* using the **-o** option or the **-o** and **-e** options. For example, common system barlines are represented by the presence of an equals sign (=) at the beginning of a data token. Thus the user might yank specific measures from a file by defining the appropriate barline delimiter and providing a range of (measure) numbers. Consider the following command:

```
yank -o ^= -r 1,12-13,25 joplin
```

This command will extract the first, twelfth, thirteenth and twenty-fifth measures from the file `joplin`. Unlike the **-m** option, the **-o** option interprets the range list as *ordinal* occurrences of segments delineated by the delimiter. Whole segments are output rather than specified records as is the case with **-m.** As in the case of markers, delimiters are interpreted according to regular expression syntax. Each input record containing the delimiter is regarded as the *start* of the next logical segment. In the above command, the range (**-r**) specifies that the first, twelfth, thirteenth, and twenty-fifth logical segments (measures) are to be yanked. All records starting with the delimiter record are output up to, but not including, the next occurrence of a delimiter record.

Where the input stream contains data prior to the first delimiter record, this data may be addressed as logical segment "zero." For example,

```
yank -o ^= -r 0 mahler
```

can be used to yank all records prior to the first common system barline. Notice that *actual* measure numbers are irrelevant with the **-o** option: **yank** selects segments according to their *ordinal* position in the input stream rather than according to their *cardinal* label.

Not all segments are defined by a single marker. For example, unlike barlines, `**kern` phrases are marked by separate phrase-begin signifiers ('{') and phrase-end signifiers ('}'). The **-e** option for **yank** can be used to explicitly identify markers that *end* a segment. For example, the following command extracts the first four phrases in the file `tailleferre`:

```
yank -o { -e } -r '1-4' tailleferre
```

When the **-n** option is invoked, however, **yank** expects a numerical value to be present in the input immediately following the user-specified delimiter. In this case, **yank** selects segments based on their numbered label rather than their ordinal position in the input. For example,

```
yank -n ^= -r 12 goldberg
```

will yank all segments beginning with the label `=12` in the input file `goldberg`. If more than one segment carries the specified segment number(s), all such segments are output. That is, if there are five measures labelled "measure 12", all five measures will be output. Note that the dollar sign anchor cannot be used in the range expression for the **-n** option. Note also that input tokens containing non-numeric characters appended to the number will have no effect on the pattern match. For example, input tokens such as `=12a`, `=12b`, or `=12;` will be treated as equivalent to `=12`.

As in the case of the **-o** option, a range of zero ('0') addresses material prior to the first delimiter record. (N.B. This behavior is unlike the **-m** option where zero addresses the record itself.) Like the **-o** option, the value zero may be reused for each specified input file. Thus, if `file1`, `file2` and `file3` are Humdrum files:

```
yank -n ^= -r 0 file1 file2 file3
```

will yank any leading (anacrusis) material in each of the three files.

Yanking by Section

When the **-s** option is invoked, **yank** extracts passages according to Humdrum section labels encoded in the input. Humdrum section labels will be described fully in Chapter 20. For now, we can simply note that section labels are tandem interpretations that conform to the syntax:

**>label_name*

Label names can include any character except the tab. Labels are frequently used to indicate formal divisions, such as coda, exposition, bridge, second ending, trio, minuet, etc. The following command yanks the second instance of a section labelled `First Theme` in the file `mendelssohn`:

```
yank -s 'First Theme' -r 2 mendelssohn
```

Note that with "through-composed" Humdrum files it is possible to have more than one section containing the same section-label. Such situations are described in Chapter 20.

Examples Using *yank*

As mentioned earlier, **yank** will always produce a syntactically correct Humdrum output if given a proper Humdrum input. All interpretations prior to, and within, the yanked material are echoed in the output.

Any *comments* prior to the yanked passage may be included in the output by specifying the **-c** option.

The following examples illustrate how the **yank** command may be used.

```
yank -l -r 1120 messiaen
```

yanks line 1120 in the file `messiaen`.

```
yank -n ^= -r 27 sinfonia
```

yanks numbered measures 27 from the `**kern` file `sinfonia`.

```
yank -n ^= -r 10-20 minuet waltz
```

yanks numbered measures 10 to 20 from both the `**kern` files `minuet` and `waltz`.

```
yank -o ^= -r '0,$' fugue ricercar
```

yanks any initial anacrusis material plus the final measure of both `fugue` and `ricercar`.

```
cat fugue ricercar | yank -o ^= -r '0,$'
```

yanks any initial anacrusis material from the file `fugue` followed by the final measure of `ricercar`.

```
yank -n 'Rehearsal Marking ' -r 5-7 fugue ricercar
```

yanks segments beginning with the labels `"Rehearsal Marking 5"`, `"Rehearsal Marking 6"`, and `"Rehearsal Marking 7"`. Segments are deemed to end when a record is encountered containing the text `"Rehearsal Marking "`.

```
yank -o { -e }  -r '1-$' webern
```

yanks all segments in the file `webern` beginning with a record containing "{" and ending with a record containing "}." The command:

```
yank -o { -e } -r '1-4,$-3-$' faure
```

yanks the first four and last four segments in the file `faure`, where segments begin with an open brace ({) and end with a closed brace (}). In the `**kern` representation, this would extract the first four and last four phrases in the file.

```
yank -s Coda -r 1 stamitz
```

will yank the first occurrence of a section labelled `Coda` in the file `stamitz`.

Note that yanked segments are output in exactly the order they appear in the input file. For example, assuming that measure numbers in an input stream increase sequentially, **yank** is unable to

output measure number 6 prior to outputting measure number 5. The order of output material can be rearranged by invoked the **yank** command more than once (e.g. `yank -l -r 100 ...; yank -l -r 99 ...; yank -l -r 98 ...`).

Using *yank* in Pipelines

Like the other tools we have examined, **yank** can be profitably used in conjunction with other Humdrum tools. It is often useful to employ more than one **yank** in a pipeline. In the following command, the first **yank** isolates the 'Trio' section from the input file, and the second **yank** isolates the first four measures of the extracted Trio:

```
yank -s Trio dvorak | yank -o ^= 1-4
```

Similarly, we can link two **yank** commands to extract particular phrases from specified sections. For example, suppose we wanted to compare the first phrase of the exposition with the first phrase of the recapitulation:

```
yank -s Exposition haydn | yank -o { -e } -r 1 > Ephrase
yank -s Recapitulation haydn | yank -o { -e } -r 1 > Rphrase
```

Suppose we want to know how many notes there are in measures 8-16 in a `**kern` file named `borodin`.

```
yank -n = -r 8-16 borodin | census -k
```

Are there any subdominant chords between measures 80 and 86?

```
yank -n = -r 80-86 elgar | solfa | grep fa | grep la | grep do
```

How frequent is the dominant pitch in Strauss' horn parts?

```
extract -i '*Icor' strauss | solfa | grep -c so
```

Combining **yank** and **extract** can be especially useful. What is the highest note in the trumpet part in measure 29?

```
extract -i '*Itromp' tallis | yank -n = -r 29 | census -k
```

Also, we can combine **yank** with the **midi** and **perform** commands to hear particular sections. Play the Trio section in a Waldteufel waltz:

```
yank -s 'Trio' -r 1 waldteufel | midi | perform
```

Listen to the soprano clarinet part in the fourth and eighth phrases.

```
extract -i '*Iclars' quintet | yank -o { -e } -r 4,8 \
   | midi | perform
```

Note that when using **yank** to retrieve passages by markers (such as phrase marks), care must be taken since markers may be miscoordinated between several concurrent parts. Example 12.2 shows a passage that has overlapping phrases. When trying to extract a particular phrase for a particular part, the outputs will differ significantly depending on whether the **yank** command is invoked *before* or *after* the **extract** command.

Example 12.2. A Passage Containing Unsynchronized Phrases.

```
**kern   **kern
=1-      =1-
2r       8r
.        {8g
.        8a
.        8b
=2       =2
8r       4cc
{8e      .
8f       4dd}
8a       .
=3       =3
8g       {4ee
8e       .
4d}      4ff
=4       =4
*-       *-
```

The order of execution for some commands may cause some subtle differences. Suppose we wanted to identify the melodic intervals present in measures 8-32 for some work by Sibelius. The following two commands are likely to produce different results:

```
yank -n = -r 8-32 sibelius | mint
mint sibelius | yank -n = -r 8-32
```

In the second case, an interval will probably be calculated between between the last note of measure 7 and the first note of measure 8. This interval will be absent in the first case.

Reprise

In this chapter we have learned how to extract musical parts using **extract** and how to grab musical passages using **yank.** We saw that the **extract** command is also useful for isolating specific types of information, such as the lyrics, or ensuring that no other type of information is present in a data stream. In the case of **yank** we saw that passages can be extracted by defining arbitrary delimiters. In addition to extracting by measures, by sonorities, or by labelled sections, we can extract by rests, phrase marks — in fact, by any user-defined marker. We also saw how the command **yank -m** *regular-expression* **-r 0** can be used as a more sophisticated version of **grep** — a search tool that ensures the output will conform to the Humdrum syntax.

In the next chapter we will discuss how segments of music can be put back together again.

Chapter 13

Assembling Scores

In the previous chapter we learned how to extract parts and passages from Humdrum files. In this chapter we discuss the reverse procedures: how to assemble and coordinate larger documents from individual segments and parts. We will discuss four tools: the UNIX **cat** command, and the Humdrum **assemble**, **timebase** and **rid** commands.

The *cat* Command

The UNIX **cat** command allows two or more inputs to be concatenated together. If we concatenate two files, the output will consist of the contents of the first file, followed immediately by the contents of the second file. For example, the following command will concatenate together the files `mov.1`, `mov.2` and `mov.3` and place the result in the file `complete`. The order of concatenation is the same as the order of file names given on the command line:

```
cat mov.1 mov.2 mov.3 > complete
```

If each of the concatenated files conforms to the Humdrum syntax, then the resulting combined file is guaranteed to conform to the Humdrum syntax. However, in many cases, there may be redundant information present in the concatenated output. Suppose we had three files, each of which encoded a single measure of music. Concatenating them together might result in an output such as the following:

```
!! File 1
**kern **kern **kern **kern
*M4/4  *M4/4  *M4/4  *M4/4
*G:    *G:    *G:    *G:
*k[f#] *k[f#] *k[f#] *k[f#]
4GG    4B     4d     4g
=1     =1     =1     =1
4GG    4d     4g     4b
4FF#   4d     [4a    4dd
8GG    4d     8a]    4b
8BB    .      [8g    .
4D     4d     8g]    4a
```

```
.       .       8f#     .
*-      *-      *-      *-
!! File 2
**kern  **kern  **kern  **kern
*M4/4   *M4/4   *M4/4   *M4/4
*k[f#]  *k[f#]  *k[f#]  *k[f#]
=2      =2      =2      =2
8G      4d      4g      4b
8F#     .       .       .
4E      4e      4g      4cc#
4D;     4A;     4f#;    4dd;
8A      4cn     4a      4ee
8G      .       .       .
*-      *-      *-      *-
!! File 3
**kern  **kern  **kern  **kern
*M4/4   *M4/4   *M4/4   *M4/4
*k[f#]  *k[f#]  *k[f#]  *k[f#]
*k[f#]  *k[f#]  *k[f#]  *k[f#]
=3      =3      =3      =3
4F#     4d      [4a     8dd
.       .       .       8cc
4G      4d      8a]     4b
.       .       [8g     .
8C#     8e      8g]     4a
8D#     4B      8f#     .
8E      .       8e      8g
8F#     8A      8d      8a
*-      *-      *-      *-
```

Notice that each complete measure ends with spine-path terminators and that the `**kern` exclusive interpretations are repeated. This organization has a number of repercussions for various Humdrum tools. For example, the **mint** command calculates melodic intervals between successive notes within a spine. However, **mint** will not calculate intervals between pitches that are separated by a spine-path terminator. In other words, in the above output, **mint** will fail to calculate the melodic intervals between notes in successive measures.

The *rid* Command

This problem can be resolved by using the Humdrum **rid** command. The **rid** command can be used to eliminate various kinds of records. Each option for **rid** eliminates a different class of records. Here are the record classes with their associated options:

-G eliminate all global comments
-g eliminate only global comments that are empty
-L eliminate all local comments
-l eliminate only local comments that are empty

-I eliminate all interpretations
-i eliminate only null interpretations
-D eliminate all data records
-d eliminate only null data records
-T eliminate all tandem interpretations
-t eliminate duplicate (repeated) tandem interpretations
-U eliminate unnecessary exclusive interpretations (see below)
-u same as -U

Null records are devoid of content. For example, null interpretations consist of a single asterisk in each spine; null data record consists of just null data tokens (.) in each spine; null local comments consist of a single exclamation mark in each spine. Null global comments contain just two exclamation marks at the beginning of a record.

Upper- and lower-case options are used to distinguish *all* records of a certain class (upper-case) from *empty*, *null* or *repeated* records of a certain class (lower-case).

By way of example, the following command will eliminate all global comments (including reference records) from the input:

```
rid -G Saint-Saens
```

Similarly, the following command will eliminate all tandem interpretations from an input:

```
rid -T Vaughan-Williams
```

Options can be combined. The following command eliminates all global and local comments, interpretations, and null data records:

```
rid -GLId
```

The option combination **-GLId** is very common with **rid** since only non-null data records are retained in the output.

With the **-u** option, **rid** will remove "unnecessary" exclusive interpretations. Exclusive interpretations are deemed unnecessary if they don't change the current status of the data. In the following example, the second `**psaltery` interpretation is redundant. The **rid -u** command would remove the first spine-path terminator and the second exclusive interpretation — leaving a continuous data spine.

```
**psaltery
.
.
.
*-
**psaltery
.
*-
```

In addition, **rid** provides a **-t** option which removes "duplicate" or repeated tandem interpretations. In the above example there is no need to repeat the meter signature and key signature in each measure. The following command will concatenate each of the three measures together, and then eliminate the unwanted interpretations:

```
cat bar1 bar2 bar3 | rid -ut
```

The resulting output is given below:

```
!! File 1
**kern  **kern  **kern  **kern
*M4/4   *M4/4   *M4/4   *M4/4
*k[f#]  *k[f#]  *k[f#]  *k[f#]
4GG     4B      4d      4g
=1      =1      =1      =1
4GG     4d      4g      4b
4FF#    4d      [4a     4dd
8GG     4d      8a]     4b
8BB     .       [8g     .
4D      4d      8g]     4a
.       .       8f#     .
!! File 2
=2      =2      =2      =2
8G      4d      4g      4b
8F#     .       .       .
4E      4e      4g      4cc#
4D;     4A;     4f#;    4dd;
8A      4cn     4a      4ee
8G      .       .       .
!! File 3
=3      =3      =3      =3
4F#     4d      [4a     8dd
.       .       .       8cc
4G      4d      8a]     4b
.       .       [8g     .
8C#     8e      8g]     4a
8D#     4B      8f#     .
8E      .       8e      8g
8F#     8A      8d      8a
*-      *-      *-      *-
```

Of course care should be exercised when concatenating inputs together. Although an output may conform to the Humdrum syntax, the result can nevertheless violate conventions for a specific representation such as `**kern`. For example, if we were to concatenate measure 85 to measure 87, it is possible that tied-notes won't match up, or that phrases will begin without ending, etc. These anomalies may cause problems with subsequent processing.

Assembling Parts Using the *assemble* Command

Assembling parts into a full score is slightly more challenging than concatenating together musical segments. The principle tool for joining spines together is the **assemble** command. Consider the following two files:

```
!! Assemble example
!! File 1
**Letters
! A to E
A
B
C
D
E
*-
```

```
!! Assemble example
!! File 2
**Numbers
! 1 to 5
1
2
3
4
5
*-
```

These two files can be aligned side by side using **assemble**:

```
assemble letters numbers
```

The resulting output is:

```
!! Assemble example
!! File 1
!! File 2
**Letters **Numbers
! A to E  ! 1 to 5
A         1
B         2
C         3
D         4
E         5
*-        *-
```

Note the following: (1) The spines are joined side by side from left to right in the same order as specified on the command line. (2) Local comments are preserved in their appropriate spines. (3) When identical global comments occur at the same location in both files, only a single instance of

the comment is output. (4) Dissimilar global comments are output successively.

The files joined by **assemble** are not confined to a single spine. For example, one input file may contain 2 spines and a second file may contain 20 spines. The resulting output will contain 22 spines. There is no limit to the number of files that can be assembled at one time.

In many cases, the input files will have dissimilar lengths. The **assemble** command will correctly terminate the appropriate spines. For example, in the above case, if the `numbers` file contained only the numbers 1 to 3, the assembled output would appear as follows:

```
!! Assemble example
!! File 1
!! File 2
**Letters **Numbers
! A to E  ! 1 to 3
A         1
B         2
C         3
*         *-
D
E
*-
```

If the order of the input files was reversed, **assemble** would produce an output with the appropriate spine-path changes:

```
!! Assemble example
!! File 2
!! File 1
**Numbers   **Letters
! 1 to 3    ! A to E
1           A
2           B
3           C
*-          *
D
E
*-
```

Note that if all of the input files conform to the Humdrum syntax, then **assemble** guarantees that the assembled output will also conform to the Humdrum syntax.

Aligning Durations Using the *timebase* Command

Suppose now that we wanted to join two hypothetical files containing `**kern` data. The first file contains two quarter notes, whereas the second file contains four eighth notes:

```
!! File 1
**kern
4c
4d
*-

!! File 2
**kern
8e
8g
8f
8g
*-
```

Using **assemble** to paste them together will clearly lead to an uncoordinated result. The two quarter notes in file 1 will be incorrectly matched with the first two eighth notes in file 2.

The Humdrum **timebase** command can be used to reformat either `**kern` or `**recip` inputs so that each output data record represents an equivalent slice (elapsed duration) of time. (Barlines are ignored by **timebase**.) The **timebase** command achieves this by padding an input with null data records. In the above case, we would preprocess file 1 as follows:

```
timebase -t 8 file1 > file1.tb
```

The new file would look like this:

```
!! File 1
**kern
4c
.
4d
.
*-
```

The **-t** option is used to indicate the "time base" — in this case, an eighth duration. Since all non-barline data records in both files represent elapsed durations of an eighth-note, we can continue by using the **assemble** command as before. The command:

```
assemble file1.tb file2
```

will result in the following two-part score:

```
!! File 1
!! File 2
**kern      **kern
4c          8e
.           8g
4d          8f
```

```
.           8g
*-          *-
```

Suppose that `file2` also contained a quarter-note. For example, consider a revised `file2`:

```
!! File 2
**kern
8e
8g
4f
*-
```

Before assembling the two parts, we would need to apply the **timebase** command to this file (using the same 8th-note time-base value). Assembling the two "time-based" files would produce the following result:

```
!! File 1
!! File 2
**kern      **kern
4c          8e
.           8g
4d          4f
.           .
*-          *-
```

Notice that we have a spurious null data record in the last line; both parts encode null tokens. For most processing, the presence of null data records is inconsequential. However, if we wish, these null data records can be eliminated using the **rid** command with the **-d** option. In fact it is common to follow an **assemble** command with **rid -d** to strip away unnecessary null data records. The command:

```
assemble file1.tb file2.tb | rid -d
```

would result in the following output:

```
!! File 1
!! File 2
**kern      **kern
4c          8e
.           8g
4d          4f
*-          *-
```

The **timebase** command can be applied to multi-spine inputs as well as single-spine inputs. Consider, the following input:

```
**kern   **kern **kern **kern **commentary
4g       8.r    8.cc   16ee   2nd inversion
.        .      .      8ff    .
.        32b    16cc   16gg   clash
.        32a    .      .      .
8f       8cc    8dd    8ff    suspension
*-       *-     *-     *-     *-
```

The following command will cause the addition of null data records so that each data record represents an elapsed time of a 32nd duration. Incidentally, notice that any spine contain non-rhythmic data — such as the `**commentary` spine in the above example — is also transformed so that synchronous data is maintained.

```
timebase -t 32 Corelli
```

The corresponding output is as follows.

```
**kern   **kern **kern **kern **commentary
*tb32    *tb32  *tb32  *tb32  *tb32
4g       8.r    8.cc   16ee   2nd inversion
.        .      .      .      .
.        .      .      8ff    .
.        .      .      .      .
.        .      .      .      .
.        .      .      .      .
.        32b    16cc   16gg   clash
.        32a    .      .      .
8f       8cc    8dd    8ff    suspension
.        .      .      .      .
.        .      .      .      .
.        .      .      .      .
*-       *-     *-     *-     *-
```

Notice that **timebase** has added a tandem interpretation (`*tb32`). This indicates that the output has been processed so that each non-barline data record represents an elapsed duration equivalent to a thirty-second note.

Assembling N-tuplets

Typically, one can simply use the shortest duration present as a guide for a suitable time-base value. The shortest duration can be determined using the **census -k** command described in Chapter 4. However, tuplets require a little more sophistication. Suppose we wanted to assemble two parts, one containing just eighth-notes and the other containing just quarter-note triplets. (The quarter-note triplets will be encoded as three notes in the time of a half-note, or "6th" notes.) We need to create an output whose rhythmic structure will appear as follows:

```
**kern    **kern
*M2/4     *M2/4
=1        =1
6         8
.         8
6         .
.         8
6         .
.         8
=2        =2
6         8
.         8
6         .
.         8
6         .
.         8
=3        =3
*-        *-
```

In this case, choosing a time-base according to the shortest duration (8th) will not work since a 6th note is not an integral multiple of the eighth duration. We need to find a *common duration factor* for both values. The shortest common duration is a 24th note (there are three 24th notes in an 8th note, and four 24th notes in a 6th note). Applying the time-base value '24' to both files will allow us to coordinate them properly. Remember that **rid -d** can be used to eliminate unnecessary null data records once we have finished assembling the spines. In the worst case, you can determine a common duration factor by simply multiplying together the shortest notes in the files to be assembled. For example, $6 \times 8 = 48$; so a time-base of 48 will be guaranteed to work for both files.

Checking an Assembled Score Using *proof*

In assembling any score from a set of parts, there is always the danger of using the wrong time-base value. When parts are miscoordinated, it is typically the consequence of one or more notes being discarded by **timebase**. Fortunately, such miscoordinations are easily detected by applying the **proof** command to any assembled `**kern` output. The **proof** utility checks `**kern` representations for a wide variety of possible encoding errors or ambiguities:

```
proof fullscore
```

By way of summary, creating a full score from a set of `**kern` parts involves the following five tasks: (1) Identify a common duration factor for all the parts. Use **census** to determine the shortest duration; if any of the parts contains an N-tuplet, then the common duration factor may be smaller than the shortest note. (2) Use the **timebase** command to expand each input file separately using the common duration factor. (3) Assemble the parts using **assemble**. (4) If desired, eliminate unnecessary null data records using **rid -d**. (5) Check the assembled score for rhythmic coherence using the **proof** command.

Other Uses for the *timebase* Command

The most common use of **timebase** is as a way of expanding a file by padding it with null data records. However, **timebase** can also be used to contract a file, giving us only those sonorities that lie a fixed duration apart. For example, specifying a time-base of **-t 2** will cause only those sonorities that are separated by a half-note duration to be output. This sort of rhythmic reduction can be useful in certain circumstances. For example, suppose you suspect there may be a hemiola tendency in a given work by Brahms, where the duration separating hemiola notes is a dotted-quarter. The command:

```
timebase -t 4. brahms
```

can be used to extract only those sonorities that are separated by a dotted-quarter duration.

Similarly, suppose we want to extract all sonorities falling on the third beat of a waltz written in 3/2 meter. First we would edit the input file so it begins on the third beat of some measure. Then we could use the following command:

```
grep -v ^= waltz | timebase -t 1. > 3rd_beat
```

Note that the use of **grep** here is essential in order to eliminate barlines. The **timebase** command resets itself with each barline, so time-base durations are calculated from the beginning of the bar. When barlines are eliminated, **timebase** cannot synchronize to the beginning of each bar and so simply floats along at the fixed time-base.

Additional Uses of *assemble* and *timebase*

Although we normally assemble parts together, sometimes it is useful to assemble entire scores together. Suppose we wanted to listen to a theme at the same time as one of its variations. We might first use **yank** to extract the appropriate sections. At the same time we might determine a common duration factor and expand them using **timebase**.

```
yank -s Theme -r 1 blacksmith | timebase -t 32 > temp1
yank -s 'Variation 1' -r 1 blacksmith | timebase -t 32 > temp2
```

Then we assemble the two sections together, translate to the `**MIDI` representation and use **perform** to listen to both sections at the same time:

```
assemble temp1 temp2 | midi | perform
```

Similarly, suppose we would like to compare the bass lines for each variation in some set. We might extract each of the bass lines, assemble them into a single score, and then use the **ms** and **ghostview** commands to allow us to see all of the bass lines for all of the variations concurrently.

```
yank -s 'Variation 1' -r 1 goldberg | timebase -t 16 > temp1
yank -s 'Variation 2' -r 1 goldberg | timebase -t 16 > temp2
  etc. ...
assemble temp1 temp2 temp3 ... | rid -d | ms > basslines.ps
```

```
ghostview basslines.ps
```

The most common use of **assemble** is not to assemble parts, but to assemble different types of concurrent information. Suppose we would like to determine whether descending minor seconds are more likely to be *fah-mi* rather than *doh-ti*. We can use the **mint** command to characterize melodic intervals, and the **solfa** command to characterize scale degrees. Assume that our input is monophonic:

```
mint melodies > temp1
solfa melodies > temp2
```

The files `temp1` and `temp2` will have the same length, so we can assemble them together. This will generate an output consisting of two spines, `**mint` and `**solfa`. In effect, the `**mint` spine data will tell us the interval used to approach the scale degree encoded in the `**solfa` spine. We can use **grep** to search for the appropriate combinations of interval and scale degree and count the number of occurrences:

```
assemble temp1 temp2 | grep -c '-m2.*mi'
assemble temp1 temp2 | grep -c '-m2.*ti'
```

This same approach can be used to address (innumerable) questions pertaining to concurrent patterns. For example, suppose we have a `**harm` spine that identifies the 'Roman numeral' functional harmony for some choral work. We can identify complex situations such as the following: for the soprano voice, count how many subdominant pitches are approached by an interval of a rising third or a rising sixth and coincide with a dominant seventh chord. First, let's extract the soprano line and create a corresponding scale degree representation using **deg**. We can use the **-a** option to avoid outputting the melodic direction signifiers (^ and v):

```
extract -i '*Isopran' howells | deg -a > temp1
```

Next, let's again extract the soprano voice and create a corresponding melodic interval representation using **mint**. Since we are not interested in interval qualities we can invoke the **-d** option to output only diatonic interval sizes.

```
extract -i '*Isopran' howells | mint -d > temp2
extract -i '**harm' howells > temp3
```

We have also extracted the `**harm` spine and placed it in the file `temp3`. If we assemble together our three temporary files, the result will have three spines: `**deg`, `**mint` and `**harm`. We can now use **grep** to search and count all instances of subdominant pitches that are approached by ascending thirds/sixths and that coincide with dominant seventh chords (in the `**kern` representation: 'V7'):

```
assemble temp1 temp2 temp3 | grep -c '^4<tab>+[36]<tab>V7
```

The **timebase** command can also be used for tasks other than assembling parts together. Suppose we would like to determine whether secondary dominant chords are more likely to appear on the third beat than other beats in a triple meter work. The **timebase** command can be used to reformat a score so that each measure occupies the same number of data records. For example, in a 3/4 me-

ter, an eighth-note time-base will mean that each measure will contain six data records, and the fifth data record will correspond to the onset of the third beat. Recall from Chapter 12 that the **yank -m** command allows us to extract particular data records following a specified marker. In the following command, we have defined the marker as a barline (`-m ^=`) and instructed **yank** to fetch the fifth line following each occurrence of the marker (`-r 5`). In our example, the **grep** command is being used to count V/V chords occurring on third beats:

```
timebase -t 8 strauss | solfa | yank -m = -r 5 | grep re \
   | grep fe | grep -c la
```

We can repeat this command for beats one and two by changing the **-r** parameter to 1 and 3 respectively.

Reprise

In this chapter we have learned how to concatenate musical passages together using the **cat** command. We also learned how to eliminate redundant exclusive and tandem interpretations from concatenated outputs using the **-u** and **-t** options for **rid.** In addition, we learned how to assemble two or more spines into a single output file using **assemble**. In the case of `**kern` and `**recip` representations, we learned how to use the **timebase** command to preprocess each constituent file so that all data records represent equivalent elapsed durations. Having assembled a full score from parts, **rid -d** can be used to eliminate any residual or unnecessary null data records. The **proof** command can be used to ensure that any assembled `**kern` data is correctly aligned.

Finally, we learned that the **timebase** command can be used for other analytic purposes. Specifically, it can be used to reduce a score rhythmically so only particular onset points or beats are retained. In Chapter 23 we will see additional uses of **timebase** for a variety of types of rhythmic tasks.

Chapter 14

Stream Editing

Most computer users are familiar with editing an electronic document using an interactive word-processor or text editor. *Stream editors* are non-interactive editors that automatically process a given input according to a user-specified set of editing instructions. A stream editor can be used, for example, to automatically transform a document from British spelling to American spelling. Stream editors are especially useful when processing large numbers of documents — such as a series of files encoding some musical repertory. In this chapter we will introduce two stream editors: **sed** and **humsed**.

The *sed* and *humsed* Commands

The **humsed** command is simply a Humdrum version of the common **sed** stream editor. The syntax and operation of **sed** and **humsed** are virtually identical. However, **humsed** will modify only Humdrum data records, whereas **sed** will modify any type of record, including Humdrum comments and interpretations. Both stream editors provide operations for *substitution*, *insertion*, *deletion*, *transliteration*, *file-read* and *file-write*. When used in combination, these operations can completely transform an input stream or document.

Simple Substitutions

The most frequently used stream-editing operation is substitution. Both **humsed** and **sed** designate substitutions by the lower-case letter `s`. Substitutions require two strings: the *target string* to be replaced, and the *replacement string* to be introduced. The syntax for substitutions is as follows:

s<*delimiter*><*target string*><*delimiter*><*replacement string*><*delimiter*><*options*>

No spaces are permitted between these elements. The delimiter can be any character; however, the same delimiter character must be used throughout the operation. The following substitution command causes occurrences of the letter 'A' to be replaced by the letter 'B':

```
s/A/B/
```

Since the slash character (/) appears immediately following the `s`, it becomes the delimiter for the

rest of the operation. In this case no option has been given at the end of the substitution. Since the delimiter can be any character, the above command is functionally identical to the following:

```
sxAxBx
```

If it is necessary to use the delimiter character (as a literal) within either the target string or the replacement string it can be escaped using the backslash character.

There are two ways to execute a substition operation such as given above. One way is to give the substitution as a command-line argument to **sed** or **humsed**:

```
humsed s%A%B% filename
```

Alternatively, the operation can be placed in a file (for example, named `revise`):

```
s%A%B%
```

Then the stream editor can be invoked to execute the operations contained in this file using the **-f** option:

```
humsed -f revise inputfile
```

By default the output will be displayed on the screen. Using file-redirection (>) the output can be placed in some other file. Note that you should never redirect the output to the same file as the input — this will destroy the original input file. If necessary, send the output to a temporary file, and then use the UNIX **mv** command to rename the output.

Suppose that you had encoded a musical work in the `**kern` representation. Having finished the encoding, you realize that what you thought were *pizzicato* marks are really *spiccato* marks. In the `**kern` representation, pizzicatos are indicated by the double quote (`"`) whereas spiccatos are represented by the lower-case letter `s`. We can change all pizzicato marks to spiccato marks using the following command:

```
humsed 's/"/s/g' inputfile
```

Since the double quote is interpreted as a special character by the UNIX shell, we have escaped the entire substitution operation by placing it in single quotes. (Alternatively, we could place a backslash immediately before the double-quote character.) Note also the presence of the `g` option at the end of the string. Permissible options include any positive integer or the letter `g`. Without any option, the **sed** and **humsed** substitute (s) operation will replace only the *first* occurrence of the string in each data record. The `g` option specifies a "global" substitution, in that all occurrences on a given data record are replaced. If the option consisted of the number '3', then only the third instance of the target string would be replaced on each line.

Selective Elimination of Data

The *target string* in substitution operations is actually a regular expression. This means that we can specify patterns using the full power of regular expression syntax. Frequently, it is useful to

eliminate certain kinds of information from a file. For example, we can eliminate all sharps and flats from a **kern-format file as follows:

```
humsed s/[#-]//g inputfile
```

Suppose we wanted to eliminate all beaming information in a score. In the **kern representation, open and closed beams are represented by L and J respectively; partial beams are represented by K and k.

```
humsed s/[JLkK]//g inputfile
```

Alternatively, we might want to eliminate all data except for the beaming information:

```
humsed s/[^JLkK]//g inputfile
```

Sometimes we need to restrict the circumstances where the data are eliminated. For example, we might want to eliminate all measure numbers. Eliminating all numbers from a **kern file will have the undesirable consequence of eliminating all note durations as well. Most **humsed** operations can be *preceded* by a regular expression delineated by slashes. This tells **humsed** to execute this substitution only if the data record matches the leading regular expression. For example, the following command eliminates measure numbers but not note durations:

```
humsed /^=/sX[0-9]*XXg inputfile
```

The operation may be interpreted as follows: look for lines that match a pattern where the first character in the line is an equals sign; if you find this pattern look for zero or more instances of any number between zero and nine, and replace that by an empty string; do this substitution for all numbers on the current data record.

Incidentally, Humdrum provides a **num** command that can be used to insert numbers in data records. The **num** command supports an elaborate set of options, but is not used often, so we won't describe it here. The following command renumbers all of the barlines in an input so that the first measure begins with the number 72. (Refer to the *Humdrum Reference Manual* for details regarding **num**.)

```
humsed /^=/sX=[0-9]*X=Xg inputfile | num -n ^= -x == -p = -o 72
```

Suppose we wanted to eliminate all octave numbers from a **pitch representation. In this case we want to delete all numbers except when they occur in conjunction with a barline. Our substitution should occur only when the current record does not match a leading equals sign:

```
humsed /^[^=]/s%[0-9]%%g inputfile
```

Suppose we wanted to determine which of two MIDI performances exhibits more dynamic range — that is, which performance has a greater variability in key-down velocities. Recall from Chapter 7 that MIDI data tokens consist of three elements separated by slashes (/). The third element is the key velocity. First, we want to eliminate key-up data tokens. These tokens can be distinguished by the minus sign associated with the second data element. An appropriate substutition is:

```
s%[0-9][0-9]*/-[0-9][0-9]*/[0-9]* *%%g
```

(That is, replace by nothing any data that matches the following: a numerical digit followed by zero or more digits, followed by a slash, followed by a minus sign, followed by a digit, followed by zero or more digits, followed by a slash, followed by zero or more digits, followed by zero or more spaces.)

Having isolated only the key-down data tokens, we now need to eliminate everything but the third data element, the MIDI key-down velocities:

```
s%[0-9][0-9]*/[0-9][0-9]*/%%g
```

The *stats* Command

We can determine the range or variance of these velocity values by piping the output to the **stats** command. The **stats** command calculates basic statistical information for any input consisting of a column of numbers. A sample output from **stats** might appear as follows:

```
n:     124
total: 5700
mean:  45.9677
min:   9
max:   102
S.D.:  232.37
```

The value `n` indicates the total number of numerical values found in the input; the `total` specifies the sum of these numbers; the `mean` identifies the average; the `min` and `max` report the minimum and maximum values encountered, and the `S.D.` represents the standard deviation. The standard deviation provides a useful way of characterizing which performance has greater variability in key-down velocities.

Assuming that the above two stream-editing substitutions are kept in a file called `revise` we can compare the dynamic range for the two performances as follows:

```
extract -i '**MIDI' perform1 | grep -v ^= | humsed -r revise \
   | rid -GLId | stats
extract -i '**MIDI' perform2 | grep -v ^= | humsed -r revise \
   | rid -GLId | stats
```

The **extract** command has been added to ensure that we only process `**MIDI` data; the **grep** command ensures that possible barlines are eliminated, and the **rid** command eliminates comments and interpretations prior to passing the data to the **stats** command.

Eliminate Everything But ...

A common use for **humsed** is to eliminate signifiers that are not of interest. Stream editors like **sed** and **humsed** can be used to dramatically simplify some representation.

Did Monteverdi use equivalent numbers of sharps and flats? Or did he favor one accidental over the other? A simple way to determine this is to throw away everything but the sharps and flats. We can generate an inventory of just sharps and flats:

```
humsed 's/[^#-]//g' montev* | rid -GLId | sort | uniq -c
```

In some tasks, we might wish to transform a **kern-format file so that only pitch-related information is preserved:

```
humsed 's/[^a-gA-G#-]//g' inputfile
```

In extreme cases, we may wish to eliminate all Humdrum data from an input. The following command replaces all data tokens by null tokens:

```
humsed 's/[^      ][^      ]*/./g' inputfile
```

(That is, globally substitute all instances of the string not-a-tab followed by zero or more instances of not-a-tab characters, by a single period character.) This sort of command can be useful in generating a file that maintains the *structure* but not the *content* of some document. Incidentally, neither the **sed** nor the **humsed** commands support extended regular expressions, so we are not able to use the + metacharacter in the above substitution.

Deleting Data Records

Sometimes it is useful to delete entire data records rather than simply eliminating certain kinds of information. The d operation causes lines to be deleted. Normally, it is preceded by a regular expression that identifies which records should be eliminated. Deleting barlines can be done using the following command:

```
humsed /^=/d inputfile
```

Note that this is functionally equivalent to:

```
grep -v ^= inputfile
```

In the general case, **humsed /.../d** is preferable to **grep -v**. Remember that **humsed** only manipulates Humdrum data records; it never touches comments or interpretations. The **grep** command has no such restriction. Consider, for example, the following command to eliminate grace notes (acciaccaturas) from a **kern-format file.

```
humsed '/q/d' inputfile
```

By contrast, the command:

```
grep -v q inputfile
```

would also eliminate any comments or interpretation records containing the letter 'q'.

Suppose that we wanted to know whether a melody still evokes a certain key perception even if we eliminate all the tonic pitches. First we translate the representation to scale degree and assemble this file with the original **kern representation for the melody.

```
deg input > temp
assemble input temp | humsed '/1$/d' | midi | perform
```

Of course deleting all of the tonic notes will disrupt the original rhythm. An alternative is to replace all tonic pitches by rests:

```
deg input > temp
assemble input temp | humsed '/1$/s%[A-Ga-g#-]*%r%' | midi \
    | perform
```

Perhaps we might want to eliminate all the pitch information, and simply listen to the rhythmic structure of a work. That is, we might change all of the pitches in a work to a single pitch — in the following case, middle C:

```
humsed 's/[A-Ga-g#-]*/c/' | midi | perform
```

Adding Information

The substitute command can also be used to add information to points in a Humdrum input. For example, we might wish to add an explicit breath-mark (,) to the end of each phrase in a **kern-format input:

```
humsed s/}/},/g inputfile
```

Any occurrence of the ampersand (&) in the replacement string of a substitution is a standard stream-editing convention which means "the matched string." Suppose we want to add a tenuto mark to every quarter-note in a work. The following substitution seeks the number '4' followed by any character that is not a digit or period. This pattern is replaced by itself (&) followed by a tilde (~), the **kern signifier for a tenuto mark:

```
humsed s/4[^0-9.]/&~/g inputfile
```

Multiple Substitutions

Some tasks may require more than one substitution command. Multiple operations can be invoked by separating each operation by a semicolon. In the following example, we change all **kern quarter-notes to eighth-note durations:

```
humsed 's/4[A-Ga-g]/8&/g; s/84/8/g' inputfile
```

The first substitution finds strings that match the number '4' followed by an upper- or lower-case letter from A to G. The matched string is then output preceded by the number '8'. This operation will change all quarter notes and rests to eighty-fourth durations. The ensuing substitution opera-

tion changes '84' to '8' and so completes the transformation.

Switching Signifiers

In some situations, we will want to switch two or more signifiers — make all A's B's and all B's A's. These sorts of tasks require three substitutions and involve creating a unique temporary string. For example, the following command changes all `**kern` up-bows to down-bows and vice versa.

```
humsed 's/u/ABC/g; s/v/u/g; s/ABC/v/g' inputfile
```

The first substitution changes down-bows ('u') to the unique temporary string ABC. (In the `**kern` representation ABC is an illegal pitch representation, so it is bound to be a unique character string.) The second substitution changes up-bows (v) to down-bows. The third substitution changes occurrences of the temporary string ABC to up-bows.

Executing from a File

When several instructions are involved in stream editing, it can be inconvenient to type multiple operations on the command line. It is easier to place the editing instructions in a file, and use the **-f** option (with either **sed** or **humsed**) to execute from the file. Consider, for example, the task of rhythmic diminution, where the durations of notes are halved. We might create a file called `diminute` containing the following operations:

```
s/[0-9][0-9]\*/&XXX/g
s/64XXX/128/g
s/32XXX/64/g
s/16XXX/32/g
s/8XXX/16/g
s/4XXX/8/g
s/2XXX/4/g
s/1XXX/2/g
s/0XXX/1/g
```

Each substitution command is applied (in order) to every line or data record in the file. The first substitution adds the unique string XXX to every number. The ensuing substitutions transform these numbers to appropriate diminution values. We can execute these commands as follows:

```
humsed -f diminute inputfile
```

Writing to a File

A useful feature of **humsed** is the "write" or w operation. This operation causes a line to be written to the end of a specified file. Suppose, for example, we wanted to collect all seventh chords into a separate file called `sevenths`. With a `**harm`-format input, the appropriate command would be:

```
humsed '/7/w sevenths' inputfile.hrm
```

Each line containing the number 7 wll be written to a file named `sevenths`.

Similarly, we could copy all sonorities containing pauses to the file `pauses`.

```
humsed '/;/w pauses' inputfile
```

Of course there are other ways of achieving the same goal:

```
yank -m ';' 0 inputfile > pauses
```

Or even:

```
grep ';' inputfile | grep -v '^[!*]' > pauses
```

In some cases, a stream editor can be used to eliminate or modify data that will confound subsequent processing. For example, suppose we wanted to count the number of phrases that begin on the subdominant and the number of phrases that end on the subdominant. The **deg** command will allow us to identify subdominant pitches (via the number '4'). Since we would like to maintain the phrase indicators, we will avoid the **-x** option for **deg**. However, the **-x** option will pass *all* of the non-pitch related signifiers, including the duration data which encodes numbers. Hence, we will not be able to distinguish the subdominant ('4') pitch from a `**kern` quarter-note ('4'). The problem is resolved by first eliminating all of the duration information (numbers) from the original input:

```
humsed 's/[0-9.]//g' input.krn | deg | egrep -c '({.*4)|4.*{)'
humsed 's/[0-9.]//g' input.krn | deg | egrep -c '(}.*4)|4.*})'
```

In texts for vocal works, identify the number of notes per syllable.

```
extract -i '**kern' input | humsed 's/X//g' > tune
extract -i '**silbe' input | humsed 's/[a-zA-Z]*/X/' > lyrics
assemble tune lyrics | cleave -i '**kern,**silbe' -o '**new' \
   > combined
context -b X -o '[r=]' combined | rid -GLId | awk '{print NF}'
```

Identify the number of notes per word rather than per syllable.

```
extract -i '**kern' input > tune
extract -i '**silbe' input | humsed 's/^[^-].*[^-]$/BEGIN_END/;
s/-.*[^-]$/END/; s/^[^-].*-/BEGIN/' > lyrics
assemble tune lyrics | cleave -i '**kern,**silbe' -o '**new' \
   > combined
context -b BEGIN -e END -o '[r=]' combined | rid -GLId \
   | awk '{print NF}'
```

Reading a File as Input

Another useful feature is the **humsed** "read" or `r` operation. Whenever a leading regular expression is matched, a file is read in at that point. Suppose, for example, that we want to annotate a file with Humdrum comments identifying the presence of cadential 6-4 chords. First, we might create a file — `comment.6-4` — containing the following Humdrum comment:

```
!! A likely cadential 6-4 progression.
```

We can use the Humdrum **pattern** command (to be described in Chapter 21), as follows:

File `template`:

```
 .*
Ic
^\. *
=   *
V[^I]
```

Command:

```
pattern -f template inputfile > output
humsed 'cadential-64/r comment.6-4' output > commented.output
```

Reprise

The **sed** and **humsed** commands provide stream editors that can automatically edit a data stream. We've seen that multiple operations can be carried out, either from the command line or from a file containing editing instructions. It should be noted that the **sed** and **humsed** commands provide many more editing facilities than those discussed in this chapter. Some 25 operations are provided by **sed** and **humsed**. For example, segments of text can be stored in various buffers, the contents of these buffers modified, and the results placed anywhere in the output text. Markers can be set at particular points and conditional branch statements executed. Stream-editing scripts have been written to execute programs of considerable complexity. However, for most tasks, the simple substitute (**s**) and delete (**d**) operations are the most useful. For further information about stream editing using **sed**, refer to the book on **sed** and **awk** written by Dale Dougherty (listed in the bibliography).

Chapter 15

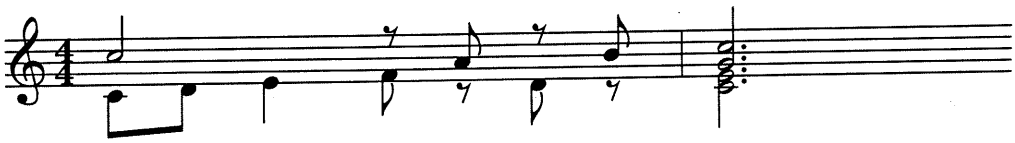
Harmonic Intervals

In Chapter 11 we examined Humdrum tools related to melodic pitch intervals. This chapter returns to the discussion of intervals by focussing on two tools pertaining to characterizing harmonic intervals: the **hint** and **ydelta** commands.

Types of Harmonic Intervals

Harmonic intervals identify pitch distances between nominally concurrently-sounding notes. Example 15.1 illustrates five types of harmonic intervals. The most obvious harmonic intervals are those that occur between pitches that have concurrent onsets. We might call these *explicit harmonic intervals.* An example of an explicit harmonic interval is the perfect octave in the first sonority of Example 15.1

Example 15.1 Types of harmonic intervals.



```
**kern    **kern
*M4/4     *M4/4
=1-       =1-
8c        2cc
8d        .
4e        .
8f        8r
8r        8a
8d        8r
8r        8b
=2        =2
2.c 2.e   2.g 2.cc
*-        *-
```

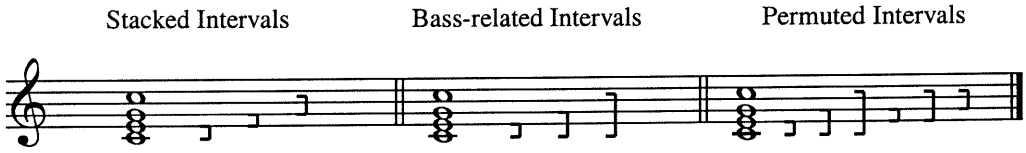
Less obvious harmonic intervals arise when tone onsets are not synchronous — yet both pitches are sustained simultaneously for a period of time. We might call these *passing harmonic intervals.*

Two examples are evident in Example 15.1: the minor seventh between the D and the sustained C, and the minor sixth between E and C. Most music theorists would argue that the minor sixth interval is more important than the minor seventh. Compared with the D, the following E has a longer duration and coincides with a stronger metrical position. Hence the interval of the minor sixth is likely to be more perceptually salient than the minor seventh interval. We might call these intervals *strong passing intervals* and *weak passing intervals* respectively.

Another type of harmonic interval may be deemed to occur even when there are no concurrently sounding pitches. In Example 15.1, the alternating F→A and D→B imply subdominant and dominant harmonic functions and so suggest the intervals of a major third followed by a major sixth. We might call these intervals *implicit harmonic intervals.* The harmonic status of the pitch sequence A→D is more contentious; many theorists would argue that no harmonic interval is implied either because of the weak→strong metric relationship or because the harmonic interval is less stereotypic of the given the cadential cliché. Once again, implicit harmonic intervals might be distinguished as weak or strong.

A further complication in identifying harmonic intervals arises when there are more than two pitches involved. In the final four-note chord of Example 15.1, what harmonic intervals are implied? Example 15.2 shows three possible break-downs of the harmonic relationships. *Stacked harmonic intervals* include only the intervals between successive pitches ordered from low to high. Four pitches thus generate three intervals. *Bass-related harmonic intervals* include all intervals calculated with respect to the lowest pitch. Four pitches again leads to three (different) intervals. This latter way of characterizing intervals is the most similar to figured bass notation. *Permuted harmonic intervals* include all intervals that can be calculated between all notes of a sonority. Four pitches leads to six intervals.

Example 15.2 Interpreting Interval Content in Chords



A final issue in characterizing harmonic intervals is whether or not the presence of unisons is important. In most applications (such as figured bass) the presence of unisons is unimportant. In other applications (such as tabulating instrument doubling in orchestration) the occurrence of unisons is noteworthy.

By way of summary, we have distinguished five types of harmonic intervals, and three measurement methods: explicit harmonic intervals, implicit harmonic intervals (both strong and weak), plus passing harmonic intervals (again both strong and weak varieties). Measurement methods include (1) stacked harmonic intervals, (2) bass-related harmonic intervals, and (3) permuted harmonic intervals.

As in the case of melodic intervals, harmonic intervals can be calculated according to a variety of units — including diatonic intervals, semitones, cents, frequency, and even cochlear coordinates. We will consider just two tools for calculating harmonic intervals: **hint** and **ydelta.**

Harmonic Intervals Using the *hint* Command

The Humdrum **hint** command calculates harmonic intervals for pitch-related representations such as `**kern`, `**pitch`, `**solfg`, and `**Tonh`. As in the case of the **mint** command, output intervals are expressed as a combination of diatonic interval size plus interval quality (such as 'perfect fourth' and 'minor ninth').

In the default operation, **hint** calculates only explicit harmonic intervals; for sonorities containing more than two pitches, only stacked harmonic intervals are calculated. The output from the **hint** command always consists of a single `**hint` spine. Any number of spines may be present in the input, but only pitch-related spines are processed. Given the default invocation, the output corresponding to Example 15.1 is as follows:

```
**hint
*M4/4
=1-
P8
-
-
-
-
-
-
=2
M3 m3 P4
*-
```

Notice that sonorities that contain only a single pitch result in the outputting of a hyphen (-). The hyphen indicates that pitched material is present, but there are no explicit harmonic intervals. Input records that contain no pitch tokens result in the outputting of a null token (.). If a single duplicated pitch is present, then the output will indicate a perfect unison (P1). Unisons can be suppressed from the output via the **-u** option for **hint**.

When more than two pitches are present in a sonority, *permuted harmonic intervals* can be calculated by invoking the **-a** option (i.e. *all* intervals). For example, with the **-a** option, the final chord in Example 15.1 would produce the following output:

```
**hint
*all
M3 P5 P8 m3 m6 P4
*-
```

Notice the presence of the `*all` tandem interpretation in the above output. This interpretation is added to the output in order to warn users that the representation should not be interpreted as stacked intervals.

Bass-related harmonic intervals can be calculated with the **-l** option. In this case, harmonic intervals are calculated with respect to the lowest pitch in the sonority. This option is helpful in determining figured bass. For the final chord in Example 12.1 the corresponding output would be

```
**hint
M3 P5 P8
*-
```

Two further options for **hint** allow the user to tailor how the intervals are represented. The **-c** option causes compound intervals such as a minor tenth (m10) to be output as non-compound equivalents (m3). This means that the interval of an octave (P8) will be rendered as a unison (P1). The **-d** option suppresses the outputting of interval qualities and results in only diatonic interval sizes being output. Again, this option is helpful in determining figured bass. The command:

```
hint -clud
```

will produce the following output for the final major chord in Example 12.1:

```
**hint
3 5
*-
```

Propagating Data Using the *ditto* Command

In the default operation, **hint** calculates intervals only between pitches that are explicitly present in an input data record. This means that passing intervals are not calculated.

In order to generate passing intervals, we will make use of the Humdrum **ditto** command. The **ditto** command replaces null tokens with the previous non-null data token in the same spine. Suppose we had an arbitrary input such as the following:

```
**flip   **flop
A        xyz
.        jkl
.        .
B        abc
.        .
C        .
.        .
*-       *-
```

The effect of **ditto** would be the following:

```
**flip   **flop
A        xyz
A        jkl
A        jkl
B        abc
B        abc
C        abc
C        abc
*-       *-
```

Each null token has been replaced by the preceding data token within the spine.

Consider the effect of **ditto** on the `**kern` data in Example 15.1:

```
ditto -p example15.1
```

The following output results:

```
**kern    **kern
*M4/4     *M4/4
=1-       =1-
8c        2cc
8d        (2cc)
4e        (2cc)
8f        8r
8r        8a
8d        8r
8r        8b
=2        =2
2.c 2.e   2.g 2.cc
*-        *-
```

Notice that the half-note C5 has been repeated. The **-p** option has caused each repetition of the data token to be placed in parentheses so they can be easily recognized. By using **ditto,** we have transformed previously passing intervals into explicit sonorities whose intervals can now be identified by **hint.** We can combine the two commands in a single pipeline:

```
ditto example15.1 | hint
```

The resulting output for Example 15.1 includes the two passing intervals (m7 and m6) in the first measure:

```
**hint
*M4/4
=1-
P8
m7
m6
-
-
-
-
=2
M3 m3 P4
*-
```

The **ditto**, command provides to additional options that are worthy of note: the **-s** and **-c** options. The **-s** option allows **ditto** to skip or ignore the presence of certain data records. Suppose, for example, that we had a barline in the midst of some null tokens:

```
A
.
=
.
```

Often, we would like to propagate certain data tokens *around* some other types of data tokens, so the result might be:

```
A
A
=
A
```

By providing **ditto** with a suitable regular expression, we can have the data token 'A' skip over the barline:

```
ditto -s ^=
```

Without this option, the final data token in the above example would be an equals-sign rather than the token 'A'.

The **-c** option for **ditto** allows the user to selectively identify which characters are propagated. For example, the following command will cause only the lower-case letters 'a' and 'b' to be propagated:

```
ditto -c ab
```

This feature allows users to replicate only certain kinds of data — such as pitches, durations, dynamic marks, etc.

As we will see in future chapters, the **ditto** command proves useful in a wide variety of situations apart from calculating intervals.

Using the *ditto* and *hint* Commands

Let's pause and consider some of the ways we might use the **ditto** and **hint** commands. First, let's determine if some input contains a particular interval. Are there any augmented sixth intervals in Bach's two-part inventions? The following commands look for explicit and passing sixths respectively. Notice the use of the **-c** option so octave equivalents will also be identified:

```
hint -c inventio* | grep A6
ditto -s ^= inventio* | hint -c | grep A6
```

Are there any diminished octave intervals between any two concurrent notes in any of Beethoven's piano sonatas?

```
ditto -s ^= sonatas* | hint -a | grep d8
```

In orchestral works, some pairs of instruments are more likely than others to double each other at

the unison or octave. What proportion of the intervals formed by the oboe and flute notes are doubled? Since we are looking for a proportion, we need to make two counts: the total number of (explicit) intervals formed by the oboe and flute, and the number of those intervals that are octave equivalents. (We will assume that there is only one oboe and one flute part in the file `Rimsky-K`:)

```
extract -i '*Ioboe,*Iflts' Rimsky-K | hint -c | rid -GLId \
   | grep -c P1
extract -i '*Ioboe,*Iflts' Rimsky-K | hint -c | rid -GLId \
   | grep -c [MmPAd]
```

The second **grep** counts the total number of intervals by looking for all of the interval qualities (major, minor, perfect, etc.)

Suppose we have extracted two horn parts from an orchestral score. Are octave intervals between the horns more likely to occur on the dominant pitch or the tonic pitch?

```
solfa horns > temp1
hint horns > temp2
assemble temp1 temp2 | grep -c ^do.*P8
assemble temp1 temp2 | grep -c ^so.*P8
```

Determining Implicit Harmonic Intervals

Recall that *implicit harmonic intervals* may be deemed to occur between tones that don't actually sound at the same time. This arises when one part has a rest while the other part is sounding. Note that if we could eliminate rest tokens, then we could use **ditto** to repeat previous pitch tokens in place of the rests and so generate implicit harmonic intervals.

The **humsed** command (described in Chapter 14) is well suited to this task. We want to transform any data token containing the letter 'r' to a null token. Consider the following substitution:

```
humsed 's/.*r.*/.     ./' example15.1
```

Unfortunately, this isn't quite right. The above substitution will find any data record containing the letter 'r' and transform the entire record to a single null data record. We need to address individual data tokens. In this example, rest tokens may be in two possible positions: the first token in the record or the last token in the record. We need two different regular expressions to address each of these conditions. First, a regular expression to identify rests in the first token:

```
/^[^    ]*r[^    ]*    /
```

(That is, the beginning of the record (^), followed by zero or more instances of any character other than a tab ([^]*), followed by the letter 'r', followed by zero or more instances of any character other than a tab ([^]*), followed by a tab.)

Similarly, our second regular expression identifies rests in the last token:

```
/    [^    ]*r[^    ]*$/
```

Now we can eliminate rest tokens using the following two substitution commands within a single invocation of **humsed**:

```
humsed 's/^[^  ]*r[^  ]*  /.  /; s/  [^  ]*r[^  ]*$/  ./' example15.1
```

The following output results:

```
**kern    **kern
*M4/4     *M4/4
=1-       =1-
8c        2cc
8d        .
4e        .
8f        .
.         8a
8d        .
.         8b
=2        =2
2.c 2.e   2.g 2.cc
*-        *-
```

If we now apply **ditto** and recalculate the intervals, the resulting output will identify some implicit intervals as well:

```
humsed 's/^[^  ]*r[^  ]*  /.  /; s/  [^  ]*r[^  ]*$/  ./' example15.1 \
   | ditto -p
```

Below we see the output assembled with the output from the corresponding **hint** command:

```
**kern    **kern   **hint
*M4/4     *M4/4    *M4/4
=1-       =1-      =1-
8c        2cc      P8
8d        (2cc)    m7
4e        (2cc)    m6
8f        (2cc)    P5
(8f)      8a       M3
8d        (8a)     P5
(8d)      8b       M6
=2        =2       =2
2.c 2.e   2.g 2.cc M3 m3 P4
*-        *-       *-
```

The *ydelta* Command

Often it is useful to represent intervals by the number of semitones (or some other numerical value). We might begin by using the **semits** command to translate Example 15.1 to a `**semits` representation.

```
semits example15.1
```

The resulting output would be as follows:

```
**semits    **semits
*M4/4       *M4/4
=1-         =1-
0           12
2           .
4           .
5           r
r           9
2           r
r           11
=2          =2
0 4         7 12
*-          *-
```

Numerical differences for values on a single data record can be computed using the **ydelta** command. The **ydelta** command is comparable to **xdelta** however, numerical differences are calculated between simultaneous numerical values (delta-*y*) rather than between successive numerical values (delta-*x*).

Like the **hint** command, **ydelta** always outputs a single spine. The user must specify which input spines are to be processed using the **-i** option. In the following command, only **semits input is to be processed:

```
semits example15.1 | ydelta -s = -i '**semits'
```

The **-s** option allows the user to identify data records to be *skipped* while processing. In this case, the regular expression '=' is used to identify barlines, so measure numbers will be excluded when processing the numerical data.

The above command yields the following output:

```
**Ysemits
*
=1-
12
.
.
.
.
.
.
=2
4 7 12
*-
```

Notice that **ydelta** prepends the upper-case letter 'Y' to the given input interpretation. All output values are calculated with respect to the lowest value in the current data record. Hence, the '4 7

12 in the last data record means that there are pitches 4 semitones above the lowest note, 7 semitones above the lowest note, and 12 semitones above the lowest note. (If necessary, the lowest or offset value for each record can be output in square brackets using the **-o** option.)

Like the **hint** command, **ydelta** calculates numerical intervals only when more than one value is present on a given input data record. As in the case of **hint,** we might use the **ditto** command to propagate pitch values — replacing all the null data tokens. A suitable command would be:

```
semits example15.1 | ditto -s = | ydelta -s = -i '**semits'
```

The resulting output would be:

```
**Ysemits
*
=1-
12
10
8
.
.
.
.
=2
4 7 12
*-
```

More Examples Using the *ydelta* Command

What is the average semitone distance between the cantus and altus voices in Lassus motets? We can answer this question by first extracting the appropriate voices and translating to a semitone representation.

```
extract -f 1,2 motet* | semits > temp1
```

Using **ditto** we can expand the pitched material so that concurrently-sounding tones will generate explicit intervals. We then use **ydelta** to calculate the actual semitone interval distances. The **rid** command can be used to eliminate non-data records, and the **grep -v** command can be used to further eliminate barlines. Finally, we can calculate the mean interval distance using the **stats** command:

```
ditto -s = temp1 | ydelta -s = -i '**semits' | rid -GLId \
   | grep -v = | stats
```

Suppose we have a two-part input. Are there tritone intervals (explicit and passing) that are not spelled as either an augmented fourth or diminished fifth? We can answer this question by using both **hint** and **ydelta** and a suitable sequence of **grep** commands. The **ditto** command is used to ensure that both explicit and passing intervals are generated.

```
ditto -s = 2part | semits | ydelta -s = -i '**semits' > temp1
ditto -s = 2part | hint > temp2
assemble temp1 temp2 | rid -GLId | grep ^6 | grep -v A4 \
    | grep -v d5
```

Notice the use of **grep -v** to first exclude any records that match an augmented fourth, and then to exclude any remaining records that match a diminished fifth.

Reprise

Harmonic intervals can be measured in a variety of ways. They can be characterized as diatonic qualities such as minor sevenths or augmented sixths. They can be measured in terms of semitone distance — or even in cents or hertz (frequency difference). Only the diatonic size may be of interest (e.g., "a fifth"), and compound intervals (e.g., major tenth) can be expressed by their non-compound equivalents (major third).

At least five types of harmonic intervals can be distinguished including *explicit harmonic intervals*, *strong passing intervals*, *weak passing intervals*, *strong implicit harmonic intervals* and *weak implicit harmonic intervals*. In addition, we distinguished three different ways of characterizing harmonic intervals: *stacked harmonic intervals*, *bass-related harmonic intervals* and *permuted harmonic intervals*.

In this chapter we have seen how to use the **hint** command to calculate these various kinds of intervals. We have also seen how **ydelta** can be used to measure purely numerical distances between concurrent values.

Chapter 16

The Shell (II)

In Chapter 8 we introduced some of the shell special characters. By way of review, we learned that the shell interprets the octothorpe (#) as the beginning of a comment. By itself, the asterisk (*) is "expanded" by the shell to the names of all files in the current directory. When linked with other characters, such as `A*` or `*B`, the shell expands the expression to the names of all files beginning with A or ending with B. The greather-than sign (>) directs the output to a named file. The vertical bar or pipe (|) allows the output of one command to be directed to the input of the following command. When placed at the end of a command line, the ampersand (&) causes the shell to execute the command as a background process, and immediately returns a prompt so the user can execute other commands. The semicolon (;) indicates the end of a command; this allows more than one command to be placed on a single line. The backslash (\) escapes the special meaning of the immediately following character so it is treated literally. The single quote or apostrophe (') can escape the special meaning of all characters up to the appearance of a matching single quote.

In this chapter we will continue to describe shell special characters and identify their functions. In addition, we will learn about the shell *alias* function.

Shell Special Characters

The remaining special shell characters include the following: the dollars sign ($), the greve (`), the less-than sign (<), the question mark (?), and the double quote ("). We'll consider the function of each of these characters in turn.

Shell Variables

Like any programming language, the shell allows information to be stored and retrieved through shell *variables*. Variables can be given all sorts of names, such as `value`, `Meter`, `A34x` and `BARLINE3`. In order to retrieve information from a variable, the variable name is preceded by a dollars sign. For example, the string `$VARIABLE` means "the current value of the variable named `VARIABLE`. Suppose you had a file named $FILE in the current directory ($FILE is a legitimate filename on UNIX systems). If you type:

```
sort $FILE
```

The shell will assume that there is a variable named FILE, and retrieve its contents. Since the contents are likely to be empty, the above command is identical to typing:

```
sort
```

In order to sort the file named $FILE, the dollars sign would need to be escaped:

```
sort \$FILE
```

Depending on the type of shell, variables can be assigned numerical or string values in various ways. For most shells, variables can be assigned using the equals sign (with no intervening spaces). For example, the integer 7 can be assigned to the variable X as follows:

```
X=7
```

Or the string "hello" can be assigned to a variable by placing the string in quotation marks:

```
X="hello"
```

Single quotation marks can also be used:

```
X='hello'
```

If you had a file named `hello` in the current directory, and if the varialbe `X` had been assigned as above, then the following command would sort this file:

```
sort $X
```

The Shell Greve

It is often useful to be able to save the results of some operation in a shell variable. Suppose for example, that we want to sort a file containing the word `zebra`. But we're not certain what file (or files) contain this word. Manually, we would need to carry out two operations. First we would search for any file(s) containing the word:

```
grep -l zebra *
```

We might find that the word "zebra" appears in the files `animals` and `mammals`. Having determine what files to sort, now we would actually carry out the appropriate sort command:

```
sort animals mammals
```

If we found that word "zebra" occurred in 50 files, then typing the appropriate sort command would require a lot of typing. Alternatively, we could use a shell variable to store the results of the first command, and then retrieve the filenames in the second command. For this, we must use the greve character (`). UNIX shells will execute whatever command(s) appear between two greve characters; the result of the operation can then be treated as a string which may be assigned to a shell variable or used in some other way. In the following commands, the filenames produced by

the **grep** command are assigned to a shell variable named FILES. In the subsequent command a dollars sign instructs the shell to retrieve the contents of this variable:

```
FILES=`grep -l zebra *`
sort $FILES
```

Alternatively, we can avoid the FILES variable altogether, and execute the following command:

```
sort `grep -l zebra *`
```

The shell interprets the above command as follows: First it recognizes the presence of the command delineated by greves. This command is executed *before* the **sort** command. The **grep -l** command will generate as output a string of filenames. This output will replace the material delineated by greves. Finally, the **sort** command will be executed — using the filenames generated by the **grep** command.

This command structure is useful in a variety of circumstances. For example, suppose we wanted to identify any encoded works that are composed by Josquin and are also in triple meter:

```
grep -l '!!!COM: Josquin' `grep -l '!!!AMT:.*triple' *`
```

Here we have imbedded one **grep** "inside" another. Remember that the command delineated by the greve is executed first. In this case, we begin by searching all of the files in the current directory for an AMT reference record containing the keyword "triple." The **-l** option causes the output to consist of only filenames. Then the second **grep** is executed. It looks for files that contain a COM reference record containing the keyword "Josquin." But this second **grep** only searches those filenames passed to it by the first **grep**. In other words, the composer search is restricted to only those files that have a triple meter designation.

Consider another way of using the greve structure. Suppose we have a file named opus16. We would like to know what other works contain the same instrumentation as opus16, but we've forgotten what the precise instrumentation is. We can first seach opus16 for the instrumentation data (encoded in the AIN: reference record), and then search for this information in all files in the current directory. This task can be carried out using a single command line:

```
grep -l `grep '!!!AIN:' opus16` *
```

In this example, the imbedded command provides the regular expression rather than the files to be searched.

Single Quotes, Double Quotes

In Chapter 8 we learned that single quotation marks can be used to escape the special meanings of reserved shell characters — such as * and $. Double quotation marks (") have a similar effect with one important exception. The dollars sign continues to retain its special meaning inside double quotes.

The UNIX **echo** command causes information to be printed or displayed. Consider the following

three commands:

```
echo $A
echo "$A"
echo '$A'
```

In the first and second commands, the shell looks for a variable named A and attempts to echo the contents of this variable on the display. Unless A happens to be a defined shell variable, only an empty line will be displayed. In the third command, the string $A is treated literally, and is echoed back to the display. There are circumstances where the double quotes are more useful, but for most casual users, the single quotes provide the best means for disengaging the meanings of special characters.

Using Shell Variables

Let's consider an example where shell variables prove to be useful in Humdrum processing. Suppose for some score that we want to change the stem-directions in measures 34 through 38 from up-stems to down-stems. First, we need to establish the line number corresponding to the beginning of measure 34 and the line number corresponding to the end of measure 38 (i.e. beginning of measure 39). In the following script, **grep** is used to assign these line numbers to the shell variables $A and $B.

```
A=`grep -n ^=34`
B=`grep -n ^=39`
```

Now we can construct an appropriate **humsed** command. Recall that each substitute (s) command in **humsed** can be preceded by a range indication. In the following command, the $A and $B variables convey the appropriate range to each substitution. This means that the substitutions are limited to the line numbers ranging between $A and $B.

```
humsed "$A,$Bs/\/XXX/g; $A,$Bs//\/g; $A,$Bs/XXX//g" inputfile
```

Notice that we have used double quotes (") rather than single quotes. The quotation marks are necessary to pass all three substitutions as an argument to **humsed**. Using singe quotes, however, would have caused $A and $B to be treated as literal strings rather than shell variables.

Aliases

An alias is an alternative name for something. The shell provides a way of defining aliases, and these aliases can prove very convenient.

Consider, by way of example, the following common pipeline:

```
sort inputfile | uniq -c | sort -n
```

In Chapter 17 we will see that this is a useful way for generating inventories. Typically, this sequence occurs at the end of a pipeline where some preliminary processing has taken place, such

as:

```
timebase -t 8 input | ditto | hint | rid -GLI \
    | sort | uniq -c | sort -n
```

Since the construction `sort | uniq -c | sort -n` is so common, we might want to define an alias for it. To do so, we simply execute the **alias** command. In this case, we've defined a new command called `inventory`:

```
alias inventory="sort | uniq -c | sort -n"
```

Having defined this alias, we can now make use of it. Any time we type the word `inventory`, the shell will expand it to "`sort | uniq -c | sort -n`". The above command can be shortened as follows:

```
timebase -t 8 input | ditto | hint | rid -GLI | inventory
```

Another common task is eliminating barlines. Frequently, we need to use the construction:

```
grep -v ^=
```

Actually, this is not the most prudent construction. Depending on the spines present in a document, sometimes barlines will be mixed with null tokens in other spines that do not encode explicit barlines. E.g.

```
.   =23   =23   .   .   =23
```

A more careful way of eliminating barlines would use the following regular expression:

```
egrep -v '^(\.     )*='
```

That is, eliminate all lines that either begin with an equals-sign, or have one or more leading null tokens followed by a token with a leading equals-sign. Since this is somewhat complicated to remember, we might alias it. In the following command, we have created a new command called `nobarlines`:

```
alias nobarlines='egrep -v '^(\.     )*='
```

In Humdrum, a good use of aliases is to define commonly used regular expressions. Consider the regular expression used to define tandem interpretations that encode meter signatures. Here we are searching for an asterisk at the beginning of a line, followed by the upper-case letter 'M' followed by a digit, followed by zero or more digits, followed by a slash, followed by a digit:

```
grep '^\*M[0-9][0-9]*/[0-9]' inputfile
```

Actually, this regular expression will fail to find any meter signature that is not in the first spine. A more circumspect regular expression will include the possibility of a leading tab:

```
grep '   *\*M[0-9][0-9]*/[0-9]' inputfile
```

Since this is a cumbersome regular expression, it can help to provide an alias. Here we have aliased the regular expression to the name `metersig`:

```
alias metersig="' *\*M[0-9][0-9]*/[0-9]'"
```

Now we can search for meter signatures as follows:

```
grep metersig inputfile
```

Reprise

In this chapter we have discussed how the shell interprets the dollars sign ($), the greve (‘), and the double quote ("). When followed by printable characters, the dollars sign is interpreted as designating the value of a shell variable. Any command enclosed between two greve characters is executed by the shell first, and the returned output of the command is available as an input parameter to some other command. Like single quotes, double quotes can be used to escape special shell characters; however, an important difference is that the dollars-sign retains its special meaning within the double quotes. This allows shell variables to be embedded into text strings.

We have also learned that the shell **alias** command can be used to provide a convenient short-hand or way of abbreviating a complex pipeline or regular expression into a single user-defined keyword.

Chapter 17

Creating Inventories

Many research problems can be addressed by building an *inventory* — that is, identifying the number of occurrences of various types of data. Questions such as the following all pertain to the generation of inventories:

- Does Liszt use a greater variety of harmonies than Chopin?
- What is the most frequently used dynamic marking in Beethoven, and how does Beethoven's practice compare with that of Brahms?
- Are flats more common than sharps in Monteverdi?
- Did Bartók's preferred articulation marks change over his lifetime?
- Is there a tendency to use the subdominant pitch less often in pop melodies than in (say) French chanson?
- How frequent are light-related words such as "lumen" or "lumine" in the different monastic offices for Thomas of Canterbury?
- Is it true that 90 percent of the notes in a given work by Bach use just two durations (such as eighths and sixteenths, or eighths and quarters)?
- What is the most common instrumental combination for orchestral works by Musorgsky?

At the end of this chapter we will show the Humdrum commands needed to answer each of the above questions.

The above questions are all variations of one of the following forms:

> How many different types of _____ are there?
> What is the most/least common _____?
> What is the frequency of occurrence for various _____s?

In some cases, we're asked to compare two or more repertories when answering one of these basic questions.

For illustration purposes, consider the case of a Humdrum file named `alpha` containing the following simple input:

```
**alpha
A
B
A
A
C
B
*-
```

It doesn't matter what the data represent. The "A", "B", and "C" might signify different articulation marks, chords, harmonic intervals, or instrumental configurations. Whatever is represented, the process of generating an inventory is the same. Ultimately, we'd like to produce a simple distribution that indicates:

```
3 occurrences of "A"
2 occurrences of "B"
1 occurrence  of "C"
```

Filter, Sort, Count

Building an inventory is a three-step process. First we need to *filter* the input so only the data of interest is present. Second we need to *sort* like-with-like. And third we need to *count* the number of occurrences of each type of data token.

Let's begin by discussing the second process. In Chapter 3 we saw how the UNIX **sort** command will rearrange lines of data so that they are in alphabetical/numerical order. The command:

```
sort alpha > sorted.alpha
```

will sort the file `alpha` and place the results in a file named `sorted.alpha`. The file `sorted.alpha` will contain the following:

```
**alpha
*-
A
A
A
B
B
C
```

Notice that the asterisk is treated as alphabetically prior to the the letter 'A', so all the Humdrum interpretation records have been moved to the beginning of the output. Notice also that all of the lines beginning with the letter 'A' are now collected on successive lines in the output. Similarly, the 'B's have been rearranged on successive lines.

The third step in generating an inventory is to count the number of occurrences of each unique data token. The **uniq** command described in Chapter 3 will eliminate successive duplicate lines. For example, if we type:

```
uniq sorted.alpha
```

The output will be as follows:

```
**alpha
*-
A
B
C
```

Notice that repetitions of the data "A" and "B" have disappeared. The simple **uniq** command is useful for telling us *how many different things* there are in an input. For example, the above output identifies just five different records — and three different types of data records.

Recall that the **-c** option for **uniq** will cause a 'count' to be prepended to each output line. The command:

```
uniq -c sorted.alpha > unique.alpha
```

will produce the following output:

```
1   **alpha
1   *-
3   A
2   B
1   C
```

The prepended counts tell us that 'A' occurs three times, 'B' occurs twice, and all other records occur just once.

In the above output, `**alpha`, and `*-` are Humdrum interpretations rather than data, so we probably don't want them to appear in our inventory. If our file had contained comments, or null data records, then these would have also appeared in our output, although we are not likely to be interested in them. This leads us to what is normally the first step in generating an inventory — *filtering* the input in order to eliminate records that we'd prefer to omit from our final output.

Filtering Data with the *rid* Command

As we saw in Chapter 13, the **rid** command can be used to eliminate various classes of Humdrum records. For example, **rid -G** eliminates all global comments; **rid -D** eliminates all data records, etc. The option combination **-GLId** is very common with **rid** since only data records are retained in the output. That is, eliminating all global and local comments, omitting all interpretations, and deleting all null data records will result in an output consisting only of non-null data records.

Returning to our `**alpha` data, we can eliminate everything but data records as follows:

```
rid -GLId alpha > filtered.alpha
```

By way of summary, generating an inventory is a three-step process. First we *filter* the input so only the data of interest is present. Typically, this means using the **rid** command with one or more options to eliminate comments, interpretations, and perhaps null data records. Second we *sort* the

data using the **sort** command so that identical records are amalgamated as neighbors. Finally, we use the **uniq -c** to *count* the number of occurrences of each type of data token. All three steps can be amalgamated into a single pipeline:

```
rid -GLId alpha | sort | uniq -c > inventory.alpha
```

Notice that the inventory will pertain to whatever data was provided in the original input. We've been using the abstract data "A", "B", and "C". However, this data might represent any type of discrete data, such as Latin text, piano fingerings, or dance steps.

Inventories for Multi-spine Inputs

In the above example, we assumed that the input consists of a single Humdrum spine (i.e. a single column of data). However, Humdrum files can have any number of spines, and each spine might represent radically different types of data. For example, the following file (named `alphabet`) contains two spines, one with "alpha" data, and the second with "bet" data. These data types might represent melodic intervals and fingering information, or dynamic markings and stem-directions, or whatever.

```
**alpha   **bet
A         $50
B         $50
A         $50
A         $200
C         $50
B         $50
*-        *-
```

If we apply our above inventory-generating commands for the file "alphabet," the result will be as follows:

```
1   A   $200
2   A   $50
2   B   $50
1   C   $50
```

Notice that the inventory is based on *entire records* containing both "alpha" and "bet" data. This is the reason why the alpha-bet data-pair "`A  $50`" is considered different from alpha-bet data "`A  $200`". Depending on the user's goal, this may or may not be the most appropriate output.

A situation where this approach might be desired arises when we are counting the number of different spellings of chords (e.g., how many different sonorous arrangements are there?). If **alpha and **bet represent pitches in two concurrent voices, then it may be important to have both concurrent data tokens participating in the inventory.

In other circumstances, we may not want this. For example, if we are interested only in alpha-related data, we need to eliminate the irrelevant **bet data so it won't interfere. This can be done using the Humdrum **extract** command.

For example, we can create an inventory of just the `**bet` data:

```
extract -i '**bet' alphabet | rid -GLId | sort | uniq -c \
   > inventory.bet
```

The resulting `inventory.bet` file will contain:

```
1   $200
5   $50
```

— meaning 5 occurrences of the data "$50" and 1 occurrence of "$200".

Sometimes it is useful to create an aggregate inventory of the data in each separate spine. In such cases, we will need to use **extract** several times so that each spine is placed in a separate file:

```
extract -i '**alpha' alphabet > justalpha
extract -i '**bet' alphabet > justbet
```

The **cat** command can then be used to concatenate the files end-to-end so they form a single column of data. With each data token of interest is on its own line, we can generate the appropriate inventory:

```
cat justalpha justbet | rid -GLId | sort | uniq -c
```

Sorting By Frequency of Occurrence

When the output inventory list is short, it is easy to identify which records are the most common and which records are the least common. Frequently inventory lists will contain dozens or hundreds of items so it may be more difficult to scan through the output to find the most frequent or least frequent occurrences. For such long outputs, it might be more convenient to produce an output sorted according to frequency of occurrence. Notice that each output record from **uniq -c** begins with a number, and so the output is ideally suited for numerical sorting. We've already learned that the **sort** command rearranges input records in alphabetic/numeric order.

If we type

```
sort inventory.alpha
```

The output will be as follows:

```
1   C
2   B
3   A
```

Now the output is sorted so that the least frequent occurrences are at the beginning, and the most frequent occurrences are at the end of the output. Incidentally, **sort** has a **-r** option that causes the output to be sorted in reverse order. If we use **sort -r**, then the most common occurrences will be placed at the beginning of the output:

```
sort -r inventory.alpha
```

produces the following output:

```
3   A
2   B
1   C
```

Once again, we can amalgamate all of the required commands into a single pipeline. The following pipeline produces an inventory for any type of Humdrum input, sorted from the most common to the least common data:

```
rid -GLId alpha | sort | uniq -c | sort -r > inventory.alpha
```

Counting with the *wc* Command

In other circumstances, it may be helpful to determine the proportion or percentage values rather than the actual numerical count. This can be calculated by dividing each of the inventory count numbers by the total number of data records processed. A convenient way to count records is via the UNIX **wc** (word count) command. The **wc** command provides three options. With the **-c** option, **wc** counts the number of characters in an input. With the **-w** option, **wc** counts the number of words in an input. A "word" is defined as any sequence of characters delineated by white space, such as spaces, tabs or new lines. With the **-l** option, **wc** counts the nmber of lines or records in the input.

We can count the total number of non-null data records in a Humdrum input using the following pipeline:

```
rid -GLId alpha | wc -l
```

This will give us the total number of items in our inventory. Simple division will generate the percentages for each type of data record.

Suppose, for example, that the total number of data records was determined to be 874. Using the UNIX **awk** command will allow us to easily generate the percentages for each data type via the command:

```
awk '{print $1/847*100 "\t" $2}' inventory.alpha
```

This will create a two-column output. The first column will indicate the percentage of occurrence, and the second column will identify the corresponding type of data.

Excluding or Seeking Rare Events

Recall from Chapter 3 that the **uniq** command provides other options (besides the **-c** option). The **-d** option causes **uniq** to output *only* those records that are duplicated. In other words, records that occur only once are eliminated from the input. This option can be useful when there are a lot of single-occurrence data tokens and you are only interested in those data records that occur more frequently.

By contrast, the **uniq -u** option causes *only* those records that are unique (occur only once) to be

output. This option can be useful when looking for rare circumstances in our data.

```
rid -GLId alpha | sort | uniq -u    (output only the rare events)
rid -GLId alpha | sort | uniq -d    (eliminate all the rare events)
```

Transforming and Editing Inventory Data

Notice that two data records must be identical in order for them to be considered "the same" by **sort** and **uniq**. This means that records such as the following are considered entirely different:

```
ABC
abc
Abc
"ABC"
ABC.
CBA
```

Remember that step #1 in generating inventories requires that we filter the data so only the data of interest is passed to **sort** and **uniq**. This means we must be careful about the state of the input. Depending on your goal, we will either want to *translate* the input to some other more appropriate representation, or *edit* the existing representation in order to discard or transform otherwise confounding data.

Translating data involves changing from one type of information to another — that is, changing the exclusive interpretations. For example, if we want to produce an inventory of melodic intervals, then we might use the **mint** or **xdelta** commands to generate a suitable representation. Alternatively, we might want to generate an inventory of scale degrees using the **deg** or **solfa** commands.

Instead of translating our data, we might wish to edit the data using the **sed** or **humsed** stream editors. Suppose we had a file (named "notes") consisting of pitch information, and we wanted to create an inventory of the diatonic pitch-letter names. Our input might look like this:

```
**notes
A
B
B
D
F#
D#
E
*-
```

Without modification, our inventory would appear as follows:

```
1   A
2   B
1   D
1   D#
1   E
1   F#
```

But this inventory distinguishes D-sharp from D-natural — which is not what we want. The answer is to filter our input so that the sharps are removed.

Adding the appropriate **humsed** command to our pipe:

```
humsed 's/#//' notes | rid -GLId | sort | uniq -c
```

— will produce the following output:

```
1   A
2   B
2   D
1   E
1   F
```

Further Examples

Given your current background, you should now be able to generate inventories to answer a wide variety of questions. You should now understand how the commands given below can be used to solve the question posed:

Does Liszt use a greater variety of harmonies than Chopin?

```
extract -i '**harm' liszt* | rid -GLId | sort | uniq | wc -l
extract -i '**harm' chopin* | rid -GLId | sort | uniq | wc -l
```

What is the most frequently used dynamic marking in Beethoven, and how does Beethoven's practice compare with that of Brahms?

```
extract -i '**dynam' beeth* | rid -GLId | sort | uniq -c \
   | sort -r | head -1
extract -i '**dynam' brahm* | rid -GLId | sort | uniq -c \
   | sort -r | head -1
```

Are flats more common than sharps in Monteverdi? Let's presume that the input is monophonic `**kern` data.

```
humsed 's/[^#-]//g' montev* | rid -GLId | sort | uniq -c
```

Did Bartók's preferred articulation marks change over his lifetime? Assume that copies of early and late works have been concatenated to the files `early` and `late`. The **humsed** command here eliminates all data with the exception of `**kern` articulation marks. (See Chapter 6 for details on

**kern articulation marks.)

```
extract -i '**kern' early | humsed 's/[^"'`~^:I]//g' \
    | rid -GLId | sort | uniq -c
extract -i '**kern' late | humsed 's/[^"'`~^:I]//g' \
    | rid -GLId | sort | uniq -c
```

Is there a tendency to use the subdominant pitch less often in pop melodies than in (say) French chanson? Once again assume that the inputs are monophonic.

```
deg -t pop* | grep -c '4'
deg -t chanson* | grep -c '4'
```

How frequent are light-related words such as "lumen" or "lumine" in the different monastic offices for Thomas of Canterbury? Familiarity with regular expressions helps:

```
extract -i '**text' office* | egrep -ic 'lum.+n[e]*$'
```

Is it true that 90 percent of the notes in a given work by Bach use just two durations (such as eighths and sixteenths, or eighths and quarters)?

```
humsed 's/[^0-9.]//g' bach | rid -GLId | sort | uniq -c
```

(Repeat the above command for each work and inspect the results.)

What is the most common instrumental combination for sonorities by Mussorgsky?

This problem is addressed in Chapter 36.

Reprise

In this chapter we have discussed how to answer questions that involve the creation of inventories. Creating an inventory typically entails *filtering* some data so only the information of interest is preserved, *sorting* the data so that like data are amalgamated, and then *counting* each instance of each data type.

In later chapters we will see how classifying data, identifying musical contexts, and marking occurrences of patterns can be used to significantly enhance the inventory-building tools described in this chapter.

Chapter 18

Fingers, Footsteps and Frets

Throughout this book, our examples have tended to rely on just a handful of Humdrum representations — `**kern` in particular. Of course the Humdrum syntax provides opportunities for an unlimited number of representations. In this chapter, we will consider some less common representations. Most of the chapter will deal with the `**fret` representation — a pre-defined Humdrum representation for fretted instrument tablatures. However, we will begin with a grab-bag of unorthodox representations.

Heart Beats and Other Esoterica

The Humdrum syntax provides a framework within which different symbol systems can be defined. Each symbol system or representation scheme is denoted by a unique exclusive interpretation. There is no restriction on the number of schemes that can be used or created by the user. Each time you create a new exclusive interpretation, all of the alphanumeric characters can be redefined. A representation scheme springs into existence simply by encoding some data.

In research activities, it is common to create representation schemes for a specific task. A user might define a spine that represents the heart-rate (in beats per minute) of a music listener:

```
**cardio
76
76
74
73
73
*-
```

A scheme might be created to represent different types of contrapuntal motion:

```
**motion
similar
contrary
parallel
oblique
*-
```

Chords might be classified — using words:

```
**chords
minor
.
augmented
major
*-
```

Or using abbreviations:

```
**chords
m
.
A
M
*-
```

Fingerings might be represented. Each hand may have a separate spine:

```
**finger    **finger
*left       *right
.           1
1 5         2
.           3
.           5
1 5         2 5
*-          *-
```

Or the hands might be combined in a single spine:

```
**finger
R1
L1 L5 R2
R3
5
L1 L5 R2 R5
*-
```

Time-scales might be large:

```
**Periods
Medieval
Renaissance
Baroque
Classical
Romantic
*-
```

Or miniscule:

```
**milliseconds
0.03
0.8
23.2
31.6
*-
```

A user might define a highly refined special-purpose representation. For example, the following scheme is fashioned after the Benesh dance notation:

```
!! Kellom Tomlinson's Gavot of 1720.
!! Transcribed from Feuillet's notation.
**kern          **Benesh
*MM120          *MM120
*M2/2           *M2
*e:             *
!! First Couplet
!               ! Half Coupee
4.gg             | |u| | |
.               %|+| | | |
.                |=|v| | |
.               (| | | | |
.               (|-| | | |{
8ff#             |_| | | |2
=1              -----------
2ee             m| | | | |}
!               ! Bound
.               _|^| | | |{
.               (|+| | | |
4.b             -| | | | |
.               (|+| | | |}
.               _| | | | |{
8a              _|+|+| | |
=2              -----------
!               ! Bouree
4g               | | | | |
.                |=| | | |
4e               |=| | | |}
.               _|=| | | |
4g               |o| | | |
!               ! Bouree
4a              _|+|+| | |
=3              -----------
*-              *-
```

Depending on the task, user-defined schemes can be either carefully designed, or "throw-away" concoctions created for momentary purposes. The *Humdrum Reference Manual* provides detailed

advice on how to go about designing special-purpose Humdrum representations.

For the remainder of this chapter we will examine a pre-defined representation scheme for fretted instrument tablatures. Although not all users will be interested in the `**fret` representation, it provides an instructive contrast to the score- and MIDIbased representations that we have relied on for most of the examples in this book.

The ***fret* Representation

The `**fret` representation is a pre-defined Humdrum scheme that provides a comprehensive system for representing performance aspects for fretted instruments. The `**fret` scheme is suitable for representing tablature information for most fretted instruments, such as various guitars, lute, mandore, theorbo, chitarrone, mandoline, banjo, dulcimer, and even viols. The `**fret` interpretation is not limited to equal-temperament tuning, and so can be used to represent non-Western fretted instruments, such as the *oud* and the *sitar*.

The `**fret` representation is performance-oriented rather than notationally-oriented. Thus `**fret` is not suitable for distinguishing different visual renderings — such as differences between traditional French or German lute tablatures. Some other Humdrum representation should be used if the user's goal is to distinguish different forms of visual signifiers.

The basic pitches produced by fretted instruments depend on three factors: (1) the relative tuning of the strings with respect to each other, (2) the absolute overall tuning of the instrument, and (3) the position of the frets. Three tandem interpretations allow the user to specify each of these aspects.

The absolute tuning of an instrument is indicated by encoding the pitch of the lowest string using the `*AT:` tandem interpretation. For the common six-string guitar, the lowest pitch is normally tuned to E2, and so would be encoded with the following tandem interpretation:

```
*AT:E2
```

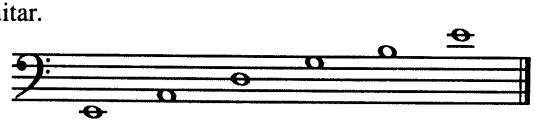
The `*AT:` interpretation makes use of `**pitch`-type pitch designations and may also include cents deviation. For example, an instrument tuned 45 cents sharp might be represented as `*AT:E2+45`. Encoding the absolute tuning is optional with `**fret`; when absent, a default tuning of E2 will be assumed by various processing tools.

A second tandem interpretation (`*RT:`) specifies the relative tuning as well as the number and arrangement of strings. Some instruments pair strings together in close physical proximity so that two strings are treated by the performer as a single virtual "string." Such paired strings are referred to as *courses*. For example, the 12-string guitar is constructed using 6 courses, and is played much like a 6-string guitar — except that two strings sound together, rather than a single string.

The `*RT:` tandem interpretation encodes the relative tuning of each string by specifying the number of semitones above the lowest string — where each course is delineated by a colon (:). In Example 18.1 three sample tunings are shown. Example (a) defines the most common relative tuning for the six-string guitar. Successive strings are tuned 0, 5, 10, 15, 19, and 24 semitones above the

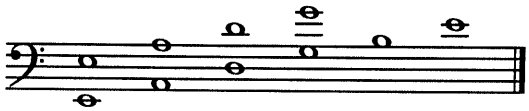
Example 18.1. Sample Tunings for Fretted Instruments.

(a) Common 6-string guitar.



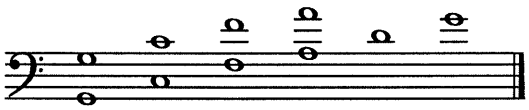
```
AT:E2
*RT:0:5:10:15:19:24
```

(b) Common 12-string guitar.



```
AT:E2
*RT:0,12:5,17:10,22:15,27:19,19:24,24
```

(c) Vieil accord lute.



```
AT:G2
*RT:0,12:5,17:10,22:14,14:19,19:24,24
```

lowest string.

Example (b) defines the most common relative tuning for the 12-string guitar. The six courses are delineated by colons and the tuning of strings withint courses are delineated by commas. In this case, the lower four courses consists of two strings tuned an octave apart, whereas the upper two courses consist of paired unison strings.

Example (c) shows the most common tuning of the 6-course *lute* a tuning referred to as the so-called *vieil accord*: G2, C3, F3, A3, D4, and G4. During the first half of the 16th century, it was common to tune the lower three courses in octaves.

For non-Western and other instruments, it is possible to encode non-integer semitone values for various strings, such as a string tuned 9.91 semitones above the lowest string.

In addition to the absolute and relative tunings, `**fret` also allows the user to specify the tuning of successive frets using the `FT:` tandem interpretation. In Western instruments, frets are normally placed in semitone increments. For a 12-fret instrument, this semitone arrangement may be explicitly represented using the following tandem interpretation:

```
*FT:1,2,3,4,5,6,7,8,9,10,11,12
```

Each successive numerical value indicates the number of semitones above the open string for successive fret positions. The interpretation begins with the tuning of the first fret rather than the tuning of the open string. The above interpretation is similar to the *default fret tuning* — which is an increase of precisely one semitone for each successive fret. The default fret tuning is not limited to 12 frets as in the above example. An instrument constructed with nine 1/4-tone fret positions

can be encoded as follows:

```
*FT:.5,1,1.5,2,2.5,3,3.5,4,4.5
```

The only restriction imposed by `*FT:` is that all strings must have identical fret distances. That is, if the first fret is positioned 1 semitone above the open string, then this relative pitch arrangement must be true of all strings.

The `**fret` representation distinguishes three types of data tokens: tablature-tokens, rests, and barlines. *Tablature-tokens* encode information regarding the fret/finger positions, the manner by which individual strings are plucked (or bowed), pitch-bending, vibrato, damping, harmonics, and other effects. The actions of individual fingers can also be represented. Each tablature-token consists of a several subtokens in the form of Humdrum multiple-stops. Subtokens are delimited by spaces and represent individual courses/strings. A six-string (or six-course) instrument will require six subtokens in each tablature-token. For example, the following tablature token encodes the plucking of the first and sixth string:

```
| - - - - |
```

Subtokens consist of up to five component elements: (1) the string/course status, (2) fret position, (3) bowing/strumming, (4) finger action, and (5) percussive effects. In addition, the tablature-token can encode bowing and strumming information.

In the `**fret` representation, the status of a string/course can occupy one of sixteen states. An *inactive* string is signified by the minus sign (–). An ordinary *plucked* string is represented by the vertical line (|). Plucking near the bridge (*plucked ponticello*) is represented by the slash character (/). Plucking near the tone-hole (*plucked sul tasto*) is represented by the backslash character (\). The repeated *plucked-tremolo* (commonly used on the mandoline) is represented using the octothorpe or hash character (#). *Pizzicato* is represented by the small letter 'z'. Normal bowing of a string is represented by the plus sign (+); *ponticello* bowing is represented by the open parenthesis '(' whereas *sul tasto* bowing is represented by the closed parenthesis ')'. *Spiccato* (bouncing the bow) is represented by the open curly brace '{'. *Col legno* (using the wood of the bow) is represented by the closed curly brace '}'. *Tremolo bowing* is represented by the ampersand (&). *Natural harmonics* and *artificial harmonics* are represented by the lower-case 'o' and upper-case 'O' respectively. String *ringing* is denoted by the colon (:), and the *damping* of a string is denoted by the small letter 'x'.

By way of illustration, the following tablature-token represents a six-string or six-course instrument, where the first through sixth strings are respectively (1 and 2) plucked, (3) damped, (4) bowed, (5) plucked sul tasto, (6) inactive.

```
| | x + \ -
```

Note that the layout of the strings in a tablature-token always corresponds to the tuning specified in the relative-tuning interpretation. In most representations, the lower-pitched strings will be toward the left side of the tablature token.

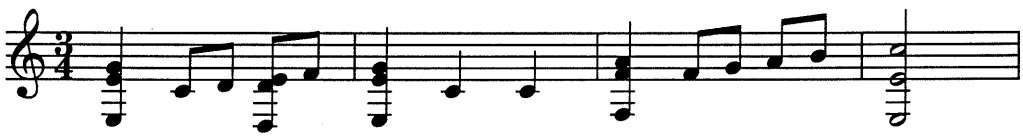
Fret-position information is indicated through the use of numbers, with the first fret signified by the number '1'. Fret-position numbers are encoded immediately to the right of their respective

string/course. For example, the following tablature-token encodes a six-string/course instrument in which the second and third strings are both stopped at the second fret.

```
| |2 |2 | | |
```

Example 18.2 shows a sample passage for guitar with a corresponding `**fret` representation displayed beneath. The `**fret` representation does not encode duration information. It is common to join the `**fret` spine with a `**recip` spine representing the nominal duration data. In example 18.2 a `**kern` spine is also shown indicating the pitches in the `**fret` representation.

Example 18.2. J.S. Bach, *Anna Magdalena Bach Notebook* Menuet II. Guitar arr.



```
**recip   **kern **fret
*         *      *AT:G2
*         *      *RT:0,12:5,17:10,22:14,14:19,19:24,24
*M3/4     *      *M3/4
=1        =1     =1
4         E e g  - |4 - - - |0
8         c      - : : |3 : :
8         d      - : : : |0 x
8         D d e  - |2 : : |2 :
8         f      - : : : |3 :
=2        =2     =2
4         E e g  - |4 : : : |0
4         c      - : : |3 : :
4         c      - : : |3 : x
=3        =3     =3
4         F f a  - |5 : : : |2W
8         f      - : : : |3 :
8         g      - : : : : |0
8         a      - : : : : |2
8         b      - : : : : |4
=4        =4     =4
2         E e cc - |4 : : : |5v
*-        *-     *-
```

The `**fret` representation also provides several short-hand abbreviations for common ornaments and effects. Trills are indicated by the letters 't' (one semitone) and 'T' (two semitones). Mordents are indicated by the letters 'm' (one semitone) and 'D' (two semitones). Inverted mordents are indicated by the letters 'w' (one semitone) and 'W' (two semitones). Turns are indicated by the letters 'S' and '$' (for the inverted "Wagnerian" turn). Two types of vibrato are distin-

guished: 'v' for transverse vibrato and 'V' for lateral vibrato. Pitch bending is signified by the tilde (~).

Apart from tablature-tokens, `**fret` also permits the encoding of rests and barlines. Rests tokens are denoted simply by the lower-case letter 'r'. Barlines are represented using the "common system" for barlines used by `**kern` and other representations.

Additional Features of ***fret*

Bowing-direction and *strumming* information is prepended to the beginning of the tablature-token. The direction of bowing/strumming is encoded using the left and right angle brackets: > means to bow/strum from the strings on the left side of the representation toward the strings on the right side of the representation. (On most instruments this means strumming "downward" — from the lowest- to the highest-pitched strings.) The left angle bracket: < means to strum in the opposite direction. A rough indication of the speed of bowing/strumming can be represented by duplicating these signifiers. For example, >> means a slower "downward" bow/strum, and <<< means an especially slow "upward" bow/strum. The percent sign (%) is used to signify the so-called *rasgueado* — or flamboyant Spanish strum. Once again these signifiers appear at the beginning of a tablature-token — whenever they are encoded. Strumming all 6 open strings downward on a commonly-tuned guitar is represented as:

```
*AT:E2
*RT:0:5:10:15:19:24
>| | | | | |
```

Notice that there is no space between the right angle bracket and the first vertical bar.

The `**fret` representation also permits the optional encoding of *fingering* information. For the plucking-hand (normally right hand), traditional musical abbreviations are used: *P* (pollex) for the thumb, *I* (index) for the index finger, *M* (medius) for the middle finger, *A* (annularis) for the ring finger, and *Q* (quintus) for the little finger. In addition, the lower-case letter *p* is used to signify the palm of the hand. Note that these letters are applied only to the 'plucking' hand. In the case of the 'fret-board' hand, the lower-case letters *a-e* are used to denote the thumb, index finger, middle finger, ring finger, and little fingers, respectively. Like the fret information, fingering information is encoded immediately to the right of the string to which the information applies. By way of illustration, the finger actions used in the above example may be made explicit as follows:

```
>|P |2bP |2cP |P |P |P
```

The strum is carried out by the thumb, while the index and middle fingers of the fret-hand stop the second and third courses/strings at the second fret. In the following continuation of this representation, the first course/string is replucked by the thumb. With the exception of the second and third courses/strings, the other strings are allows to ring.

```
>|P |2bP |2cP |P |P |P
>|P xIM xIM : : :
```

Notice that in damping the vibrations of the second and third strings, both the index and middle

fingers of the 'pluck' hand are used on both strings.

On rare occasions, guitarists will substitute fingers on the fret-board while a string remains sounding. The following example illustrates such a finger-substitution where the middle finger is replaced by the ring finger:

```
| |2b |2c | | |
: :2b :2d : : :
```

Note that in the `**fret` representation, no special signifiers are provided for so-called 'hammer-on' or (ascending-slur), nor for the so-called 'pull-off' or (descending-slur). During the ascending-slur, the sound is produced simply by engaging the next fret. This can be represented in `**fret` by using the "let ring" signifier (:) in conjunction with the appropriate fret notation. The descending-slur can be similarly notated.

Four types of "percussion effects" can be represented using `**fret`. The two most common *tambours* involve tapping on the bridge (represented by the lower-case letter 'u') and tapping on the strings near the bridge (represented by the upper-case letter 'U'). A simple 'tap' on the top-plate is represented by the lower-case letter 'y', whereas a lower-pitched 'thump' on the top-plate is represented by the upper-case letter 'Y'. When sounded alone, these signifiers appear on a line by themselves. When sounded in conjunction with a plucked or (uncommonly) bowed string, these signifiers appear at the beginning of the tablature-token.

The complete system of signifiers used by `**fret` is summarized in Table 18.1.

Table 18.1. Signifiers used by ***fret*.

	Fret-board (left) Hand
1	first fret position
2	second fret position, ...
11	eleventh fret position, etc.
0	open string (not necessarily sounded)
~	bend up in pitch
v	vibrato (transverse)
V	vibrato (lateral)
t	trill (1 fret distance)
T	trill (2 frets distance)
m	mordent (1 fret distance)
D	mordent (2 frets distance)
w	inverted mordent (1 fret distance)
W	inverted mordent (2 frets distance)
S	turn
$	inverted (Wagnerian) turn
a	thumb (of fret hand)
b	index finger (of fret hand)
c	middle finger (of fret hand)

d	ring finger (of fret hand)
e	little finger (of fret hand)
n	no finger (of fret hand)

Pluck (right) Hand

-	unplucked or unactivated string
|	plucked string (normal)
/	plucked string — near bridge (ponticello)
\\	plucked string — near tone-hole (sul tasto)
#	tremolo (plucked, ala mandoline)
z	pizzicato
:	let string ring
x	damp string
o	natural harmonic
O	artificial harmonic
+	bow (normal)
(	bow — near bridge (ponticello)
)	bow — toward fret-board (sul tasto)
{	spiccato
}	col legno (with wood of the bow)
&	tremolo (bowed)
>	strum from low notes to high notes (= down-bow)
<	strum from high notes to low notes (= up-bow)
>>	slower down-strum; slower down-bow
>>	slower up-strum; slower up-bow
>>>	very slow down-strum; very slow down-bow
<<<	very slow up-strum; very slow up-bow
%	rasgueado (Spanish strum)
P	pollex: thumb (of pluck hand)
I	index: index finger (of pluck hand)
M	medius: middle finger (of pluck hand)
A	annularis: ring finger (of pluck hand)
Q	quintus: little finger (of pluck hand)
p	palm (of pluck hand)
N	no finger (of pluck hand)
u	tambour (tap on bridge)
U	tambour (tap on strings near bridge)
y	'tap' on top-plate
Y	'thump' on top-plate

Summary of ****fret** *Signifiers*

A number of pitch-related Humdrum commands accept `**fret` encoded data as inputs, including **cents**, **freq**, **kern**, **pitch**, **semits**, **solfg**, and **tonh**.

Reprise

In this chapter we have tried to reinforce the lesson that `**kern` is only one of an unbounded number of existing and possible Humdrum representations. As a Humdrum user, you are free to concoct your own representations to better address the kinds of information you are interested in manipulating. As long as the resulting representation conforms to the Humdrum syntax, the most important Humdrum tools can still be used to manipulate your data.

Chapter 19

Musical Contexts

Much of what makes an event of interest is the context of the event. We may be interested in what precedes or follows a note or chord. We have already seen how the **-A** and **-B** options for **grep** can be used to output 'before' and 'after' contexts. In Chapter 21 we will see how the **patt** and **pattern** commands can provide further flexibility for searching.

However, in this chapter we introduce the deceptively simple **context** command.

The *context* Command

The effect of the **context** command is easier to illustrate than describe. Consider a file (named `input`) that contains the numbers 1 through 6 on successive lines. A null token is interposed between the numbers 2 and 3:

```
**numbers
1
2
.
3
4
5
6
*-
```

The command

```
context -n 3 input
```

will produce the following output:

```
**numbers
1 2 3
2 3 4
.
3 4 5
4 5 6
.
.
*-
```

In effect, **context** amalgamates data tokens from successive records and assembles them as multiple-stops on a single record. Notice that the number of data records in the output is the same as in the input: **context** has simply padded the trailing data records with null tokens. Also notice that individual data tokens can appear more than once. For example, the number 3 appears at the end of the second line, in the middle of the third line, and at the beginning of the fourth line. In the above example only the numbers 1 and 6 appear once. Finally, notice that null tokens are simply ignored: the null token in the fourth line of our input also appears in the fourth line of the output.

The **-n 3** option tells **context** how many data tokens to amalgamate on each output line. With the specification **-n 2**, just two data tokens would be amalgamated on each output line.

How might **context** be useful? Suppose we wanted to determine how harmonic octave intervals are approached in Bach's two-part keyboard *Inventions*. What harmonic interval tends to precede an octave? We can use the **hint** command to generate the harmonic intervals for each successive sonority. To calculate all *passing intervals*, we will preprocess using **ditto**:

```
ditto inventions* | hint
```

Typical outputs might look like this:

```
M3
M6
A4
=12
M6
m7
M3
A4
M6
A4
M6
P4
M6
M7
m9
m10
d12
```

```
m10
P11
M9
=13
m10
P4
M9
M10
```

Using **context** with the **-n 2** option will cause pairs of successive intervals to appear in the data records. Each data record will consist of a double-stop containing two harmonic intervals. We simply need to identify those data records that have `P8` as the second token of the double-stop. In short, we are interested in data records that end with `P8`. The dollars-sign can be used in a regular expression to anchor the pattern to the end of the line. Hence:

```
ditto inventions* | hint | context -n 2 -o = | grep ' P8$'
```

The **-o =** option tells **context** to omit any data tokens matching the equals-sign — that is, to omit barlines from the amalgamated multiple stops. (The **-o** option accepts any regular expression as a parameter, so omitted data can be defined in a much more refined manner than simply specifying an equals-sign.) The **grep** command grabs all of the lines ending with `P8`. We can now create an inventory of harmonic interval pairs and order them from least common to most common:

```
ditto inventions* | hint | context -n 2 -o = \
   | grep ' P8$' | sort | uniq -c | sort -rn
```

In the case of Bach's fifteen two-part *Inventions* the results look as follows:

```
24  m10 P8
24  M10 P8
23  m7 P8
21  M6 P8
19  M9 P8
12  P5 P8
11  m6 P8
 9  P12 P8
 8  m13 P8
 8  - P8
```

In other words, the octave is most commonly approached by contracting from minor and major tenths rather than expanding from a major sixth interval.

This same basic process can be used to address a variety of similar problems. For example, suppose we wanted to determine the most common word following "gloria" in Gregorian chant texts. We first extract the `**text` spine, use **context** to create pairs of words, and search in the normal way:

```
extract -i '**text' chants* | context -n 2 \
   | grep -i ' gloria$' | sort | uniq -c | sort -nr
```

A slight change to the regular expression for **grep** will allow us to determine what word typically *follows* after the word "gloria." In this case, we need to anchor the word "gloria" to the beginning of the line by using the caret (^).

```
extract -i '**text' chants* | context -n 2 \
   | grep -i '^gloria ' | sort | uniq -c | sort -nr
```

Suppose we wanted to determine what scale degree most commonly precedes the dominant pitch in a sample of Czech folksongs. First we translate the folksongs to the **deg representation using the **deg** command, and then process as above:

```
deg Czech* | context -n 2 -o = | grep '5 ' | sort \
   | uniq -c | sort -nr
```

Harmonic Progressions

The V-I progression is the most common chord progression in Western tonal music. After the V-I progression, what is the most common chord progression in Bach's chorale harmonizations? We will assume that a Roman numeral **harm spine already exists. First we extract the appropriate spine. Then we create context records holding pairs of harmony data (omitting barlines). Then we eliminate global and local comments, interpretations, and null data. We then sort the data records, eliminate duplicates while counting, and then sort by numerical count in reverse order.

```
extract -i '**harm' chorales* | context -n 2 -o = \
   | rid -GLId | sort | uniq -c | sort -nr
```

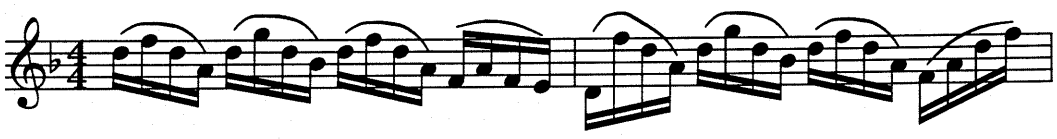
Of course, there is no need to restrict ourselves to pairs of successive data tokens (i.e. **-n 2**) as we have done in the above example. Given a database of melodies, we can determine the most common sequence of five melodic intervals as follows:

```
mint melodies* | context -n 5 -o = | rid -GLId | sort \
   | uniq -c | sort -nr
```

Using *context* with the *-b* and *-e* Options

Example 19.1 shows an excerpt from a flute study by Anderson. Although the work is monophonic, the work's structure is based on an underlying chord progression that is realized as a series of arpeggiation figures.

Example 19.1 Joachim Anderson, Opus 30, No. 24.



The harmonic structure can be made more explicit by amalgamating all of the notes in each arpeggio. There are several possible ways of doing this, but the slurs are particularly useful delineators. The **-b** option for **context** allows the user to specify a regular expression that marks the *beginning* of each collection of data tokens. Consider the following command:

```
context -b '(' Anderson
```

Whenever a data record contains an open parenthesis a new amalgamation begins. The appropriate output for measure 1 of Example 19.1 would be:

```
**kern
*clefG2
*k[b-]
*d:
*M4/4
=1-
(16dd 16ff 16dd 16a)
.
.
.
(16dd 16gg 16dd 16b-)
.
.
.
(16dd 16ff 16dd 16a)
.
.
.
(16f 16a 16f 16e) =2
```

etc.

Notice how the barline for measure 2 has been included in the fourth group. (Groups continue until the next open parenthesis is encountered.) Once again we might eliminate barlines by using the **-o** option. However, sometimes the barlines prove useful in further processing.

In the above passage by Anderson, the close of each slur provides a convenient marker for ending each chord. We can be more explicit in defining the grouping boundaries by also including the **-e** option for **context**. This option allows the user to specify a regular expression that marks the *end* of each collection of data tokens. A suitably revised command would be:

```
context -b '(' -e ')' Anderson
```

The resulting output would begin as follows:

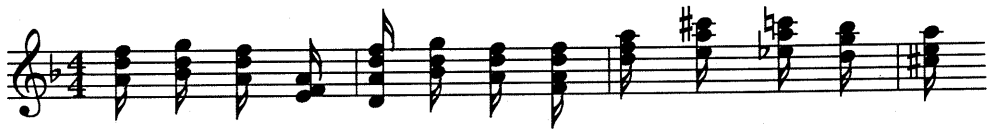
```
**kern
*clefG2
*k[b-]
*d:
*M4/4
=1-
(16dd 16ff 16dd 16a)
.
.
.
(16dd 16gg 16dd 16b-)
.
.
.
(16dd 16ff 16dd 16a)
.
.
.
(16f 16a 16f 16e)
.
.
.
=2
(16d 16ff 16dd 16a)
```

etc.

We could pipe this output to the **ms** command in order to display the re-arranged passage. We place the output in a postscript file and use a display tool such as **ghostview** to display the output:

```
context -b '(' -e ')' Anderson | ms > output.ps
```

Example 19.2 Arpeggio Amalgamation.



Notice that the resulting notation is "ungrammatical" because the meter signature disagrees with the total duration for each measure.

Having reformatted our input data using **context**, we can continue by translating the data to another representation. For example, we might use the **deg** command to reformulate each pitch group as scale degrees. This might allow us to search for particular harmonic patterns such as (say) an augmented sixth chord:

```
context -b '(' -e ')' Anderson | deg | grep '6-' | grep '4+' \
   | grep '1'
```

Any regular expression can be used to identify the beginning and/or ending of an amalgamated group. For example, tokens might be grouped by barlines. Suppose the **census** command tells us that a monophonic work contains sixty-fourth notes. We might want to know whether the sixty-fourth notes all tend to happen in one or two measures, or whether they occur throughout the work. Just how many measures contain sixty-fourth notes?

```
context -b = inputfile | rid -GLId | grep -c '64'
```

Similarly, for `**kern` inputs, the following command counts the number of measures that contain at least one trill:

```
context -b = inputfile | grep -c '^=.*[Tt]'
```

In `**kern` representations, the beginnings and endings of beams are indicated by the letters 'L' and 'J' respectively. We might group notes according to the beaming:

```
context -b L -e J inputfile
```

For example, the following command determines the location of any beams that cross over phrase boundaries:

```
context -b L -e J inputfile | grep -n '}.*{'
```

As in the case of the **-b** option, the **-e** option can be used by itself. This option might prove useful, for example, when collecting all chord functions preceding a cadence. In Bach's chorale harmonizations, for example, cadences are conveniently marked by a pause. In the `**harm` representation, pauses are indicated by the semicolon (`;`). We can create phrase related harmonic sequences as follows:

```
context -o = -e ';' input
```

For example, we might count the number of harmonic functions in each phrase as follows:

```
context -o = -e ';' input | rid -GLId | awk '{print $NF}'
```

In Chapter 22 we will learn how to classify data into discrete categories. Using the **recode** command described in that chapter, we might group notes together according to changes of melodic direction. That is, each group of would consist of notes that are all ascending or all descending in pitch.

Using *context* with *sed* and *humsed*

The stream-editors (**sed** and **humsed**) are especially handy companions for **context**. Suppose we wanted to identify by measure number those measures that contain a *iii-V* progression. Given a `**harm` input, we would first amalgamate all harmony tokens for each measure.

```
context -b ^= inputfile | grep 'iii V' | sed 's/ .*//; s/=//'
```

Here we have used **grep** to isolate all those records that contain the character sequence `iii V`. We have then used **sed** to eliminate all data following the first occurrence of a space. This will leave only the barline token — including the measure number.

When using **grep** it is common for the output to no longer conform to the Humdrum syntax. (This is the reason why we used **sed** rather than **humsed** in the above example.) Remember that we can always use the **yank -m** command to create "grep-like" output that still conforms to the Humdrum syntax. If we wanted to maintain the Humdrum syntax, an equivalent to the above command would be:

```
context -b ^= inputfile | yank -m 'iii V' -r 0 \
   | humsed 's/ .*//; s/=//'
```

The range option (**-r**) specifies that we grab the current record (0) that matches the marker (`iii V`). However, we are free to specify any other range. Consider the following command variation:

```
context -b ^= inputfile | rid -d | yank -m 'iii V' -r 1 \
   | grep 'ii IV' | humsed 's/ .*//; s/=//'
```

This command identifies all those measures containing a *ii IV* progression that have been preceded by a *iii V* progression in the previous measure.

Consider another example. Suppose we wanted to determine whether the first pitch in a phrase tends to be lower than the last pitch in a phrase. As before, we might first amalgamate all notes in each phrase onto individual data records. We can use **humsed** to eliminate all notes other than the first and last. The regular expression `/ .* /` specifies any sequence of characters preceded by a space and followed by a space. Replacing matching strings with a single space will leave output data records consisting of double-stops. The first note of the double-stop will be the first note of the phrase, and the second note of the double-stop will be the last note of the same phrase:

```
context -b { -e } file | humsed 's/ .* / /'
```

We can continue processing by piping the output to the **semits** command. This will leave pairs of numbers representing the semitone distances from middle C. We might then isolate the data records by using **rid**.

```
. . . | semits | rid -GLId | awk '{print $2-$1}'
```

Finally, we have used the UNIX **awk** utility to carry out some simple numerical processing: in this case, substracting the first semitone value from the second one. Phrases that end on a pitch higher than the beginning pitch will have positive semitone outputs. Phrases that end on a pitch lower than the beginning pitch will have negative semitone outputs.

If we wanted to determine the semitone pitch distance *between* phrases, we need only to reverse the begin (**-b**) and end (**-e**) criteria. That is, we will amalgamate the last note of one phrase with the first note in the subsequent phrase. The full pipeline would be as follows:

```
context -b { -e } file | humsed 's/ .* / /' | semits \
   | rid -GLId | awk '{print $2-$1}'
```

Linking *context* Outputs with Inputs

Frequently, we would like to answer context-related questions that mix different types of data together. For example, how many ascending major sixth intervals occur in phrases that end on the dominant? For this question, we need concurrent access to both melodic interval data as well as scale degree information. The solution to such questions typically involves linking different types of data together using the **assemble** command. Suppose the first phrase in our input begins as follows:

```
**kern
*F:
*M3/4
{8Bn
8c
=1
4.a
8g
4f
=2
4g
4d
4e
=3
2c}
*-
```

We need to pursue two independent lines of processing. First we creat a temporary file of scale degree information:

```
mint inputfile > temp.mnt
```

Then we amalgamate the pitch data according the phrasing information, and translate the resulting data to the `**deg` representation:

```
context -b { -e } -o ^= inputfile | deg > temp.deg
```

Next we assemble the two temporary files together to form a single document.

```
assemble temp.mnt temp.deg
```

The first phrase output will appear as follows:

```
**mint    **deg
*F:       *F:
*M3/4     *M3/4
[B]       4+ ^5 ^3 v2 v1 ^2 v6 ^7 v5
+m2       .
=1        .
+M6       .
-M2       .
-M2       .
=2        .
+M2       .
-P4       .
+M2       .
=3        .
-M3       .
```

etc.

We need to search for the interval of an ascending major sixth (+M6) associated with a phrase ending on the dominant (5$). Before using the approprate **grep** command, we need to use **ditto** to propagate the scale degree data over the null data tokens in the **deg spine; **ditto** will generate the following output:

```
**mint    **deg
*F:       *F:
*M3/4     *M3/4
[B]       4+ ^5 ^3 v2 v1 ^2 v6 ^7 v5
+m2       4+ ^5 ^3 v2 v1 ^2 v6 ^7 v5
=1        4+ ^5 ^3 v2 v1 ^2 v6 ^7 v5
+M6       4+ ^5 ^3 v2 v1 ^2 v6 ^7 v5
-M2       4+ ^5 ^3 v2 v1 ^2 v6 ^7 v5
-M2       4+ ^5 ^3 v2 v1 ^2 v6 ^7 v5
=2        4+ ^5 ^3 v2 v1 ^2 v6 ^7 v5
+M2       4+ ^5 ^3 v2 v1 ^2 v6 ^7 v5
-P4       4+ ^5 ^3 v2 v1 ^2 v6 ^7 v5
+M2       4+ ^5 ^3 v2 v1 ^2 v6 ^7 v5
=3        4+ ^5 ^3 v2 v1 ^2 v6 ^7 v5
-M3       4+ ^5 ^3 v2 v1 ^2 v6 ^7 v5
```

etc.

Finally, we use **grep** to search for the composite data:

```
assemble temp.mnt temp.deg | ditto | grep '^+M6.*5$'
```

In addition to linking together different types of data, sometimes we may also need to use a stream editor to modify the data in some way. Suppose we wanted to test a theory that the tonic pitch tends to be followed by a greater variety of melodic intervals than precedes it. That is, we might suspect that the tonic tends to be approached in stereotypic ways — such as from the leading-tone (+m2), from the supertonic (-M2) or from the dominant (+P4); but what follows the tonic may be

less restricted.

In effect, we need to generate two inventories: one for intervals that approach the tonic, and one for intervals that follow the tonic. We already know how to create an inventory of intervals approaching a particular scale-degree:

```
deg -a inputfile > temp1
mint inputfile > temp2
assemble temp1 temp2 | grep '^[v^]*1  ' | sort | uniq -c \
    | sort -rn > inventory.pre
```

For the intervals following the tonic, we need to use **context -n 2**. This will create pairs of intervals: the first interval will indicate the approach, and the second interval in each pair will indicate the continuation.

```
deg -a inputfile > temp1
mint inputfile | context -n 2 -o ^= | temp2
humsed 's/ .*//' temp2 > intervals.pre
humsed 's/.* //' temp2 > intervals.post
assemble temp1 intervals.pre | grep '^1   ' | sort | uniq -c \
    | sort -rn > inventory.pre
assemble temp1 intervals.post | grep '^1  ' | sort | uniq -c \
    | sort -rn > inventory.post
```

In some tasks, it may be necessary to generate more than one **context** output. For example, suppose we wanted to identify possible "cross relations" between two voices. A cross relation occurs when an accidental occurs in one voice but not in another voice within a brief period of time. One approach is to extract each voice, translate to scale-degree and create brief contexts of (say) 2 or 3 notes. E.g.

```
extract -f 1 inputfile | deg | context -n 3 -o ^= > lower.tmp
extract -f 2 inputfile | deg | context -n 3 -o ^= > upper.tmp
```

We can then assemble the two contexts together:

```
assemble lower.tmp upper.tmp
```

Suppose our inputs consisted of an ascending C major scale played in the lower voice concurrent with an E major scale in the upper voice. Our output would look as follows:

```
**deg          **deg
*C:            *C:
1 ^2 ^3        3 ^4+ ^5+
^2 ^3 ^4       ^4+ ^5+ ^6
^3 ^4 ^5       ^5+ ^6 ^7
^4 ^5 ^6       ^6 ^7 ^1+
^5 ^6 ^7       ^7 ^1+ ^2+
^6 ^7 ^1       ^1+ ^2+ ^3
.              .
.              .
*-             *-
```

In effect, each data record contains an agglomeration of three successive notes from both voices. Seaching for cross-relations would entail looking for scale degrees that are both modified and unmodified concurrently. For example, in the case of the subdominant pitch, we could search for such instances as follows:

```
assemble lower.tmp upper.tmp | rid -GLId \
   | egrep '4[+-].* .*4([^+-])|$'
```

The regular expression given to **egrep** searches for a subdominant pitch in the lower voice that is either raised or lowered — concurrent with a subdominant pitch in the upper voice that has not been modified. Notice the use of the tab character in the regular expressions to specify the precise voice being searched. We would also need to test for the reverse situation, where the modified pitch is in the upper voice:

```
assemble lower.tmp upper.tmp | rid -GLId \
   | egrep '4[^+-].*    .*4[+-]'
```

In a similar fashion, the user can mix together spines representing highly diverse types of contextual information to carry out searches for complex patterns or conditions. For example, a user might search for a specific piano fingering that coincides with particular interval-transitions and harmonic contexts.

Using *context* with the *-p* Option

The **-p** option for **context** allows the output data records to be "pushed" forward by a specified number of lines. Consider the normal operation of **context** as illustrated below. The left-hand spine represents the input and the right-hand spine represents the output where the option **-n 2** has been specified.

```
**kern   **kern
*C:      *C:
c        c d
d        d e
e        e f
f        f g
g        g a
a        a b
b        b cc
cc       .
*-       *-
```

Now consider the effect of adding the **-p** option. In this case, the complete command is:

```
context -n 2 -p 1
```

The corresponding result is:

```
**kern   **kern
*C:      *C:
c        .
d        c d
e        d e
f        e f
g        f g
a        g a
b        a b
cc       b cc
*-       *-
```

The data records have been pushed forward by one line: a null token now appears at the beginning of the output spine rather than at the end. Similarly, consider the effect of the following command:

```
context -n 4 -p 2
```

The corresponding result is:

```
**kern   **kern
*C:      *C:
c        .
d        .
e        c d e f
f        d e f g
g        e f g a
a        f g a b
b        g a b cc
cc       .
*-       *-
```

The output is now padded with two preceding null tokens with a trailing null token at the end of the spine. In summary, the **-p** option pushes the context records by a specified number of lines.

This allows us to move the contextual information around, and so provides more possibilities for searching. In the above case, the pitch 'e' is aligned with contextual information that indicates the two pitches that precede 'e' and the one pitch that follows it.

By way of example, suppose we are looking for a submediant pitch that is approached by two melodic intervals of an ascending major third followed by a descending major second. First, we generate independent `**mint` and `**deg` outputs. Next we process the `**mint` data using **context** to create pairs of successive intervals. Without the **-p** option, the assembled output might look as follows:

```
**deg   **mint
*C:     *C:
3       [e] +m2
^4      +m2 +M2
^5      +M2 +M3
^7      +M3 -M2
v6      -M2 +m3
^1      +m3 -P4
v5      .
*-      *-
```

With **-p 1** the output becomes:

```
**deg   **mint
*C:     *C:
3       .
^4      [e] +m2
^5      +m2 +M2
^7      +M2 +M3
v6      +M3 -M2
^1      -M2 +m3
v5      +m3 -P4
*-      *-
```

Now we can search directly for the situation of interest:

```
grep '6 +M3 -M2$'
```

Reprise

The **context** command essentially transforms sequences of events into collections of pseudo-concurrent events. This pseudo-concurrent arrangement enables processing using line-oriented or record-oriented tools — most notably **grep**, **sed**, **humsed** and **awk**. For example, it facilitates pattern searching using **grep** and also allows useful manipulations via tools such as **humsed**. The manner by which data tokens are collected together can be defined by a starting marker or an ending marker or both. Particular types of data can be excluded or omitted from the collections using the **-o** option, and the collections can be transported or pushed forward through the spine using the **-p** option.

We've seen a number of ways by which **context** can be used to establish a particular context for

data. In Chapter 21 we will see how the **patt** command can be used to establish other kinds of contexts and how both of these commands can be used together.

Chapter 20

Strophes, Verses and Repeats

We often tend to think of musical information as a linear stream of successive events. However, there are many circumstances where musical information exhibits more complex structures. These include such structural devices as repeats, da capos, first and second ending, multiple verses, alternative or *ossia* passages, different performance renditions, divergent sources, and competing editions or versions.

This chapter describes the basic Humdrum mechanisms for representing non-linear musical structures. The two critical mechanisms are the Humdrum *section* and *strophe*. We will encounter examples using the **yank**, **thru**, and **strophe** commands.

Section Labels

Musical scores are often notated to take advantage of repetitions in the music. Devices such as repeat marks, *Da Capo*, *Dal Segno*, *Codas*, and other mechanisms make it possible to represent a musical work in an abbreviated format. Humdrum provides corresponding mechanisms that allow works to be represented in succinct ways.

Humdrum files may be logically divided into segments or passages by encoding Humdrum *section labels*. A section label is a type of tandem interpretation that consists of a single asterisk, followed by a greater-than sign, followed by a keyword that labels the section. The following are examples of section labels.

```
*>Coda
*>1st Ending
*>Refrain
*>Exposition>2nd Theme
```

Notice that spaces can appear in section labels — as in `1st Ending`. Sections begin with a section label and generally end when another section label is encountered. Sections also end whenever all spines are assigned new exclusive interpretations, or all spines terminate. If there is more than one spine present in a passage, identical section labels must appear concurrently in all spines.

Expansion Lists

Rather than encode multiple copies of a passage, a single instance may be encoded and labelled as a section. The complete version of the work can be reconstructed by referring to an *expansion list.* An expansion list is another tandem interpretation that contains an ordered list of section labels. The list is specified in square brackets. Like section labels, expansion lists begin with an asterisk followed by a greater-than sign. In effect, the expansion list indicates how the abbreviated file should be expanded to a full-length encoding. Consider the following expansion list:

```
*>[verse1,refrain,verse2,refrain]
```

This list indicates that the abbreviated file contains (at least) three sections, labelled "`verse1`," "`verse2`" and "`refrain`." When the file is expanded, the "`refrain`" section should be repeated following each verse.

Using *yank* to Extract Sections

We encountered the **yank** command earlier in Chapter 12. Recall that **yank** can be used to extract material by *section* using the **-s** option. For example, if the appropriate section is labelled, we might extract the coda of a work as follows:

```
yank -s Coda -r 1 file
```

Recall that the **-r** option is manditory with **yank**; in this case, it identifies the *first* occurrence of a section labelled `Coda`.

Using the *thru* Command to Expand Encodings

The Humdrum **thru** command expands *abbreviated format* representations to a so-called *through-composed format* in which repeated passages are expanded according to an expansion list. When the **thru** command is invoked, it eliminates any expansion lists present in the input; in addition, **thru** places a `*thru` tandem interpretation in all spines immediately following each instance of an exclusive interpretation in the input. This marks the file as being in a through-composed format. Any other `*thru` tandem interpretations encountered in the input are subsequently discarded. As a result, running a file through **thru** twice will not result in further changes to the file.

Alternative Versions

For works encoded in an abbreviated format, it is not always useful to expand it according to a single fixed recipe. Depending on the performance practice, individual performer, or edition, certain repeats may be avoided, passages may be added, or material eliminated altogether. In short, several different versions or interpretations of the overall organization of a work may exist.

Humdrum provides a mechanism by which several alternative versions of the overall organization of a work may co-exist in the same file. This is achieved simply by encoding more than one expansion list. In order to distinguish different versions, each expansion list is given a unique *ver-*

sion label. Consider the following expansion lists:

```
*>Gould1982[A,A,B]
*>Landowska[A,A,B,B]
```

Here we see two expansion lists, one carries the version label `Gould1982` and the other is labelled version `Landowska`. These expansion lists might encode different interpretations of the repeats in a rounded binary form — Landowska performed the second repeat whereas Gould (1982) did not. When the **thru** command is invoked, the user can specify which *version* is intended using the **-v** option. The appropriate through-composed expansion will be output.

The following example illustrates the use of the **thru** command in selecting particular versions of data in a file. Three sections are encoded in the file — labelled A, B and C. Each section in this example contains just a single data record. Three expansion lists are encoded: one is unlabelled, a second is labelled `long` and a third is labelled `weird`.

```
**example          **example
*>[A,B,A,C]        *>[A,B,A,C]
*>long[A,A,B,A,C]  *>long[A,A,B,A,C]
*>weird[C,A,C]     *>weird[C,A,C]
*>A                *>A
data-A             data-A
*>B                *>B
data-B             data-B
*>C                *>C
data-C             data-C
*-                 *-
```

Consider the following command:

```
thru -v weird file
```

The corresponding "through-composed" output would be as follows:

```
**example   **example
*thru       *thru
*>C         *>C
data-C      data-C
*>A         *>A
data-A      data-A
*>C         *>C
data-C      data-C
*-          *-
```

Notice that all expansion-list records have been eliminated from the output. A `*thru` tandem interpretation has been added to all output spines immediately following the exclusive interpretation. Also notice that there are now two sections in the output sharing the same label (`*>C`). This duplication of section-labels is not permitted in abbreviated-format encodings and can only occur in through-composed documents.

Without the **-v** option, **thru** expands the abbreviated file according to the *unlabelled* (default) ex-

pansion list. So the following command would result in an output consisting of section A, followed by section B, followed by section A (again), followed by section C:

```
thru file
```

Section Types

Suppose we had two different theorists — Smith and Jones — who had analyzed the same work differently. Smith thinks there are basically two sections in the work, whereas Jones argues that there are essentially three sections. Humdrum permits alternative schemes of section labels to coexist in a file by allowing the user to designate section *types*. A section label is considered to have a "type" when more than one greater-than sign (>) is present in the label. Consider the following example of sections defined by Smith and Jones:

```
**Example
*>Smith>A
*>Jones>A
data1
*>Jones>B
data2
*>Smith>B
data3
*>Jones>C
data4
*-
```

Both Smith and Jones label the work as beginning with section 'A'. Later Jones's 'B' section begins; then Smith's 'B' section; then Jones's 'C' section. Note that Smith's 'B' section also contains the material Jones has identified as section 'C'.

Normally, the **yank** command extracts a labelled section up to the next occurrence of a section label. However, the **-t** option causes **yank** to ignore all section labels except for a specified type. We could extract Smith's 'B' section by using the **-t** option to limit extraction to "Smith"-type section labels:

```
yank -t Smith -s B
```

This command would produce the following output:

```
**Example
*>Smith>B
data3
*>Jones>C
data4
*-
```

Without the **-t** option, **yank** will simply extract material up to the occurrence of the next section label. Note that section types can be used to define innumerable alternative organizations for a single document.

Hierarchical Sections

For many applications, it is useful to define "nested" structures where two or more sections form part of a larger section. Humdrum section labels allow users to distinguish hierarchical *levels*. Levels are indicated by the number of greater-than signs following the section type. Consider the following:

```
**Example
*>Form>Exposition
data1
*>Form>>1st Theme
data2
*>Form>>2nd Theme
data3
*>Form>Development
data4
*>Form>Recapitulation
*>Form>>1st Theme
data5
*>Form>>2nd Theme
data6
*>Form>Coda
data7
*-
```

All of the above section labels are identified as type `Form`. However, two levels are distinguished (denoted by > and >>). Subsections are specified by increasing the number of greater-than signs, hence `2nd Theme` is a subsection. When **yank** is invoked, it will extract the identified section up to the next section of comparable level. The operation is illustrated in the following sample commands: indicating the first and second themes.

```
yank -t Form -s '1st Theme' -r 1     (extracts up to >Form>>2nd Theme)
yank -t Form -s '2nd Theme' -r 1     (extracts up to >Form>Development)
yank -t Form -s 'Exposition' -r 1    (extracts up to >Form>Development)
```

For example, the second theme from the recapitulation can be extracted as follows:

```
yank -t Form -s '2nd Theme' -r 2
```

Alternatively:

```
yank -t Form -s Recapitulation file | yank -t Form -s '2nd Theme' -r 1
```

Using the *yank* and *thru* Commands

Section labels can be used in a wide number of applications. By way of illustration, here are a few pipeline processes involving section labels. First, we might ask the question — how does the user know what sections labels are present in a document? This is a task for **grep**:

```
grep '^\*>' file
```

This command will also output any expansion-lists. If we want to restrict our output to identifying which *versions* are available for a document we would look for the presence of square brackets:

```
grep '^\*>.*\[.*\]' file
```

How many notes are there in the exposition?

```
yank -t Form -s Exposition -r 1 file | census
```

How many phrases are there in the development?

```
yank -t Form -s Development -r 1 file | grep -c '{'
```

Extract the figured bass for the third recitative:

```
yank -s Recitativo -r 3 file | extract -i '**B-num'
```

Compare the estimated key for the second theme in the exposition versus the estimated key for the second theme in the recapitulation:

```
yank -t Form -s '2nd Theme' -r 1 file | key
yank -t Form -s '2nd Theme' -r 2 file | key
```

Determine the nominal (non-rubato) duration of Gould's performance of the work:

```
thru -v Gould1982 file | extract -i '**kern' | extract -f 1 \
    | dur -d | rid -GLId | grep -v '^=' | stats | grep -i total
```

Perform the first three measures from the second section of a binary form:

```
yank -s B file | yank -o = -r 1-3 | midi | perform
```

Strophic Representations

Section labels and versions allow Humdrum users to select alternative groups of (horizontal) records within a Humdrum file or document. In other circumstances it is useful to be able to select alternative (vertical) paths within a file. Strophic representations may be conceived as "alternative concurrent paths" through a Humdrum document. Examples of alternative concurrent representation paths might include (1) texts for different verses of a song, (2) alternative renditions of the same passage (such as *ossia* passages), or (3) differing editorial interpretations of a given note or sequence of notes.

Structurally, strophic data must begin from a single common spine, split apart into two or more alternative spines, and then rejoin to form a single spine. Since the strophes split from a common spine, they all necessarily begin by sharing the same exclusive interpretation. Different exclusive interpretations may be introduced in the strophic passage — provided all strophic spines end up

sharing the same data type just prior to being rejoined.

The beginning of a strophic passage is signalled by the presence of a *strophic passage initiator* — a single asterisk followed by the keyword "strophe" (`*strophe`). The end of a strophic passage is signalled by the *strophic passage terminator* — a single asterisk followed by the upper-case letter 'S' followed by a minus sign (`*S-`). Each spine within the strophic passage begins with a *strophe label* and ends with a *strophe end indicator* (`*S/fin`). Strophe labels may consist of either alphanumeric names, or numbers. Numerical labels should be used when the strophic data imply some sort of order, such as verses in a song. Alphanumeric labels are convenient for distinguishing different editions or *ossia* passages. The following example encodes a melodic phrase containing four numbered verses from "Das Wandern" from *Die Schoene Muellerin* by Schubert:

```
!! Franz Schubert, 'Das Wandern' from "Die Schoene Muellerin"
**kern          **silbe
*k[b-e-]        *Deutsch
*               *solo
*>[1,1,1,1]     *>[1,1,1,1]
*>1             *>1
*               *strophe
*               *^
*               *^              *^
*               *S/1            *S/2        *S/3        *S/4
8f              Das             Vom         Das         Die
=5              =5              =5          =5          =5
8f              Wan-            Was-        sehn        Stei-
8b-             -dern           -ser        wir         -ne
8a              ist             ha-         auch        selbst,
8ee-            des             -ben        den         so
=6              =6              =6          =6          =6
(16dd           Mül-            wir's       Rä-
16ff)           |               |           |           |
(16dd           -lers           ge-         -dern       sie
16b-)           |               |           |           |
8f              Lust,           lernt,      ab,         sind,
8dd             das             vom         den         die
=7              =7              =7          =7          =7
(8.cc           Wan-            Was-        Rä-         Stei-
16a)            |               |           |           |
8b-             dern!           ser!        dern!       ne!
8r              %               %           %           %
*               *S/fin          *S/fin      *S/fin      *S/fin
*               *v              *v          *v          *v
*               *S-
*-              *-
```

Notice that this file contains a single section labelled '1' and that an expansion list occurs near the beginning of the file that indicates section 1 is to be repeated 4 times in total.

The strophic passage pertains only to the spine marked `**silbe`. The `**silbe` representation pertains to syllabic text encoding and is a pre-defined representation in Humdrum. The `**silbe`

representation is discussed in Chapter 27. Following the strophic passage indicator (`*strophe`), the spine is split apart until the required number of verses are generated. Then each spine is labelled with its own strophe label. Since the verses have an order, it is appropriate to label them with numbers: `*S/1`, `*S/2`, and so on. The individual verses are terminated with strophe end indicators (`*S/fin`), the spines rejoin, and then a strophic passage terminator (`*S-`) marks the end of the strophic passage.

The *strophe* Command

The Humdrum **strophe** command can be used to isolate or extract selective strophic data. The **-x** option for **strophe** allows the user to extract a particular labelled strophe. Consider, for example the effect of the following command:

```
strophe -x 3 schubert
```

Using the above data, the result is:

```
!! Franz Schubert, 'Das Wandern' from "Die Schoene Muellerin"
**kern          **silbe
*k[b-e-]        *Deutsch
*>[1,1,1,1]     *>[1,1,1,1]
*               *solo
*>1             *>1
8f              Das
=5              =5
8f              sehn
8b-             wir
8a              auch
8ee-            den
=6              =6
(16dd           Rä-
16ff)           |
(16dd           -dern
16b-)           |
8f              ab,
8dd             den
=7              =7
(8.cc           Rä-
16a)            |
8b-             dern!
8r              %
*-              *-
```

Notice that all of the tandem interpretations related to the strophe organization are eliminated from the output.

Suppose that we wanted to create a through-composed version of the entire work. We would expect as output, just two spines — the `**kern` spine and the `**silbe` spine. First, we need to create the full length version using the **thru** command. This will take the default expansion list, and

repeat the appropriate section for each successive verse.

```
thru schubert
```

The effect of this will be to simply repeat section 1 four times. However, each repetition will contain all four verses. We can use the **strophe** command to eliminate the unwanted verse texts at each verse. When no option is given, **strophe** operates by preserving strophes in numerical order. That is, when it encounters the first strophic section it will preserve strophe #1 (`*S/1`); then when it encounters the next strophic section it will preserve strophe #2 (`*S/2`). And so on. In summary, the follow command will create a proper through-composed rendition of the Schubert lieder illustrated above.

```
thru schubert | strophe
```

Incidentally, the input passage need not necessary begin with strophe #1. The **strophe** command will adapt to the input, and use the lowest previously unencountered strophe number.

Using the *strophe* and *thru* Commands

As noted, the strophe technique can be used to encode different editorial interpretations of a single work. Suppose for example that we had two editions of the Bach chorale harmonizations: Erk and Reimenschneider. We could select the Erk edition as follows:

```
strophe -x Erk chorale166
```

In a strophic song, suppose we would like to compare the number of syllables in the first and second verses. We begin by selecting the appropriate verse, extract the syllable spine, eliminate all non-data records, eliminate any other special signifiers (like barlines), and finally count the number of remaining records. We repeat this procedure for both verses:

```
strophe -x 1 file | extract -i '**silbe' | rid -GLId \
    | grep -v [=\|%] | wc -l
strophe -x 2 file | extract -i '**silbe' | rid -GLId \
    | grep -v [=\|%] | wc -l
```

(In the `**silbe` representation, the vertical bar (|) and the percent sign (%) have special meanings so the **grep -v** is used to eliminate them along with barlines.)

Reprise

Between stophes and sections, highly non-linear musical documents can be constructed. We have seen how section labels can be defined, how lists of sections ("expansion lists") can be constructed and expanded to through-composed formats using the **thru** command. An unlabelled expansion list is the default version. Other versions have labelled expansion lists.

Several different *types* of section labels can coexist in the same document and the **yank** command can be instructed to ignore all sections other than a certain type via the **-t** option.

The basic ideas introduced in this chapter are summarized in the following table.

section	passage defined by a section label, ends with occurrence of section label of identical level
section label	tandem interpretation beginning: *> and not containing square brackets
section type	first part of section label: *>*type*>
expansion list	tandem interpretation beginning *> and containing a list of section labels in square brackets, e.g. *>[A,B,A]
version	a labelled expansion list, e.g. *>ternary[A,B,A]
level	hierarchical level of a section, designed by the number of '>' following the section type, e.g. *>type>>>name is lower than *>type>name
abbreviated format	Humdrum document encoded using expansion lists
through-composed	Humdrum document encoded without expanion lists
thru	command to create a through-composed document from an abbrevatiated format
thru -v	command to create a particular version of a through-composed document
yank -s	command to extract sections
yank -t -s	command to extract sections limited to sections of a particular type
strophe	1. alternative spine path, 2. command for extracting a particular strophe
strophic passage initiator	tandem interpretation indicating the beginning of a strophe (*strophe)
strophic passage terminator	tandem interpretation indicating the end of a strophe (*S-)
strophe label	tandem interpretation labelling one of several alternative spine-paths, begins *S/
strophe end indicator	tandem interpretation indicating the end of some spine path, e.g. *S/fin

Summary of terms related to sections and strophes

In Chapter 37 we will see further examples of how sections and strophes are especially useful when producing electronic editions.

Chapter 21

Searching for Patterns

The **grep** and **egrep** commands are useful for identifying patterns that occur on single lines. As we saw in Chapter 19, the **context** command can be used to amalgamate groups of successive data tokens on a single line — and so facilitate searching for sequential patterns using **grep** or **egrep**. For many tasks, the combination of **context** and **grep** provides the most convenient way to search for user-specified patterns. However, not all patterns can be conveniently identified using this approach. In this chapter we will introduce two additional tools that are intended to search directly for sequential patterns without having to use **context** to create pseudo-simultaneous collections.

The *patt* Command

The **patt** command may be regarded as a two-dimensional version of **grep**. Like **grep**, **patt** searches for lines that match user-specified regular expressions. However, unlike **grep**, **patt** can search for a sequence of records that match a sequence of user-specified regular expressions. Specifically, **patt** will look for an input line that matches the first (of potential many) user-specified regular expression. Then **patt** will determine whether the following input line matches the second user-specified regular expression ... and so on, until the entire sequence of the user-specified regular expressions are exhausted. A pattern match is deemed to occur only if all of the successive regular expressions match a contiguous sequence of input lines.

The operation of **patt** is easier to describe through an example. Consider the following input using the German *Tonhöhe* pitch designations described in Chapter 4. Recall that the **Tonh system of pitch names allows Bach to spell his name (B=B-flat; H=B-natural). Less well-known is the fact that Dmitri Shostakovich also used the German pitch system to create motives based on his name: D-S-C-H (S=Es=E-flat). (The German transliteration of the cyrillic is Schostakowitsch.)

```
**Tonh
D4
Es4
C4
H3
*-
```

Suppose we were looking for possible instances of D-S-C-H. The **patt** command requires a template file that contains one or more successive regular expressions. A suitable template file (named `dmitri`) would be as follows:

```
D
Es
C
H
```

We would invoke the search as follows:

```
patt -f dmitri inputfile
```

The **-f** option is mandatory: it conveys to **patt** the name of the template file used in the search.

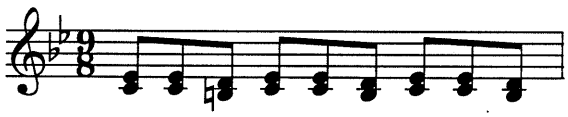
In the default operation, **patt** simply outputs a global comment identifying the location of any matching segments. One global comment is output for each matching pattern. In the above case, the output would be as follows:

```
!! Pattern found at line 2 of file Tonh
```

The **patt** command will also identify any overlapping patterns. For example, suppose we had an input containing an ostinato figure in minor thirds:

Example 21.1. 'DSCH' Ostinato.

```
**Tonh    **Tonh
*k[b-e-] *K[b-e-]
*M9/8     *M9/8
=1-       =1-
C4        Es4
C4        Es4
H3        D4
C4        Es4
C4        Es4
H3        D4
C4        Es4
C4        Es4
H3        D4
=         =
```



If we applied the above **patt** command to this `ostinato` file, we would get the following output:

```
!! Pattern found at line 8 of file ostinato
!! Pattern found at line 11 of file ostinato
```

We can instruct **patt** to output specific instances of the pattern using the **-e** (echo) option. Consider the following command:

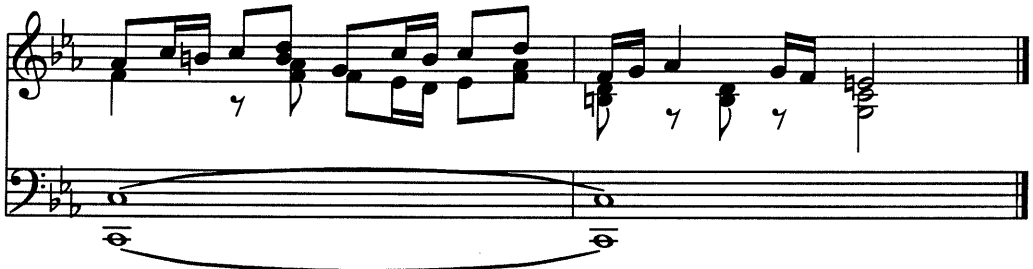
```
patt -f dmitri -e ostinato
```

The resulting output would be:

```
!! Pattern found at line 4 of file ostinato
**Tonh          **Tonh
H3              D4
C4              Es4
C4              Es4
H3              D4
*-              *-
!! Pattern found at line 7 of file ostinato
**Tonh          **Tonh
H3              D4
C4              Es4
C4              Es4
H3              D4
*-              *-
```

Notice that each instance of the found pattern is output as a stand-alone humdrum "mini-encoding," complete with initial exclusive interpretations and terminating spine-path terminators.

Example 21.2. J.S. Bach, *Well-Tempered Clavier*, Vol. 1, Fugue 2.



Most Baroque composers were fond of ending works written in minor keys on the tonic major chord — the so-called *tierce de picardie* or Picardy Third. Example 21.2 shows a typical example from the final measures of Bach's second fugue from the *Well-Tempered Clavier*, vol. 1. Suppose that we wanted to identify all works in some repertory that end with a *tierce de picardie*. We need to search for a raised third scale degree in close proximity to the end of a work for those works in a minor key. First we might identify those works in minor keys. The following **grep** command will search all files in the current directory for a tandem interpretation indicating a minor key. Recall that minor keys are identified by an asterisk followed by a lower-case pitch-letter name, followed by an optional accidental, followed by the colon character. The **-l** option will list all files that containing a matching record:

```
grep -l '^\*[a-g][-#]*:' *
```

Recall that the **deg** command is mode sensitive, whereas the **solfa** command is mode insensitive. That is, in the key of C major, **deg** will represent the pitch E as 3 and in C minor **deg** will represent the pitch E (natural) as 3+. By contrast, the **solfa** command will represent E as 'mi' whether the mode is major or minor.

In order to locate picardy thirds, we can look for raised mediants in the **deg representation. Specifically, we can look for a raised mediant pitch immediately prior to a double barline. Our template file (dubbed "picardy") might look as follows:

```
3[+]
==
```

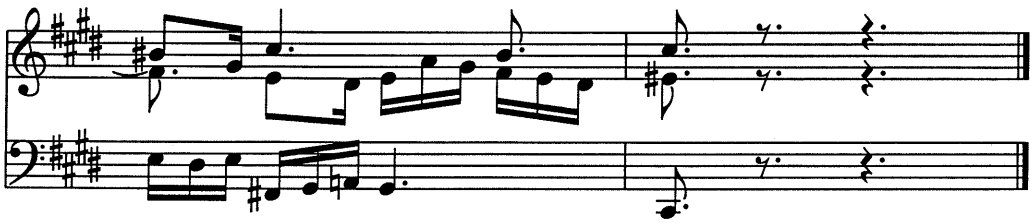
Notice that the plus sign has been placed in square brackets. The **patt** command accepts only *extended* regular expressions. The plus sign is a metacharacter that normally indicates "one or more instances." So placing it in square brackets causes the special meaning to be escaped.

In order to search for such picardy thirds, we should translate each input file to the `**deg` representation, and then search for raised mediants immediately prior to a double bar:

```
deg inputfile.krn | patt -f picardy
```

A problem with this search strategy is that it assumes that the raised third will occur in the final sonority prior to the double barline. One possible confound might be the presence of one or more rests following the final chord. This situation is evident in Fugue No. 4 from the second volume of Bach's *Well-Tempered Clavier*:

Example 21.3. J.S. Bach, *Well-Tempered Clavier*, Vol. 2, Fugue 4.



```
!!!COM: Bach, Johann Sebastian
!!!XEN: The Well-Tempered Clavier, Volume 2, Fugue 4.
**kern          **kern          **kern
*clefF4         *clefG2         *clefG2
*M12/16         *M12/16         *M12/16
*k[f#c#g#d#]    *k[f#c#g#d#]    *k[f#c#g#d#]
*c#:            *c#:            *c#:
=70             =70             =70
16E             8.f#]           8b#
16D#            .               .
16E             .               16g#
16FF##          8e              4.cc#
16GG#           .               .
16AAn           16d#            .
4.GG#           16e             .
.               16a             .
.               16g#            .
.               16f#            8.b#
.               16e             .
.               16d#            .
=71             =71             =71
8.CC#           8.e#            8.cc#
```

```
8.r          8.r          8.r
4.r          4.r          4.r
==           ==           ==
*-           *-           *-
```

The **patt** command provides a **-s** option that allows the user to skip or ignore certain records in the input. Any regular expression can be given as a parameter for the **-s** option. In the following pipeline, we have instruction **patt** to skip over any records matching the lower-case letter 'r' (the **kern rest signifier):

```
deg inputfile.krn | patt -s r -f picardy
```

Even ignoring rests may not be sufficient to identify the raised third near the double barline. For example, if any other note from the tonic chord follows after the raised third, then the third will appear several records prior to the double barline. We can solve this problem by using the **ditto** command discussed in Chapter 15; **ditto** can be used to propagate the raised third through the sustained final chord. Our revised pipeline is:

```
deg bach.krn | ditto -s = | patt -s r -f picardy
```

A similar approach can be used to identify consecutive fifths or octaves between two voices. A template file (dubbed `5ths`) might consist of the following pattern:

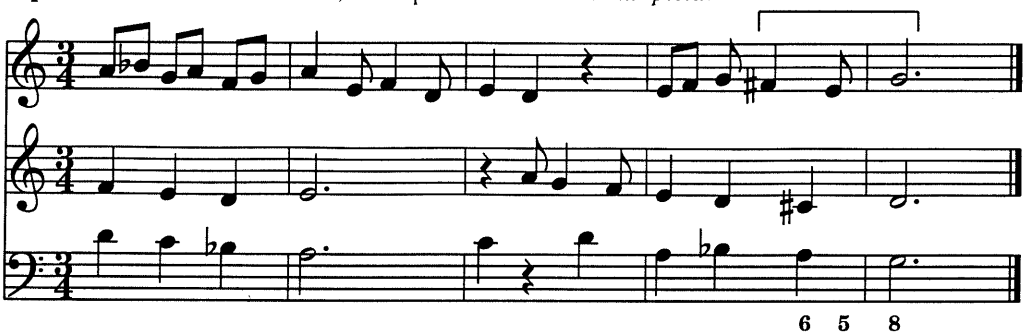
```
P5
P5
```

In order to identify consecutive fifths, we might extract two parts of interest, and then translate to the **hint harmonic-interval representation. The **-c** option for **hint** collapses compound intervals to their non-compound equivalents so consecutive twelfths, nineteenths, etc. will also be identified. In the following command pipeline, notice the use of the **-s** option for **patt** in order to skip barlines. This ensures that crossing a barline does not result in a failure to identify a consecutive fifth.

```
extract -i '*Ibass,*Itenor' Fux | hint -c | patt -s = -f 5ths
```

Sometimes patterns will tend to be obscured by the presence of other information. For example, suppose we want to identify possible Landini cadences such as the cadence shown in Example 21.4. Landini cadences are common in much 14th century polyphony including works by Machaut, Caserta, Dufay, Ciconia, as well as Landini. One characteristic of the Landini cadence is the distinctive three-note *ti* → *la* → *do* in the upper-most part. The submediant pitch is interposed between the leading-tone and the tonic. A second characteristic of the Landini cadence is the harmonic relationship between the highest and lowest voices. Three intervals are formed: *sixth* → *fifth* → *octave*. Either one or both of these characteristics might be used to help identify this distinctive cadential formula.

Example 21.4. Francesco Landini, Excerpt from *Non avrà ma' pietà*.



Below is a **kern encoding of the final two measures along with corresponding **hint and **deg spines. The **hint spine was generated using **hint -l** in order to generate intervals with respect to the lowest pitch.

```
!!!COM: Landini, Francesco
**kern  **kern  **kern  **hint  **deg  **deg  **deg
*clefF4 *clefG2 *clefG2 *       *     *     *
*M3/4   *M3/4   *M3/4   *M3/4   *M3/4 *M3/4 *M3/4
=       =       =       =       =     =     =
4A      4e      8e      P5 P5   v2    v6    ^6
.       .       8f      -       .     .     ^7-
4B-     4d      8g      M3 M6   ^3-   v5    ^1
.       .       4f#     -       .     .     v7
4A      4c#     .       M3      v2    v4+   .
.       .       8e      -       .     .     v6
=       =       =       =       =     =     =
2.G     2.d     2.g     P5 P8   v1    ^5    ^1
==      ==      ==      ==      ==    ==    ==
*-      *-      *-      *-      *-    *-    *-
```

Notice that **hint** has failed to generate the passing interval forming the perfect fifth between the E and the A. This can be remedied by using **ditto** to duplicate all of the pitches. This will cause **hint** to generate all of the passing harmonic intervals. The revised **hint spine is given below.

```
!!!COM: Landini, Francesco
**kern  **kern  **kern  **hint  **deg  **deg  **deg
*clefF4 *clefG2 *clefG2 *       *     *     *
*M3/4   *M3/4   *M3/4   *M3/4   *M3/4 *M3/4 *M3/4
=       =       =       =       =     =     =
4A      4e      8e      P5 P5   v2    v6    ^6
.       .       8f      P5 m6   .     .     ^7-
4B-     4d      8g      M3 M6   ^3-   v5    ^1
.       .       4f#     M3 A5   .     .     v7
4A      4c#     .       M3 M6   v2    v4+   .
.       .       8e      M3 P5   .     .     v6
=       =       =       =       =     =     =
```

```
2.G     2.d     2.g     P5 P8   v1      ^5      ^1
==      ==      ==      ==      ==      ==      ==
*-      *-      *-      *-      *-      *-      *-
```

One way to identify Landini cadences is to use the following harmonic-interval template file (dubbed `LandCadence`):

```
6
5
8
```

Using this template, we can identify Landini cadences as follows. (Notice the use of **-s ^=** to skip barlines.)

```
ditto -s ^= input | hint -l | patt -s ^= -f LandCadence
```

It is possible that the 6-5-8 figured bass might arise in non-cadential situations, so a more circumspect template might also include some scale-degree movements as well. The following template file (dubbed `Landini-Cadence`) combines both the harmonic-interval and scale-degree data:

```
[Mm]6
P5.*v6
P8.*\^1
```

Using this more sophisticated pattern template, a suitable sequence of commands would be the following:

```
ditto -s ^= inputfile | hint -l > temp1
deg inputfile > temp2
assemble temp1 temp2 | patt -s ^= -f Landini-Cadence
```

In general, **patt** templates can be used to specify both concurrent conditions as well as dynamic or temporal conditions. This allows users to define patterns involving a multitude of conditions involving many different types of data.

Using *patt*'s Tag Option

So far, we have seen that **patt** provides two kinds of output. In the default operation, **patt** outputs a simple global comment each time it finds a matching segment in the input. With the **-e** option, **patt** will also echo the specific passage(s) found. In addition, **patt** provides a third type of output using the **-t** option.

When the **-t** option is invoked, **patt** will output the original input, plus an addition `**patt` spine. The `**patt` spine typically consists of mostly null tokens. However, each time the input matches the sought pattern, a user-defined "tag" will appear in the `**patt` spine. Consider the following example.

Suppose we are interested in identifying deceptive cadences in Bach's chorale harmonizations.

Imagine that we already have a **harm spine containing a Roman numeral harmonic analysis. There are different ways of defining a deceptive cadence, but a frequent definition is that it involves a dominant chord followed by a submediant chord in a cadential context. In the case of Bach's chorale harmonizations, cadences are readily identified by the pause symbol. Our search template might look as follows:

```
^V([^I]|$)
(vi)|(VI);
```

This template means: "look for an upper-case letter V appearing at the beginning of a line that is followed by either the end of the line ($) or by a character other than the upper-case letter I. This record will be followed by a record containing either vi or VI followed by a semicolon."

When invoking the **patt** command, we can specify our preferred output tag along with the **-t** option as follows:

```
extract -i '**harm' bwv269.krn | patt -f template -t deceptive

**harm   **patt
I        .
I        .
ii7      .
V        deceptive
vi;      .
V        .
I        .
IV       .
IV       .
I        .
V;       .
etc.
```

In Chapter 26 we will learn how to collapse several spines into a single spine. This will allow us to assemble the results from several "passes" using **patt** — one pass for each type of cadence. For example, we could collapse several tagged outputs to produce a single spine that identifies all of the various types of cadences:

```
**harm   **cadences
I        .
I        .
ii7      .
V        deceptive
vi;      .
V        .
I        .
IV       .
IV       .
I        half
```

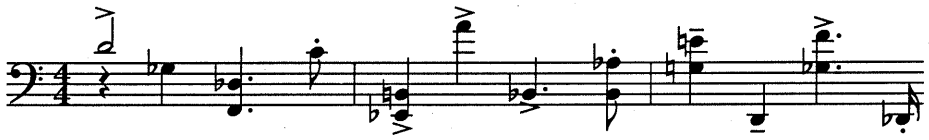
```
V;          .
etc.
```

There are no restrictions as to the types of tags that can be generated by **patt**. A user might tag the beginning of motivic or thematic statements, various harmonic progressions, variation techniques, fingering patterns, quotations or allusions, stylistic clichés, etc. In Chapter 35 we will use the **-t** option to label different set forms for statements of a twelve-tone row, such as primes, inversions, retrogrades, and retrograde inversions. We will use suitable tags to identify the specific transpositions: P0, I7, R11, RI8, etc.

Matching Multiple Records Using the *patt* Command

Twelve-tone music raises several special issues for sequential pattern matching. For example, it is common in serial music to collapse segments of a tone-row in order to create vertical chords. Consider the following excerpt from Ernst Krenek's suite for solo 'cello. The tone row consists of the ordered pitches: D, G-flat, F, D-flat, C, B, E-flat, A, B-flat, A-flat, E, G.

Example 21.5. Ernst Krenek, Opus 84 *Suite for Violoncello*; mov. 1, measures 28-30.



Using a pitch-class representation we would search for the sequence:

```
2
6
5
1
0
11
3
9
10
8
4
7
```

Due to the diads, however, the corresponding pitch-class representation for the above Krenek passage would be:

```
**pc
2
6
5 1
0
=
```

```
3 11
9
10
10 8
=
7 4
etc.
*-
```

The **-m** option for **patt** invokes a "multi-record matching" mode. In this mode, **patt** attempts to match as many successive regular expressions in the template file as possible for a given input record, before continuing with the next input and template records. In this way, several records in the template file may possibly match a single input record. In the above case, the tone-row template will be matched and the 'P0' tag issued if the following command is issued:

```
patt -f tonerow -t P0 -m Krenek
```

The *pattern* Command

Not all patterns can be identified using The Humdrum **pattern** command permits an additional regular expression feature that is especially useful in musical applications. Specifically, **pattern** permits the defining of patterns spanning more than one line or record. Record-repetition operators are specified by following the regular expression with a tab — followed by either +, *, or ?. For example, consider the following Humdrum-extension regular expression:

```
X       +
Y       *
Z       ?
```

When the metacharacters +, *, or ? appear at the end of a record, preceded by a tab character, they pertain to the number of records, rather than the number of repetitions of the expression within a record. The first record of the regular expression (X<*tab*>+) will match one or more successive lines each containing the letter 'X'. The second record of the regular expression (Y<*tab*>*) will match zero or more subsequent lines containing the letter 'Y'. The third record of the regular expression (Z<*tab*>?) will match zero or one line containing the letter 'Z'. Hence, the above multi-record regular expression would match an input such as the following: three successive lines containing the letter 'X', followed by eight successive lines containing the letter 'Y', followed by a single line containing the letter 'Z'. Similarly, the above regular expression would match an input containing one line containing the letter 'X'.

Record-repetition operators can be used in conjunction with all of the other regular expression features. For example, the following regular expression matches one or more successive **kern data records containing the pitch 'G' (naturals only) followed optionally by a single 'G#' followed by one or more records containing one or more pitches from an A major triad — the last of which must end a phrase:

```
[Gg]+[^#-]    +
[Gg]+#[^#]    ?
```

```
([Aa]+|([Cc]+#)|[Ee]+)[^#-]*
(}.*([Aa]+|([Cc]+#)|[Ee]+)[^#-]))|(([Aa]+|([Cc]+#)|[Ee]+)[^#-].*})
```

Patterns of Patterns

Music often exhibits hierarchical structures where particular types of patterns may be embedded in other patterns, or where low-level patterns join together to form higher-level patterns. As we have seen, the **-t** (tag) option for the **patt** command allows a new output spine to be generated. This spine contains user-defined labels marking the beginning of each found pattern. The labels can contain any user-defined text string such as `authentic cadence`, `episode`, `Motive 3b`, `augmentation`, `triplet figuration`, or `prolongation`.

As we will see in Chapter 26, the contents of several spines can be amalgamated to form a single spine. This means that the results for several independent pattern searches can be assembled into a single "pattern" spine. Several pattern spines may be created that related to patterns found at different hierarchical levels, or patterns found using different search methods. Of course, these pattern-spines themselves can be used as input to further pattern searches thus providing unbounded possibilities for searching for patterns of patterns.

Consider, for example, the following template for the **pattern** command:

```
Theme 1 (tonic)        +
Bridge                 *
Theme 2 (tonic)        +
Coda                   ?
```

The template reads "one or more instances of `Theme 1 (tonic)`, followed by zero or more instances of `Bridge`, followed by one or more instances of `Theme 2 (tonic)`, followed by zero or one instance of `Coda`." This template might be used by **pattern** to identify a Recapitulation. Together with outputs from parallel searches for 'Exposition' and 'Development' the results of a 'Recapitulation' search might similarly be amalgamated and used as an input for a higher level search for works exhibiting a sonata-allegro structure.

Reprise

In this chapter and previous chapters we have identified several search-related tools, including the UNIX **grep** and **egrep** commands as well as the Humdrum **patt** and **pattern** commands. Each of these tools has different strengths and weaknesses and it is not always clear which tool is best for a given task. When searching, don't forget to consider how **context**, **humsed**, **rid** and other tools might facilitate the searching task. In future chapters will will consider how "similarity" tools such as **correl** and **simil** can contribute to more sophisticated pattern searches.

Chapter 22

Classifying

Many of the most important analytic tasks involve classifying or categorizing various things. In this chapter we will discuss two general approaches to classifying: *parametric* classifying and *non-parametric* classifying. In the first instance, we will see how numerical data can be categorized according to arithmetic ranges. We will then revisit the **humsed** command and learn how it can be used to classify different types of non-numeric data tokens.

The *recode* Command

Suppose that we have a Humdrum spine that contains numerical information representing the moment-to-moment heart-rate of a listener. Heart rate is related to arousal level and so we might use our data to identify passages that tend to arouse listeners. Since the average heart-rate of listeners differs, we are interested primarily in the rate-of-change. We can use the **xdelta** command to calculate the differences in heart-rate between successive values.

```
xdelta -s = heart.dat > changes
```

The example below displays the input (left) spine and the corresponding output (right) spine for the above command:

```
**heart   *Xheart
=133      =133
55        0
56        1
55        -1
=134      =134
58        3
56        -2
55        -1
=135      =135
57        2
55        -2
56        1
```

```
=136        =136
55          -1
60          5
62          2
=137        =137
61          -1
59          -2
59          0
=138        =138
*-          *-
```

A certain amount of heart-rate variation is to be expected because of monitoring equipment and other variables. So we are primarily interested in large changes of heart-rate, such as the change occurring in measure 136. The **recode** command allows us to classify numerical data according to value or range. In the above case, we may be interested in identifying acceleration or decelerations that exceed some threshold. The **recode** command requires that the user supply a *reassignment file* that specifies how numerical values are to be reassigned. In our heart-rate application, we might create the following reassignment file, named `reassign`. Reassignment files obey the following syntax: for each line, *conditions* are given on the left followed by a single tab, followed by a *reassignment string.*

```
>3      +event
<-3     -event
else    .
```

The above reassignment file may be interpreted as follows: if the numerical value is greater than 3, then output the string "`+event`"; if the numerical value is less than −3, then output the string "`-event`"; otherwise output a string consisting of an isolated period (`.`). We can invoke an appropriate command as follows:

```
recode -f reassign -i '**Xheart' -s ^= changes
```

The **-f** option is required, and is used to identify the file containing the reassignment information. The **-i** option is also required, and is used to identify the exclusive interpretation for the data to be processed. The **-s** option tells **recode** to skip records matching some specified regular expression — in this case, to skip barlines. Finally, "`changes`" is the name of our input file.

The result of applying this process to the right-most spine in the above example is given below:

```
*Xheart
=133
.
.
.
=134
.
.
.
```

```
=135
.
.
.
=136
.
+event
.
=137
.
.
.
=138
*-
```

Notice that we have used **recode** to drastically reduce the volume of data by transforming the input into a set of more basic cateogires.

Having constructed our new output spine, we can further process this information in various ways. For example, we might assemble this spine to our original musical score. Then we might then use **grep -n** to located any points in the score where a heart-rate related event has occurred.

Permissible relational operators used by **recode** include the following:

==	equals
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
else	default relation

Conditions are tested in the order given in the reassignment file. Thus if a numerical value satisfies more than one condition, only the first string replacement is made. Consider the following reassignment file:

```
<=0       LOW
>100      HIGH
>0        MEDIUM
```

The order of specification is important here. If the `MEDIUM` condition was specified prior to the `HIGH` condition, then all values greater than one hundred would be categorized as `MEDIUM` rather than as `HIGH`. Only a single **else** condition is allowed in a reassignment file; when it is present, the else statement should appear as the last reassignment.

Classifying Intervals

The **recode** command has innumerable applications. Suppose we wanted to determine how frequently ascending melodic leaps are followed by a descending step. Let's consider two different ways of distinguishing steps and leaps: a "semitone" method and a "diatonic" method. In the first method, we might define a step interval as either one or two semitones. Our reassignment file (dubbed "`reassign`") might appear as follows:

```
>=3     up-leap
>0      up-step
==0     unison
>=-2    down-step
<=-3    down-leap
```

Given this reassignment file, we can now begin our processing. In the first method, we translate to semitone data using **semits**, translate to semitone-differences using **xdelta**, and then classify into five interval types using **recode**. The **context -n 2** command will create pairs of interval types, then **rid**, **sort** and **uniq -c** are used to generate an inventory. Finally, we use **grep** to identify what happens following ascending leaps:

```
semits melody | xdelta -s = | recode -f reassign \
   -i '**Xsemits' -s = | context -n 2 | rid -GLId | sort \
   | uniq -c | grep 'up-leap .*$'
```

An alternative way of distinguishing steps from leaps is by diatonic interval. For example, we might consider a diminished third to be a leap, while an augmented second may be considered a step. In this case, we can use the **mint** command to determine the melodic interval size; the **-d** option limits the output to diatonic intervals and excludes the interval quality (perfect, major, minor, etc.). The appropriate reassignment file would be:

```
>=3     up-leap
==2     up-step
==1     unison
==-2    down-step
<=-3    down-leap
```

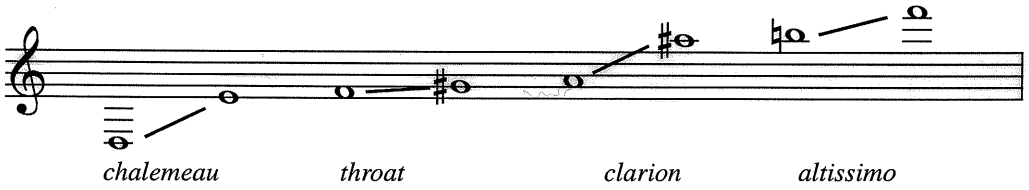
The appropriate command pipe would be:

```
mint melody | xdelta -s = | recode -f reassign -i '**mint' \
   -s = | context -n 2 | rid -GLId | sort | uniq -c \
   | grep 'up-leap .*$'
```

Clarinet Registers

Consider another use of the **recode** command. Imagine that we wanted to arrange Claude Debussy's *Syrinx* for soprano clarinet instead of flute. Our principle concern as arranger is determining what key would be especially well suited to the clarinet. Tone color is particularly important for this piece. The clarinet has four fairly distinctive tessituras as shown in Example 21.1. These are the *chalemeau* register (dark and rich), the *clarion* register (bright and clear), the *altissimo* register (very high and piercing), and the *throat* register (weak and breathy).

Example 21.1. Clarinet registers (notated at concert pitch).



Suppose we wanted to pick a key that satisfies two conditions: (1) it is not out of range for the clarinet, and (2) it minimizes the number of notes played in the throat register. We can use **recode** to classify all pitches according to the following reassignments:

```
>=30   too-high
>=23   altissimo
>=8    clarion
>=5    throat-register
>=-10  chalemeau
else   too-low
```

Now we simply explore various transpositions using **trans** and create an inventory of pitch types. For Debussy's *Syrinx*, the minimum number of throat tones (without exceeding the clarinet's range) occurs when we transpose down a major sixth:

```
trans -d -5 -c -9 syrinx | semits | recode -f reassign \
   -i '**semits' -s = | rid -GLId | sort | uniq -c
```

Open and Close Position Chords

Inputs to the **recode** command can be quite sophisticated. Consider, for example, the task of classifying chords as "open" or "close" position. According to one definition, a chord is said to be in "open" position when the the interval separating the soprano and tenor voices is an octave or greater. One music theorist has claimed that close position chords are more common than open position. How might we test this?

In determining an appropriate sequence of Humdrum commands, it is often helpful to work backwards from our goal. We'd like to end up with a spine that simply encodes the words "open" or "close" for each sonority. This classification will be based on the distance separating the soprano and tenor voices. Our reassignment file might be as follows:

```
<=12      close
>12       open
```

We will need to extract the soprano and tenor voices, translate the pitch representation to `**semits` and use **ydelta** to calculate the semitone distance between the two voices. In the following set of commands, we have also added the **ditto** command to ensure that there are semitone values for each sonority.

```
extract -i '*Itenor,*Isopran' inputfile | semits -x | ditto \
   | ydelta -s = -i '**semits' | recode -f reassign \
   -i '**Ysemits' -s = > tempfile
grep -c 'open' tempfile
grep -c 'close' tempfile
```

The **grep -c** commands tell us whether open position sonorities are more common than close position sonorities.

Flute Fingering Transitions

There is no fixed limit to the length of a reassignment file. Consider for example, the following file named `map`. Each `**semits` value from C4 (0) to C7 (36) has been assigned to a schematic representation of flute fingerings. The letter 'X' indicates a closed key, whereas the letter 'O' indicates an open key. The first letter pertains to the left thumb; the next group of four letters pertain to the ensuing fingers of the left hand; the final group of letters pertain to the right-hand fingers. The little finger of the right hand is able to play three keys (labelled X, Y, and Z). Fingerings are shown only for the first octave (from C4 to C5):

```
<0      out-of-range
==0     X-XXXO-XXXZ
==1     X-XXXO-XXXY
==2     X-XXXO-XXXO
==3     X-XXXO-XXXX
==4     X-XXXO-XXOX
==5     X-XXXO-XOOX
==6     X-XXXO-OOXX
==7     X-XXXO-OOOX
==8     X-XXXX-OOOX
==9     X-XXOO-OOOX
==10    X-XOOO-XOOX
==11    X-XOOO-OOOX
==12    O-XOOO-OOOX

etc.

else    rest
```

Suppose we wanted to determine what kinds of fingering *transitions* occur in Joachim Quantz's flute concertos. Since instrument fingerings are insensitive to enharmonic spelling, an appropriate

input representation would be `**semits`. Having used **recode** to translate the pitches to fingerings, we can then use **context -n 2** to generate diads of successive finger combinations.

```
semits con* | recode -f map -s = | context -n 2 -o = > fingers
```

For example, if our input contains the pitch G5 followed by B4, the appropriate data record in the `fingers` file would be the following Humdrum double-stop:

```
X-XXXO-OOOX X-XOOO-OOOX
```

We could create an inventory of finger transitions by continuing the processing:

```
rid -GLI fingers | sort | uniq -c | sort -n
```

We could create a similar reassignment file containing fingers pertaining to the pre-Boehm flute. Suppose the revised reassignment file was called `premodern`. We could determine how the finger transitions differ between the pre-Boehm traverse flute and the modern flute. In Chapter 29 we will see how the **diff** command can be used to identify differences between two spines. This will allow us to identify specific places in the score where Baroque and modern fingerings differ.

The **recode** command can be used for innumerable other kinds of classifications. For example, `**kern` durations might be expressed in seconds (using the **dur** command), and the elapsed times then classified as *long*, *short* and *medium* (say). Sound pressure levels (in decibels) might be classified as dynamic markings (***ff***, ***mf***, ***mp***, ***pp***, etc.), and so on.

Classifying with *humsed*

The **recode** command is restricted to classifying numerical data only. For many applications, it is useful to be able to classify data according to non-numerical criteria. As we saw in Chapter 14, stream editors such as **sed** and **humsed** provide automated substitution operations. Such string substitutions can be used for non-parametric classifying. We can illustrate this with **humsed.**

Suppose we wanted to classify various flute finger-transitions as either *easy*, *moderate* or *difficult*. For example, F4 to G4 is an easy fingering, E5 to A5 is a moderate fingering, whereas C5 to D5 is difficult. As before, it is best to use a semitone representation so we don't need to consider differences in enharmonic pitch spelling. We can use the **semits** command to transform all pitches. Then we can use **context -n 2** to generate pairs of successive pitches as double-stops. We can then create a **humsed** script file (let's call it `difficulty`) containing substitutions such as the following:

```
s/5 7/easy/              [i.e. F4 to G4]
s/16 21/moderate/        [i.e. E5 to A5]
s/12 14/difficult/       [i.e. C5 to D5]
etc.
```

We can apply the script as follows:

```
humsed -f difficulty sonata*
```

Since there are a large number of possible pitch transitions, our script file is apt to be especially large. However, notes an octave apart have a high likelihood of having identical fingerings on the modern flute. A more succinct **humsed** script would deal with fingering transitions rather than pitch transitions.

```
s/X-XXXO-XOOX X-XXXO-OOOX/easy/
s/X-XXXO-XXOX X-XXOO-OOOX/moderate/
s/O-XOOO-OOOX X-OXXO-XXXO/difficult/
```
etc.

The three substitutions shown above apply to many more pitch transitions than the original transitions F4-G4, E5-A5, and C5-D5. The above three substitutions apply also to F5-G5, F5-G4, F4-G5, E4-A4, E4-A5, and E5-A4.

Having created a file classifying all fingering transitions as "easy," "moderate" or "difficult," we can characterize our Quantz flute concertos using the following pipeline:

```
semits Quantz* | recode -f map -s = | context -n 2 -o = \
    | humsed -f difficulty
```

The output will be a single spine that classifies the difficulty of all fingering transitions.

Classifying Cadences

Consider another application where we use **humsed** to classify cadences. Suppose we have Roman-numeral harmonic data (as provided by the `**harm` representation). In the case of Bach's chorale harmonizations, for example, cadences are clearly evident by the presence of pauses (designated by the semicolon). We can easily create a spine that identifies only cadences. Consider a suitable reassignment file (dubbed `cadences`):

```
s/V I;/authentic/
s/V7 I;/authentic/
s/V i;/authentic/
s/V7 i;/authentic/
s/IV I;/plagal/
s/iv i;/plagal/
s/iv I;/plagal/
s/V vi;/deceptive/
s/V VI;/deceptive/

etc.

s/^[IiVv].*$/./
```

(The precise file will depend on your preferred way of labeling cadences.) Remember that, unlike the **recode** command, all of the substitutions in a **humsed** or **sed** script are applied to every input line. The final substitution causes any record beginning with either an *i*, *I*, *v* or *V* to be changed to a null data token. In effect, any progression that is not deemed to be an authentic, plagal or decep-

tive cadence is transformed to a null data record. Using the above reassignment file, we could create a cadence spine using the following pipeline:

```
extract -i '**harm' chorales | context -o = -n 2 \
   | humsed -f cadences | sed 's/\*\*harm/**cadences/'
```

We first extract the `**harm` spine using **extract**. We then generate a sequence of two-chord progressions using **context** — taking care to omit barlines (`-o =`). We then use **humsed** to run the script of cadence-name substitutions. Finally, we use the **sed** command to change the name of the exclusive interpretation from `**harm` to something more suitable — `**cadences`.

Many more sophisticated variants of this sort of procedure may be used. For example, one could first classify harmonies more broadly. In so-called "functional" harmony, for example, supertonic chords in first inversion are normally considered to be subdominant functions. One could construct a whole series of re-write rules that classify harmonies in a variety of ways.

Orchestration

One of the simplest classifications in a musical score is whether or not an instrument is sounding or resting. Suppose we extracted the viola part from Beethoven's Symphony No. 1. We might use the **ditto** command to ensure that each data record encodes either a note, rest, or barline:

```
extract -i '*Iviola' symphony1 | ditto -s =
```

Let's append to this pipeline a **humsed** command that makes two string substitutions. The first substitution replaces all data records containing the lower-case letter `r` (i.e., rests) with the string `-viola`. The second substitution changes any record that does not begin with either a minus sign or an equals sign to the string `+viola`. In effect, we've transformed the viola part so that all data tokens encode either `+viola`, `-viola` or are barlines.

```
extract -i '*Iviola' symphony1 | ditto -s = \
   | humsed 's/.*r.*/-viola/; /s/^[^-=].*$/+viola/' > viola
```

Now imagine that we repeat this process for every instrument in Beethoven's Symphony No. 1. In each case, we substitute the name of the instrument (preceded by a plus-sign or minus-sign) for the various note or rest tokens.

```
extract -i '*Iflt' symphony1 | ditto -s = \
   | humsed 's/.*r.*/-flt/; /s/^[^-=].*$/+flt/' > flt
extract -i '*Ioboe' symphony1 | ditto -s = \
   | humsed 's/.*r.*/-oboe/; /s/^[^-=].*$/+oboe/' > oboe
extract -i '*Iclars' symphony1 | ditto -s = \
   | humsed 's/.*r.*/-clars/; /s/^[^-=].*$/+clars/' > clars
extract -i '*Ifagot' symphony1 | ditto -s = \
   | humsed 's/.*r.*/-fagot/; /s/^[^-=].*$/+fagot/' > fagot
```

etc.

When we are finished, we reassemble all of the transformed parts into a complete score.

```
assemble cbass cello viola violn2 violn1 tromb tromp fagot \
   clars oboe flt > orchestra
```

We now have a file that contains data records that look something like the following excerpt:

```
+cbass  +cello +viola +violn +violn -tromb -tromp +fagot -clars+oboe  +flt
+cbass  +cello -viola -violn +violn -tromb -tromp +fagot -clars+oboe  +flt
+cbass  +cello +viola +violn +violn -tromb -tromp +fagot -clars+oboe  +flt
+cbass  +cello -viola -violn +violn -tromb -tromp +fagot -clars+oboe  +flt
-cbass  -cello +viola +violn +violn -tromb -tromp -fagot -clars+oboe  +flt
-cbass  -cello -viola -violn +violn -tromb -tromp -fagot -clars+oboe  +flt
=131    =131   =131   =131   =131   =131   =131   =131   =131  =131   =131
+cbass  +cello +viola +violn +violn -tromb -tromp +fagot -clars+oboe  +flt
+cbass  +cello -viola -violn +violn -tromb -tromp +fagot -clars+oboe  +flt
-cbass  -cello +viola +violn +violn -tromb -tromp -fagot -clars+oboe  +flt
-cbass  -cello -viola -violn +violn -tromb -tromp -fagot -clars+oboe  +flt
+cbass  +cello +viola +violn +violn -tromb -tromp +fagot -clars+oboe  +flt
+cbass  +cello -viola +violn +violn -tromb -tromp +fagot -clars+oboe  +flt
etc.
```

The first sonority indicates that all of the string instruments are playing, that the brass are inactive, and that all of the woodwinds are sounding with the exception of the clarinet.

A representation such as the above provides an opportunity to study instrumental combinations in Beethoven's orchestration. For example, the following command will count the number of sonorities where the oboe and bassoon sound concurrently:

```
grep -c '+fagot.*+oboe' orchestra
```

It is better to express this count as a proportion of the total work. We can count the total number of sonorities in the work by omitting any leading plus or minus sign:

```
grep -c 'fagot.*oboe' orchestra
```

How often are the oboe and bassoon resting at the same time?

```
grep -c '-fagot.*-oboe' orchestra
```

Excluding *tutti* sections, do the trumpet and flute tend to "repell" each others' presence?

```
grep -v '\-' orchestra | grep -c '+tromp.*-flt' orchestra
grep -v '\-' orchestra | grep -c '+tromp.*+flt' orchestra
grep -v '\-' orchestra | grep -c '-tromp.*-flt' orchestra
grep -v '\-' orchestra | grep -c '-tromp.*+flt' orchestra
```

When all of the woodwinds are playing, which of the remaining instruments is Beethoven most likely to omit from the texture?

```
grep '+fagot.*+clars.*+oboe.*+flt' orchestra | grep -c '-cbass'
grep '+fagot.*+clars.*+oboe.*+flt' orchestra | grep -c '-cello'
grep '+fagot.*+clars.*+oboe.*+flt' orchestra | grep -c '-viola'
grep '+fagot.*+clars.*+oboe.*+flt' orchestra | grep -c '-violn'
etc.
```

Many refinements can be added to this basic approach. For example, instead of classifying instruments as simply being "present" or "absent," we might distinguish various registers for each instrument — as we did with the clarinet when describing **recode**. We could then determine whether Beethoven tends to link, say, activity in the chalemeau register of the clarinet with low register activity in the strings.

Further refinements might include relating orchestration to structural aspects of the music. For example, we might use **yank** to extract sections of movements; we could then compare possible differences of orchestration between the first and second themes, for example. Similarly, we could reduce instruments to instrument classes, and examine how brass, woodwinds, strings, and percussion in general are related.

Reprise

A large number of analytic tasks simply involve classifying things. In general, two sorts of classifying methods can be distinguished: (1) a numerical or *parametric* classification can be used to reassign various ranges of numerical values into a finite set of classes or categories; (2) a *non-parametric* classification maps one set of words or terms into a second (usually smaller) set of words (used to label various classes or categories). In this chapter, we have seen that, for any Humdrum representation, parametric classification can be done using the **recode** command and non-parametric classification can be achieved using the *substitution* operation provided by the **humsed** command.

Chapter 23

Rhythm

The subject of rhythm touches on nearly every aspect of music. Musical elements such as pitch, harmony, and dynamics can all be regarded from the point-of-view of temporal patterns of events. A number of complex tasks arise from rhythm-related In this chapter, two rhythm-related tools are introduced: **dur** and **metpos**.

The ***recip* Representation

For many types of processing tasks it is helpful to have a representation that encodes rhythmic information only. The `**recip` representation is simply a subset of `**kern` that excludes all information apart from the nominal note durations and common system barlines. In addition, `**recip` distinguishes rests from notes by including the 'r' signifier. Without an accompanying 'f↵r' a duration is assumed to pertain to a note.

Generating `**recip` data from `**kern` is straightforward using **humsed**. For a single-spine input, the following command will make the translation:

```
humsed '/^[^=]/s/[^0-9.r ]//g; s/^$/./' input.krn \
    | sed 's/\*\*kern/**recip/'
```

The first **humsed** substitution eliminates all data other than the numbers 0 to 9, the period, the lower-case `r`, and the space (for multiple-stops). Barlines remain untouched in the output. The second **humsed** substitution changes any empty lines to null data tokens; this might be necessary in the case of grace notes. The ensuing **sed** command is used simply to change the exclusive interpretation from `**kern` to `**recip`.

A simple type of processing might entail creating an inventory of rhythmic patterns. Suppose we wanted to determine the most common rhythmic pattern spanning a measure. Using a monophonic `**recip` input, we could use **context** to amalgamate the appropriate data tokens:

```
context -b ^= -o ^= input.recip | rid -GLId | sort \
    | uniq -c | sort -nr
```

The output for the combined voices of Bach's two-part Invention No. 5 shows just seven patterns. The most characteristic patterns are the second one: `8r 16 16 8 8 4 4` and the fourth one:

```
8 16 16 8 8 4 4.

   30   16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16
   12   8r 16 16 8 8 4 4
   11   8 8 8 8 16 16 16 16 8 8
    8   8 16 16 8 8 4 4
    1   8 8 8 8 2
    1   8 32 32 32 32 4 2
    1   4 4 16r 16 16 16 16 16 16 16
```

The *dur* Command

The **dur** command produces `**dur` output from either a `**kern` or `**recip` input. The `**dur` representation scheme consists simply of the elapsed duration of notes and rests, expressed in seconds. The following example shows a simple `**dur` representation (right spine) with a corresponding `**kern` input:

```
**kern    **dur
*         *MM60
=1        =1
12g       0.3333
12g       0.3333
12g       0.3333
4g        1.0000
4r        1.0000r
8g        0.5000
8g        0.5000
4g        1.0000
=2        =2
*-        *-
```

As in the case of `**recip`, the `**dur` representation designates rests via the lower-case `r` and uses the common system for barlines. Notice that `**dur` assumes a metronome indication of quarter-note equals 60 beats per minute if no other metronome marking is given.

Suppose that we wanted to estimate the total duration of some monophonic passage (ignoring rubato). We can do this by translating the score to `**dur`, eliminating everything but notes and rests, and sending the output to the **stats** command:

```
dur -d inputfile.krn | rid -GLId | grep -v '^=' | stats
```

The **-d** option for **dur** suppresses the outputting of duplicate durations arising from multiple-stops. Note that outputs from **dur** will adapt to any changes of metronome marking found in the input, so if the work accelerates the durations will be reduced proportionally.

The **-M** option will over-ride any metronome markings found in the input stream. For example, if we wanted to estimate the duration of a monophonic passage for a metronome marking of 72 quarter-notes per minute we could use the command:

```
dur -M 72 -d input.krn | rid -GLId | grep -v '^=' | stats
```

Of course, the duration of a passage is not the same as the length of time a given instrument sounds. Suppose, for example, that we wanted to compare the duration of trumpet activity in the final movements of Beethoven's symphonies. We need to make a distinction between the duration of notes and the duration of rests. Since the duration values for rests are distinguished by the trailing letter 'r', we can use **grep -v** to eliminate all rest tokens.

```
extract -i '*Itromp' inputfile.krn | dur -d | rid -GLId \
    | grep -v '^=' | grep -v r | stats
```

The **dur** command provides a **-e** option that allows the user to echo specified signifiers in the output. The **-e** option is followed by a regular expression indicating what patterns are to be passed to the output. This option allows us to "mark" notes of special interest. For example, suppose we wanted to determine the longest duration note for which Mozart had marked a staccato.

```
dur -e \' inputfile | rid -GLId | grep \' | sed 's/\'//' \
    | stats
```

The **-e** option ensures that `**kern` staccato marks (') are passed along to the output. The **rid** command eliminates everything but Humdrum data records. Then **grep** is used to isolate only those notes containing a staccato mark. The **sed** script is used to eliminate the apostrophe, and finally the numbers are passed to the **stats** command. The `max` value from **stats** will identify the duration (in seconds) of the longest note marked staccato.

This same basic pipeline can be used for a variety of similar problems. Suppose, for example, that we want to determine whether notes at the ends of phrases tend to be longer than notes at the beginnings of phrases — and if so, how much longer? In this case, we want to have **dur** echo phrase-related signifiers:

```
dur -e '{' inputfile | rid -GLId | grep '{' | sed 's/{//' \
    | stats
dur -e '}' inputfile | rid -GLId | grep '{' | sed 's/{//' \
    | stats
```

Similarly, do semitone trills tend to be shorter than whole-tone trills?

```
dur -e 't' inputfile | rid -GLId | grep 't' | sed 's/{//' \
    | stats
dur -e 'T' inputfile | rid -GLId | grep 'T' | sed 's/{//' \
    | stats
```

Of course, we can also use **dur** in conjunction with **yank** in order to investigate particular musical segments or passages. How much shorter is the recapitulation compared with the original exposition?

```
yank -s 'Exposition' -r 1 inputfile | dur | rid -GLId \
    | grep -v '=' | stats
```

```
yank -s 'Recapituation' -r 1 inputfile | dur | rid -GLId \
   | grep -v '=' | stats
```

Do initial phrases in Schubert's vocal works tend to be shorter than final phrases?

```
yank -m { -r 1 lied | dur | rid -GLId | grep -v ^= | stats
yank -m { -r $ lied | dur | rid -GLId | grep -v ^= | stats
```

How much longer is a passage if all the repeats are played?

```
thru inputfile | dur | rid -GLID | stats -o ^=
```

Recall that the **xdelta** command can be used to calculate numerical differences between successive values. If the input to **xdelta** is `**dur` duration information, then we can determine rates of change of duration. Most music exhibits lengthy passages of similar duration notes — as in a sequence of sixteenth notes. In French overtures, successive notes are often of highly contrasting durations (longer, very-short, long, etc.). Using **xdelta** we can identify such large changes of duration. For example, the following pipeline can be used to determine the magnitude of the *differences* between successive notes.

```
dur inputfile | xdelta -s ^= | rid -GLId | stats -o ^=
```

A small `mean` from **stats** will be indicative of works that tend to have smoother or less angular note-to-note rhythms.

Classifying Durations

We can use the **recode** command to classify durations into a finite set of categories. Suppose, for example, we wish to create a inventory of long/short rhythmic patterns. We might use **recode** with reassignments such as the following:

```
>=0.4   long
else    short
```

For a monophonic input, we can create an inventory of (say) 3-note long/short rhythmic patterns as follows:

```
dur inputfile | recode -f reassign -i '**dur' -s ^= | \
   context -n 3 -o = | rid -GLId | sort | uniq -c | sort -n
```

A typical output might appears as follows:

```
 230 long long long
3422 short short short
 114 long long short
 202 short short long
  38 long short long
 117 short long long
 194 long short short
```

```
114short long short
```

Notice that we might do a similar inventory based on durational *differences* rather than on durations. For example, the **xdelta** command will allow us to distinguish short*er* note relationships from long*er* relationships. Our reassignment file would be as follows:

```
==0   equal
>0    shorter
<0    longer
```

And our processing would be:

```
dur inputfile | xdelta -s ^= | recode -f reassign \
   -i '**Xdur' -s ^= | context -n 2 -o = \
   | rid -GLId | sort | uniq -c | sort -n
```

Using *yank* with the *timebase* Command

Recall that the **timebase** command can be used to reformat an input so that each data record represents an equivalent elapsed duration. For example, in a 4/4 meter, the following command will format the output so that each full measure consists of precisely 16 data records (not including the barline itself):

```
timebase -t 16 input.krn
```

Suppose we wanted to isolate all sonorities in a 4/4 work that occur only on the fourth beat of a measure. If we use **timebase**, we can ensure that the fourth beat always occurs a certain number of data records following the barline. For example, with the following command, the onset of the fourth beat will always occur 4 records follow the barline:

```
timebase -t 4 input.krn
```

We can now use **yank -m** to extract all appropriate sonorities. The "marker" is the barline and the "range" is 4 records following the marker, hence:

```
timebase -t 4 input.krn | yank -m ^= -r 4
```

Note that this process will extract only those notes that begin sounding with the onset of the fourth beat. Some notes may have begun prior to the fourth beat and yet are sustained into the beat. If we want to extract the *sounded* sonority, we can use the **ditto** command. Begin by expanding the work with a timebase that ensures all notes are present. For a work whose shortest note is a 32nd note, we can use an appropriately small timebase value. Then use the **ditto** command to propagate all sustained notes forward through the successive sonorities:

```
timebase -t 32 input.krn | ditto -s ^=
```

Now we can yank the data records that are of interest. Notice that the **-r** (range) option for **yank -m** allows us to select more than one record. This might allow us, say, to extract only

those sonorities that occur on off-beats. For example, the following command extracts all notes played by the horns during beats 2 and 4 in a 4/4 meter work:

```
extract -i '*Icor' input.krn | timebase -t 16 \
    | yank -m ^= -r 5-8,13-16
```

In some cases, we would like to yank materials that do not themselves contain explicit durational information. Suppose, for example, that for a waltz repertory, we want to contrast those chord functions that tend to occur on the first beat with those that happen on the third beat. We will need to have an input that includes both a `**harm` spine encoding the Roman numeral harmonic analysis, as well as one or more `**kern` or `**recip` spines that include the durational information. We can use the **timebase** command to expand the output accordingly — cuing on the duration information provided by `**kern` or `**recip`. Having suitable expanded the input, we can dispense with everything but the `**harm` spine. For works in 3/4 meter, the following pipeline would provide an inventory of chords occuring on the first beat of each bar:

```
timebase -t 8 input | extract -i '**harm' \
    | yank -m ^= -r 1 | rid -GLId | sort | uniq -c | sort -n
```

And the following variation would provide an inventory of chords occuring on the third beat of each bar. (There are 6 eighth durations in a bar of 3/4, therefore the beginning of the third beat will coincide with the 4th eighth — hence the range `-r 4:`)

```
timebase -t 8 input | extract -i '**harm' \
    | yank -m ^= -r 4 | rid -GLId | sort | uniq -c | sort -n
```

The *metpos* Command

The **metpos** command generates a `**metpos` output spine containing numbers that indicate the metric strength of each sonority. By "metric position" we mean the position of importance in the metric hierarchy for a measure.

The highest position in any given metric hierarchy is given by the value '1'. This value is assigned to the first event at the beginning of each measure. In duple and quadruple meters, the second level in the metric hierarchy occurs in the middle of the measure and is assigned the output value '2'. (In triple meters, **metpos** assumes that the second and third beats in the measure are both assigned to the second level in the metric hierarchy.) All other metric positions in the measure (beats, sub-beats, sub-sub-beats, etc.) are assigned successively increasing numerical values according to their placement in the metric hierarchy. In summary, larger `**metpos` values signify sonorities of *lesser* metric significance.

By way of illustration, consider the case of successive eighth notes in a 2/4 meter. The metric hierarchy values for successive eighths are: 1, 3, 2, 3. In the case of successive sixteenth notes in 2/4, the metric hierarchy values are: 1,4,3,4,2,4,3,4. In the case of 6/8 meter, successive sixteenth durations exhibit a metric hierarchy of: 1,4,3,4,3,4,2,4,3,4,3,4.

For correct operation, the **metpos** command must be supplied with an input that has been for-

matted using the **timebase** command. That is, each data record (ignoring barlines) must represent an equivalent duration of time. In addition, **metpos** must be informed of both the *meter signature* and the *timebase* for the given input passage. This information can be specified via the command line, however it is usually available in the input stream via appropriate tandem interpretations.

The following extract from Bartók's "Two-Part Study" No. 121 from *Mikrokosmos* demonstrates the effect of the **metpos** command. The two left-most columns show the original input; all three columns show the corresponding output from **metpos**:

```
**kern    **kern  **metpos
*tb8      *tb8    *tb8
=16       =16     =16
*M6/4     *M6/4   *M6/4
8Gn       8b-     1
8A        8ccn    4
8B-       8cc#}   3
8cn       {8f#    4
8c#}      8gn     3
{8F#      8a      4
8G        8b-     2
8A        8ccn    4
8B-       4b-     3
8cn       .       4
8c#}      8fn}    3
8r        8r      4
=17       =17     =17
*M4/4     *M4/4   *M4/4
8d        2r      1
4.d       .       4
.         .       3
.         .       4
{2d_      8dd     2
.         4.dd    4
.         .       3
.         .       4
=18       =18     =18
8d        {1dd_   1
8A        .       4
8F#       .       3
8E        .       4
8D        .       2
8BB       .       4
8D        .       3
8E}       .       4
=19       =19     =19
*M3/2     *M3/2   *M3/2
{8F#      8dd     1
```

```
8A      8ffn    4
8c#     8aa     3
8A      8ff     4
8F#     8dd     2
8A      8ff     4
8F#     8dd     3
8E      8ccn    4
8D      8b-     2
8BBn    8gn     4
8D      8b-     3
8E}     8cc     4
=20     =20     =20
*-      *-      *-
```

Notice that **metpos** adapts to changing meter signatures, and correctly distinguishes between metric accent patterns such as 6/4 (measure 16) and 3/2 (measure 19).

The `**metpos` values provide additional ways of addressing various rhythmic questions. We might use **recode** for example, to recode the numerical outputs from **metpos** into a smaller set of discrete categories. For example, we might classify metric positions using the following reassignment file:

```
==1   strong
>=3   secondary
else  weak
```

The words 'strong', 'secondary', and 'weak' can then be sought by **grep** or **yank -m** allowing us to isolate points of particular metric stress. Since **metpos** adapts to changing meters, we can confidentaly process inputs that may contain mixtures of meters.

Changes of Stress

Once again we can make use of **xdelta** to identify relationships between successive metric position values. Suppose we had a collection of Hungarian melodies and we wanted to determine how each degree is approached in terms of metric strength. That is, we would like to count the number of tonic pitches that are approached by a weak-to-strong context versus the number of tonic pitches approached by a strong-to-weak context. We also want similar measures for supertonic, mediant, subdominant, etc. scale degrees.

This task involves creating an inventory where fourteen different items are possible: (1) tonic strong-to-weak, (2) tonic weak-to-strong, (3) supertonic, strong-to-weak, etc. A suitable inventory will involve creating two spines of information — scale-degree and relative metric strength.

Assuming that our Hungarian melodies encode key information, creating a `**deg` spine is straightforward. Recall that the **-a** option for **deg** avoids distinguishing the direction of approach (from above or below):

```
deg -a magyar*.krn > magyar.deg
```

Creating a spine encoding relative metric strength will be more involved. First we need to expand our input according to the shortest note. We use **census -k** to determine the shortest duration, and then expand our input using **timebase**.

```
census -k magyar*.krn
timebase -t 16 magyar*.krn > magyar.tb
```

Using **metpos** will allow us to create a spine with the metric position data.

```
metpos magyar.tb > magyar.mp
```

Note that **metpos** automatically echoes the input along with the new `**metpos` spine. At this point, the result might look as follows:

```
!!!OTL: Graf Friedrich In Oesterraaich sin di Gassen sou enge
**kern          **metpos
*ICvox          *
*Ivox           *
*M3/4           *M3/4
*k[f#]          *
*G:             *
*tb16           *tb16
{8g             2
.               4
8b              3
.               4
=1              =1
8dd             1
.               4
etc.
```

We want to be able to say that the relationship between the first eighth-note G and the eighth-note B is "strong-to-weak" and that the relationship between the eighth-note B and the eighth-note D is "weak-to-strong." In order to procede we need to eliminate all of the data records that contain only a metpos value — that is, there is no pitch present in the `**kern` spine. We can do this using **humsed**; we simply delete all lines that begin with a period character:

```
humsed '/^\./d' magyar.mp
```

The result is as follows:

```
!!!OTL: Graf Friedrich In Oesterraaich sin di Gassen sou enge
**kern          **metpos
*ICvox          *
*Ivox           *
*M3/4           *M3/4
*k[f#]          *
```

```
*G:             *
*tb16           *tb16
{8g             2
8b              3
=1              =1
8dd             1
etc.
```

Notice that the successive `**metpos` values will now allow us to characterize the changes in stress between successive notes: 2 followed by 3 indicates a strong-to-weak change of metric position, 3 followed by 1 indicates a weak-to-strong change of metric position. We can use **xdelta** to calculate the differences in metric position values: positive differences will indicate weak-to-strong changes and negative differences will indicate strong-to-weak changes. If both values have the same metric position value, then the successive notes hold equal positions in the metric hierarchy. Before using **xdelta** we need to isolate the `**metpos` spine using **extract**:

```
humsed '/^\./d' magyar.mp | extract -i '**metpos' \
    | xdelta -s ^=
```

The result is:

```
!!!OTL: Graf Friedrich In Oesterraaich sin di Gassen sou enge
**Xmetpos
*
*
*M3/4
*
*
*tb16
.
1
=1
-2
etc.
```

Now we can use **recode** to classify the changes of metric position according. Our reassignment file (named `reassign`):

```
>0    strong-to-weak
<0    weak-to-strong
==0   equal
```

Appending the appropriate command:

```
humsed '/^\./d' magyar.mp | extract -i '**metpos' \
    | xdelta -s ^= | recode -f reassign -i '**Xmetpos'  -s
    ^= > magyar.xmp
```

Now we can assemble the resulting metric change spine with our original **deg spine. Each data record will contain the scale degree in the first spine and the change of metric position data in the second spine. The final task is to create an inventory using **rid**, **sort** and **uniq**:

```
assemble magyar.deg magyar.xmp | rid -GLId | grep -v ^= \
   | sort | uniq -c
```

The final result will appear as below. The first output line indicates that there were three instances of a tonic pitch approached by a note of equivalent position in the metric hierarchy. The second line indicates that there were twenty-five instances of a tonic pitch approached by a note having a stronger metric position:

```
 3   1   equal
25   1   strong-to-weak
30   1   weak-to-strong
 3   2   equal
14   2   strong-to-weak
13   2   weak-to-strong
 1   3   equal
39   3   strong-to-weak
34   3   weak-to-strong
 3   4   equal
26   4   strong-to-weak
17   4   weak-to-strong
13   5   equal
49   5   strong-to-weak
42   5   weak-to-strong
 1   6   equal
13   6   strong-to-weak
14   6   weak-to-strong
 3   7   strong-to-weak
 6   7   weak-to-strong
 1   7-  weak-to-strong
 3   r   equal
10   r   strong-to-weak
```

Instead of scale degree, any other Humdrum spine might be used. For example, if the input contained functional harmony data (**harm) then the output inventory would identify how particular chord functions tend to be approached. For example, we could establish whether the submediant chord is more likely to be approached in a strong-to-weak or weak-to-strong rhythmic context. Similarly, this same technique can be used to determine whether particular melodic or harmonic intervals tend to be approached using particular stress relationships.

In addition, our input spine might also be transformed via the **context** command. Given a **harm spine, for example, **context** could be used to generate two-chord harmonic progressions. This would permit us to determine, for example, whether a specific progression such as *ii-V* tends to fall in strong-to-weak or weak-to-strong contexts.

Reprise

There are a vast number of issues raised in rhythm-related processing. In this chapter we have touched on a few of the more basic tasks. These include identifying the durations of various passages using **dur**; classifying and contextualizing durations using **recode** and **context**; isolating particular rhythmic moments using **timebase** and **yank -m**; determining relative metric positions using **metpos**; and characterizing metric syncopation using **synco**.

Processing data that does not explicitly contain duration-related information (such as `**harm` or `**deg`) often requires some preparation. It is often useful to maintain a coordinated file where the spines of interest are linked with duration-related spines that assist in processing.

One further topic related to rhythm remains to be discussed. The **accent** command allows the user to distinguish notes according to their estimated perceptual importance. We will consider **accent** in Chapter 31.

Chapter 24

The Shell (III)

In Chapter 16 we learned about the **alias** feature of the shell. The **alias** command allowed us to create new commands by assigning a complex pipeline to a single-word command. In this chapter we will learn how to use the shell to write more complex programs. Shell programs allow users to reduce lengthy sequences of Humdrum commands to a single user-defined command.

Shell Programs

A shell program is simply a script consisting of one or more shell commands. Suppose we had a complex procedure consisting of a number of commands and pipelines:

```
extract -i '**Ursatz' inputfile | humsed '/X/d' \
    | context -o Y -b Z > Ursatz
extract -i '**Urlinie' inputfile | humsed '/X/d' \
    | context -o Y -b Z > Urlinie
assemble Ursatz Urlinie | rid -GLId | graph
```

In the above hypothetical script, we have processed an input file called `inputfile`. It may be that this is a procedure we would like to apply to several different files. Rather than typing the above command sequence for each file, an alternative is to place the above commands in a file. Let's assume that we put the above commands in a file called `Schenker`. In order to execute this file as a shell script, we need to assign *execute permissions* to the file. We can do this by invoking the UNIX **chmod** command.

```
chmod +x Schenker
```

The **+x** option causes **chmod** to add execute permissions to the file `Schenker`. Using **chmod** we can change modes related to *executing* a file, *reading* a file, and *writing* to a file. Possible mode changes include the following:

+x	add execute permission
-x	deny execute permission
+r	add read permission
-r	deny read permission

+w add write permission
-w deny write permission

Having added execute permissions to the file, we can now execute the shell script or program. This is done simply by typing the name of the file; in effect, the filename becomes a new command:

```
Schenker
```

Each time we type this command, our script will be executed anew. Notice that in our script, the final output has not been sent to a file. As a result, the output from our **Schenker** command will be sent to the screen (standard output). It is convenient not to specify an output file in the script since this is often something the user would like to specify. When typing our new command, we can use file-redirection to place the output in a user-specified file:

```
Schenker > outputfile
```

As currently written, our program can be applied only to an input file whose name is literally `inputfile`. If we wanted to, we could edit our script and up-date the name of the input filename every time we want to use the command. However, it would be more convenient to specify the input filename on the command line — as we can do for other commands. For example, it would be convenient to be able to type commands such as the following:

```
Schenker opus118 > opus118.out
```

In order to allow such a possibility, we can use a predefined feature of the shell. Whenever the shell receives a command, each item of information on the command line is assigned to a shell variable. The first item on the command line is assigned to the variable $0 (normally, this is the command name). For example, in the above example, $0 is assigned the string value "`Schenker`." The variables $1, $2, $3, etc. are assigned to each successive item of information on the command line. So in the above example, $1 is assigned the string value "`opus118`."

These shell variables can be accessed within the shell script itself. We need to revise the script so that each occurrence of the input file is replaced by the variable $1:

```
extract -i '**Ursatz' $1 | humsed '/X/d' \
    | context -o Y -b Z > Ursatz
extract -i '**Urlinie' $1 | humsed '/X/d' \
    | context -o Y -b Z > Urlinie
assemble Ursatz Urlinie | rid -GLId | graph
```

This change means that our **Schenker** command can be applied to any user-specified input file — simply by typing the filename in the command.

Flow of Control: The *if* Statement

Suppose we wanted our **Schenker** command to apply only to tonal works — more specifically, to works with a known key. Before processing a work, we might want to have **Schenker** test for the

presence of a tandem interpretation specifying the key.

Let's begin by using **grep** to search for a key tandem interpretation. An appropriate **grep** command would be:

```
grep '^\*[A-Ga-g][#-]*:' $1
```

Recall that we can assign the output of any command to a shell variable by placing the command within back-quotes or greves, i.e. `'...'`. Let's assign the key interpretation to the variable KEY:

```
KEY=`grep '^\*[A-Ga-g][#-]*:' $1`
```

If no key indicator is found by **grep**, then the variable KEY will be empty. We can test for this condition using the shell **if** statement.

```
KEY=`grep '^\*[A-Ga-g][#-]*:' $1`
if [ "$KEY" = "" ]
then
          echo "Sorry, this input file has no key."
          exit
fi
```

Notice that we use the dollars sign prior to the variable to mean "the contents of variable KEY". The double quotation marks allow a string comparison. Our test is whether the variable $KEY is equivalent to the empty or null string "". If the test is true, then the commands following the `then` statement are executed. By convention, these commands are indented for clarity. In the above case, two commands are executed if the $KEY variable is empty. The **echo** command causes the quoted string to be output. The **exit** command causes the script to terminate. Notice the presence of the **fi** command (**if** backwards). This command simply indicates that the if-block has ended.

Of course, if there is a key designation, then it is appropriate to execute the rest of our **Schenker** script. The complete script would be as follows:

```
KEY=`grep '^\*[A-Ga-g][#-]*:' $1`
if [ "$KEY" = "" ]
then
          echo "Sorry, this input file has no key."
          exit
else
          extract -i '**Ursatz' $1 | humsed '/X/d' \
             | context -o Y -b Z > Ursatz
          extract -i '**Urlinie' $1 | humsed '/X/d' \
             | context -o Y -b Z > Urlinie
          assemble Ursatz Urlinie | rid -GLId | graph
fi
```

Notice the addition of the **else** statement. The **else** statement delineates the block of commands to be executed whenever the **if** condition fails — that is, when the $KEY variable does *not* equal the

null string. Once again, to make the script more readable, we indent the commands contained in the else-block.

The **if** command provides many other ways of testing some condition. For example, the shell provides ways to determine whether a file exists, and other features.

Flow of Control: The *for* Statement

In music research, a common task is to apply a particular process or script to a large number of score files. By way of illustration, suppose we wanted to know the maximum number of notes in any single folk melody in a collection of Czech folksongs. Suppose further that we are located in a directory containing a large number of Czech folksongs named `czech01.krn`, `czech02.krn`, `czech03.krn`, and so on.

We would like to run the **census -k** command on each file separately, but we'd prefer not to type the command for each score. The **for** statement provides a convenient way to do this. The following commands might be typed directly at the shell:

```
for J in czech*.krn
> do
> census -k $J | grep 'Number of notes:'
> done | sort -n
```

The pattern `czech*.krn` will be expanded to all of the files in the current directory that it matches. The variable **J** will take on each name in turn. The commands between **do** and **done** will be executed for each value of the variable **$J**. That is, initially **$J** will have the value `czech01.krn`. Having completed the do-done block of commands, the value of **$J** will become `czech02.krn`, and the do-done block will be repeated. This will continue until the value of **$J** has taken on all of teh possible matches for `czech*.krn`.

The output might appear as follows:

```
Number of notes:          31
Number of notes:          32
Number of notes:          32
Number of notes:          34
Number of notes:          35
Number of notes:          39
Number of notes:          39
Number of notes:          40
Number of notes:          48
Number of notes:          48
Number of notes:          55
Number of notes:          78
etc.
```

Incidentally, the output from a **for** construction such as above can be piped to further commands, so we might identify the maximum number of notes in a Czech melody by piping the output

through **sort -n**.

A Script for Identifying Transgressions of Voice-Leading

Shell programs can be of arbitrary complexity. Below is a shell program (dubbed **leader**) whose purpose is to identify all instances of betrayals of nine classic rules of voice-leading for a two-part input. A number of refinements have been added to the program — including input file checking, and formatting of the output.

The program is invoked as follows:

```
leader <file>
```

The input is assumed to contain two voices, each in a separate `**kern` spine. The nominally lower voice should be in the first spine. For music containing more than two voices, the Humdrum **extract** command should be used to select successive pairs of voices for processing by **leader**.

```
# LEADER
#
# A shell program to check for voice-leading infractions.
# This command is invoked as:
#
#   leader <filename>
#
# where <filename> is assumed to be a file containing two voices, each
# in a separate **kern spine, where the nominally lower voice is in the
# first spine.

# Before processing, ensure that a proper input file has been specified.
if [ ! -f $1 ]
then   echo "leader: file $1 not found"
       exit
fi
if [ $# -eq 0 ]
then   echo "leader: input file not specified"
       exit
fi

# 1. Record the ranges for the two voices.
echo 'Range for Upper voice:'
extract -f 2 $1 | census -k | egrep 'Highest|Lowest' | sed 's/^/      /'
echo 'Range for Lower voice:'
extract -f 1 $1 | census -k | egrep 'Highest|Lowest' | sed 's/^/      /'

# 2. Check for augmented or diminished melodic intervals.
extract -f 1 $1 | mint -b r | sed '/\[[Ad][Ad]*\]/d' | egrep -n '^[^!*].*[Ad][^1]' |\
    sed 's/:/ (/;s/$/)/;s/^/Augmented or diminished melodic interval at line: /'
extract -f 2 $1 | mint -b r | sed '/\[[Ad][Ad]*\]/d' | egrep -n '^[^!*].*[Ad][^1]' |\
    sed 's/:/ (/;s/$/)/;s/^/Augmented or diminished melodic interval at line: /'

# 3. Check for consecutive fifths and octaves.
```

```
echo 'P5'  > $TMPDIR/template;  echo 'P5'  >> $TMPDIR/template
hint -c $1 | patt -f $TMPDIR/template -s = | \
    sed 's/ of file.*/./;s/.*Pattern/Consecutive fifth/'
echo 'P1'  > $TMPDIR/template;  echo 'P1'  >> $TMPDIR/template
hint -c $1 | patt -f $TMPDIR/template -s = | \
    sed 's/ of file.*/./;s/.*Pattern/Consecutive octave/'

# 4. Check for doubling of the leading-tone.
deg $1 | extract -i '**deg' | ditto -s = | sed 's/^=.*/=/' | \
    egrep -n '^7.*7|^[^!*].*7.*7' | egrep -v '7[-+]' | \
    sed 's/:.*/./;s/^/Leading-tone doubled at line: /'

# 5. Check for unisons.
semits -x $1 | ditto -s = | \
    awk '{if($0~/[^0-9\t-]/)next}{if($1==$2) print "Unison at line: " NR}'

# 6. Check for the crossing of parts.
semits -x $1 | ditto -s = | sed 's/^=.*/=/' | \
    awk '{if($0~/[^0-9\t-]/)next}{if($1>$2) print "Crossed parts at line: " NR}'

# 7. Check for more than an octave between the two parts.
semits -x $1 | ditto -s = | awk '{if($0~/[^0-9\t-]/)next} \
    {if($2-$1>12) print "More than an octave between parts at line: " NR}'

# 8. Check for overlapping parts.
extract -f 2 $1 | sed 's/^=.*/./' | context -n 2 -p 1 -d XXX | \
    rid -GL | humsed 's/XXX.*//' > $TMPDIR/upper
extract -f 1 $1 | sed 's/^=.*/./' > $TMPDIR/lower
assemble $TMPDIR/lower $TMPDIR/upper | semits -x | ditto | \
    awk '{if($0~/[^0-9\t-]/)next}{if($1>$2) print "Parts overlap at line: " NR}'
extract -f 1 $1 | sed 's/^=.*/./' | context -n 2 -p 1 -d XXX | \
    rid -GL | humsed 's/XXX.*//' > $TMPDIR/lower
extract -f 2 $1 | sed 's/^=.*/./' > $TMPDIR/upper
assemble $TMPDIR/lower $TMPDIR/upper | semits -x | ditto | \
    awk '{if($0~/[^0-9\t-]/)next}{if($1>$2) print "Parts overlap at line: " NR}'

# 9. Check for exposed octaves.
hint -c $1 > $TMPDIR/s1
extract -f 1 $1 | deg > $TMPDIR/s2
extract -f 2 $1 | deg > $TMPDIR/s3
extract -f 1 $1 | mint | humsed 's/.*[3-9].*/leap/' > $TMPDIR/s4
extract -f 2 $1 | mint | humsed 's/.*[3-9].*/leap/' > $TMPDIR/s5
assemble $TMPDIR/s1 $TMPDIR/s2 $TMPDIR/s3 $TMPDIR/s4 $TMPDIR/s5 > $TMPDIR/temp
egrep -n 'P1.*\^.*\^.*leap.*leap|P1.*v.*v.*leap.*leap' $TMPDIR/temp | \
    sed 's/:.*/./;s/^/Exposed octave at line: /'

# Clean-up some temporary files.
rm $TMPDIR/template $TMPDIR/upper $TMPDIR/lower $TMPDIR/s[1-5] $TMPDIR/temp
```

Reprise

In this chapter we have illustrated how to package complex Humdrum command scripts into shell programs. This allows us to create special-purpose commands. We learned that files can be trans-

formed into executable scripts through the **chmod** command. We also learned how to pass parameters from the command line to the script, and how to assign and modify the contents of variables. In addition, we learned how to influence the flow of control using the **if** and **for** statements. Finally, we learned that multi-line scripts can be typed directly at the command line without creating a script file.

Shell scripts can be very brief or very long. It is possible to create scripts that carry out highly sophisticated processing such as searching for voice-leading transgressions. There are innumerable features to shell programming that have not been touched-on in this chapter. Several books are available that provide comprehensive tutorials for shell programming.

Chapter 25

Similarity

There is no precise way to measure similarity. However, there are several useful techniques that can be used to estimate the degree of similarity between different types of information. In this chapter we discuss two general tools for characterizing similarity: **correl** and **simil**. The **correl** command can be used to measure numerical similarity between two sets of numbers. The **simil** command can be used to measure similarity between non-numeric data.

In addition, we will discuss the **accent** command — a tool which estimates how salient or noticeable a given note is. The **accent** command can be used to pre-process musical passages so only those notes of greatest importance are considered when measuring musical similarity.

The *correl* Command

One way of measuring similarity is to compare the rise and fall of two sets of numbers. Suppose, for example, that we wanted to determine whether high pitches have a general tendency to be longer in duration than low pitches. For each note we would establish two numerical values: one characterizing the pitch height and one characterizing the duration (say in seconds). Our data might look as follows:

```
**semits    **dur
7           1.00
16          1.50
14          0.50
12          2.00
7           1.00
4           0.50
5           0.50
7           1.00
*-          *-
```

This data does seem to exhibit an association between higher pitches and longer notes. The longest notes are fairly high (12 and 16 semits), whereas most of the shortest notes (4 and 5 semits) are lower. There are some exceptions, however, such as the 0.5 sec. duration for a pitch of 14 semits.

The Humdrum **correl** command allows us to characterize more precisely the degree of similarity

between two sets of numbers. The **correl** command expects precisely two input spines; it is easily invoked:

```
correl inputfile
```

For the above semitone/duration data, **correl** will output the value +0.515.

Technically, the **correl** command calculates Pearson's coefficient of correlation between two spines containing numerical data. Correlation coefficients range between +1 and −1. A value of +1 indicates that both sets of numbers rise and fall in precise synchrony — although the magnitude of the numbers may differ. For example, the following input exhibits a correlation of +1.0 — even though the two sets of numbers differ in overall magnitude.

```
**foo   **bar
1       100
3       300
2       200
1       100
*-      *-
```

If we multiply these numbers by a constant, or if we substract or add a constant value to each number in one of the spines, they would still exhibit a correlation of +1.0. In summary, correlations are insensitive to the absolute magnitude and offsets for different sets of numbers.

A correlation coefficient of −1 means that the rise and fall of numerical values are exactly reversed. When one set of numbers is rising, the other set is falling — and vice versa. By contrast, a correlation coefficient of zero means that the two sets of numbers are statistically independent of each other. For example, comparing two large sets of random numbers will result in a correlation coefficient near zero.

The **correl** command attends only to numerical input data. Non-numerical data is simply ignored. If a data token contains a mix of numeric and non-numerica characters, then only the first complete numerical subtoken is considered. The following examples illustrate how **correl** interprets mixed data tokens:

Table 23.1

token	interpretation
4gg#	4
4.gg#	4
-32aa	-32
-aa33	33
x7.2yz	7.2
a7..2bc	7
[+5]12	5
$17@2	17
=28b	28
a1b2 c.3.d	1 0.3

Numerical interpretations of data tokens by **correl**.

Notice in the last example that multiple-stops are treated as potentially independent numbers. For example, if the data token encodes a double-stop, then **correl** will determine whether both subto-

kens can be interpreted numerically.

In normal operation, the **correl** command expects numerical data to be precisely matched in both input spines. That is, if a particular data record contains no numbers in the left spine, it should also contain no numbers in the right spine. Similarly, if the left spine contains three numbers (in a triple stop) then the right spine must also contain three numbers in the same record. If there is any breech of the criterion of number pairing, **correl** will issue an error message and stop.

Suppose we had a passage of two-part, first species counterpoint and we were interested in whether the two voices tend toward contrary and oblique motion rather than parallel and similar motion. In first species counterpoint, each pitch in the upper voice is matched with a pitch in the lower voice. We could measure the pitch-related correlation between the two parts as follows:

```
semits species1.krn | correl -s ^=
```

The output will consist of a single numerical value. If the value is positive, then it indicates that the parts tend to move up and down together in pitch. That is, a positive correlation indicates a preponderance of parallel and similar contrapuntal motion. Conversely, a negative correlation would indicate a preponderance of contrary and oblique motion.

Notice the use of the **-s** option in the above command. Since common system barlines often contain measure numbers (e.g. =28), they are interpretable as numeric data. For most inputs, the user will not want to have measure numbers participate in the similarity calculation. The **-s** option allows the user to specify a regular expression indicating data records to skip.

Now suppose that we wanted to measure a similar pitch-related correlation for a passage of second species counterpoint. In second species counterpoint, there are two pitches in the upper voice for each pitch in the lower voice. Translating our pitch data to semitones will result in a failure of the matched-pairs criterion. There are two ways of overcoming this problem. One method is to use **ditto** to repeat the sustained semitone value for the slower-moving part:

```
semits species2.krn | ditto -s ^= | correl -s ^=
```

Another approach would be to omit from consideration those notes that are not concurrent with a note in the other voice. The **-m** option for **correl** disables the matched-pairs criterion. That is, if numerical data is missing from either one of the input spines, **correl** will simply discard the entire data record from the correlation calculation. Using this approach, we would omit the **ditto** command:

```
semits species2.krn | correl -m -s ^=
```

Note that in formal statistical tests, the **-m** option should never be used.

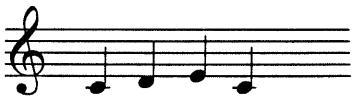
Using a Template with *correl*

In the above examples, **correl** generates a single output value indicating the degree of numerical similarity between two spines. A more valuable use of **correl** involves scanning a spine for portions that are similar to a brief excerpt or template. In this mode of operation, the input consists of

a single input spine plus a separate template that represents a pattern being sought.

The **-f** option for **correl** allows the user to specify a file that acts as a template which is then scanned across some input. By way of example, suppose we are looking for motivic instances similar to the first four notes of *Frère Jacques*. Our template file might look as follows:

```
**semits
0
2
4
0
*-
```



We would like to scan an entire work looking for possible matches or similar passages. The following example shows a sample input and corresponding output — given the above template. The left-most spine is the original input represented using the French `**solfg` scheme. The middle spine is the input (translated to `**semits`) supplied to the **correl** command. The right-most spine was generated using the following command:

```
correl -s ^= -f template jack.bro
```

```
**solfg     **semits    **correl
=1          =1          =1
do          0           1.000
re          2           -0.500
mi          4           -0.866
do          0           0.866
=2          =2          =2
do          0           1.000
re          2           -0.500
mi          4           0.000
do          0           0.945
=3          =3          =3
mi          4           0.982
fa          5           -0.327
so          7           -0.655
=4          =4          =4
mi          4           0.982
fa          5           .
so          7           .
*-          *-          *-
```

The similarity values generated by **correl** are output as a `**correl` spine. Each successive value in the output spine is matched with a data token in the target input file (`**semits`). For example, the initial output value (1.000) indicates that an exact positive correlation occurs between the template and the input. Another exact positive correlation occurs at the beginning of measure 2. More interesting, perhaps, are the high correlations (+0.982) at the beginning of measures 3 and 4. Although the semitone patterns differ (do, re, mi = +2 +2 semits; mi, fa, so = +1 +2 semits), the correlations remain high because of the approximate numerical similarity. This property gives **correl** a certain flexibility when searching for melodic similarity.

For more sophisticated melodic similarity searches, both pitch and rhythm might be considered. Two different correlations can be calculated — one for semitone contour similarity and one for durational similarity. We can generate two `**correl` spines as follows. First generate `**semits` and `**dur` data so our inputs to **correl** are numerical.

```
semits inputfile > temp.sem
dur inputfile > temp.dur
```

Generate independent `**correl` spines for the semitone pitch and duration data, and assemble the two spines together:

```
correl -s ^= -f template.sem temp.sem > correl.sem
correl -s ^= -f template.dur temp.dur > correl.dur
assemble correl.sem correl.dur
```

The resulting output consists of two `**correl` spines: one tracing the moment-by-moment pitch similarity, and the other tracing the moment-by-moment duration similarity. The output might appear as follows:

```
**correl    **correl
0.438       0.284
-0.118      0.226
0.487       -0.008
0.606       0.377
0.733       0.648
0.514       0.400
0.555       0.013
0.320       -0.158
-0.145      -0.160
```

There are various ways of combining the pitch and duration data to create a composite similarity measure. For example, one might sum together the correlations on each line: passages that exhibit high pitch/duration similarity will tend to have a large positive summed score. Alternatively, one might set a threshold for both each of the pitch and duration correlation coefficients and use **recode** to mark promising points of high correlation. Values between +0.8 and +1.0 might be recoded as "similar"; values between +0.5 and +0.8 might be recoded as "maybe"; all other values might be recoded as null tokens. Assembling the recoded `**correl` spines, one could use **grep** to search for moments in the score that are suitable marked as "similar" for both pitch and duration.

Finally, a word of caution is in order regarding the use of the **correl** command. Correlation coefficients indicate only the magnitude of the association between two sets of data. High correlation values can occur purely by chance. In particular, the noteworthiness (statistical significance) of a correlation value depends on the number of input values given in the template. Longer templates reduce the likelihood of spurious positive correlations. However, longer templates can also reduce the likelihood of discovering points of true similarity.

The *simil* Command

The problem of measuring similarity entails two questions: the *criterion* of similarity and the *metric* of similarity.

First, what is the criterion of similarity? A bassoon is similar to a *cor anglais* in tone color, however a bassoon is more similar to a 'cello in pitch range. Moreover, the word "bassoon" is more similar in spelling to "baboon" than either "'cello" or "cor anglais." The second question is how do we characterize the "distance" between objects? How *much* is the difference in pitch range between a 'cello and a bassoon? How *much* is the difference in spelling between "bassoon" and "baboon"?

In the **correl** command, the criterion of similarity arises from the user's choice of input representations. If the input represents duration, then the results pertain to durational similarity. If the input represents frequency, then the results pertain to frequency similarity. The *metric* used by **correl** is a linear numerical correlation. Since **correl** can deal only with numerical data, it is referred to as "parametric" method for measuring similarity. However, we know that non-numerical data can also be similar. An "apple" is more similar to an "orange" than it is to a "bassoon."

The **simil** command is a "non-parametric" tool for characterizing similarity. Like **correl,** the criterion of similarity depends on the user's choice of input representations. If the input represents metric position, then the results pertain to metric-position similarity. If the input represents phonetic text, then the results pertain to phonetic similarity, etc.

The *metric* used by **simil** is a so-called "edit distance" metric. The degree of similarity is characterized by how much modification would be required to transform one representation to another. By way of example, consider the spelling of the words "bassoon" and "baboon." Suppose we are allowed the following operations: (1) insertion of a character, (2) deletion of a character, and (3) substitution of a character. We can transform "bassoon" to "baboon" by deleting a letter 's' and substituting the letter 'b' for the remaining letter 's'. If each edit operation was assigned a "penalty" value of 1.0, then we would say that the edit-distance between "bassoon" and "baboon" is 2.0.

Before we describe **simil** in detail, let's examine some sample inputs and outputs. Two inputs are required by **simil** — the *source* and *template* inputs. Both inputs must contain single columns of data; multi-column inputs are forbidden. The *source* input must conform to the Humdrum syntax, however the *template* should contain only data records.

Depending on the mode of operation, **simil** outputs either one or two spines of continuous information regarding the similarity of the two inputs. The length of **simil**'s output matches that of the *source* file.

The following example illustrates the operation of **simil**. Like **correl**, **simil** provides a template mode where a relatively short template is scanned over a source input. In the following example, the source input is given in the left-most spine (labelled `**foo`) and is held in a file named `source`; the middle column consists of the letters A, B and C, and is held in a file named `template`. The following command:

```
simil source template
```

generates the third column (labelled `**simil`):

(source input	(template input)	(simil output)
`**foo`	A	`**simil`
X	B	0.51
A	C	1.00
B		0.51
C		0.37
D		0.51
A		0.72
B		0.72
B		0.51
C		0.51
B		.
A		.
*-		*-

Each succesive value in the output spine is matched with a data token in the source input file. For example, the second value (1.00) in the `**simil` spine arises from an exact match of the (A,B,C) pattern beginning with the second data token in the source input. The second highest value (0.72) occurs in both the sixth and seventh `**simil` data records, indicating that fairly similar sequences occur beginning with the sixth and seventh data records in the source input. Specifically, **simil** has recognized that the sequence (A,B,B,C) is only one edit-operation (a deletion) different from the template (A,B,C). In the ensuing record, **simil** has recognized that the sequence (B,B,C) is also only one edit-operation (substitute A for B) different from (A,B,C). Notice that the final value (0.51) indicates that the edit distance for (C,B,A) is less like the template. Also notice that the lowest value (0.37) corresponds to an input pattern (beginning D,D,A) that bears little resemblance to the template.

A musically more pertinent example is given below. Here our template consists of a harmonic pattern: *I-IV-V-I.*

(source input	(template input)	(simil output)
`**harm`	I	`**simil`
I	IV	0.87
vi	V	0.87
ii7	I	0.51
V		0.38
V7		0.41
I		0.82
r		0.41
V/V		0.38
V		.
iii		.
iiib		.
*-		*-

It is important to understand that **simil** operates by comparing entire data tokens, so the token V7 differs as much from V as the token vi. It is the user's responsibility to choose an input represen-

tation that facilitates recognition of interchangeable or equivalent data. For example, in the follow example, the harmonic data given above has been reclassified (using **humsed**) so that the number of distinct harmonic categories has been reduced. For example, the `ii7` chord has been classified as a form of `subdominant` function. Notice how the `**simil` values better reflect the presumed harmonic similarity:

```
(source      (template    (simil
input        input)       output)
**Harm       tonic        **simil
tonic        subdom       0.92
subdom       dom          0.90
subdom       tonic        0.87
dom                       0.44
dom                       0.41
tonic                     0.83
r                         0.66
secondary                 0.41
dom                       .
mediant                   .
mediant                   .
*-                        *-
```

Defining Edit Penalties

Technically, the **simil** command implements a Damerau-Levenshtein metric for edit distance (see Orpen & Huron, 1992). Permissible edit operations include substitutions and deletions. Each edit action incurs a penalty, and the cumulative edit-distance determines the similarity.

In the default operation, **simil** assigns equivalent edit penalties (1.0) for deletions and substitutions. However, the user can explicitly define these penalties via an initialization file. The initialization file must be named `simil.rc` and be located in the current directory or the user's home directory. Arbitrary costs may be assigned to any of eight edit operations shown in Table 23.2.

Table 23.2

Name Tag	Edit Operation
D1	Delete a non-repeated token in String 1
D2	Delete a non-repeated token in String 2
R1	Delete a repeated token in String 1
R2	Delete a repeated token in String 2
S0	Substitute a token that is repeated in neither String 1 nor String 2
S1	Substitute a token that is repeated in String 1 only
S2	Substitute a token that is repeated in String 2 only
S3	Substitute a token that is repeated in String 1 and String 2

Edit operations used by **simil**.

In describing the edit operations, String 1 is the source string and String 2 is the template string. Notice that there is no overt edit operation for insertion: an insertion in String 1 is equivalent to a deletion in String 2. However, different edit penalties may be defined for deletions from String 1 (D1) compared with deletions from String 2 (D2). In musical applications defining such asymmetrical penalties may be important. For example, two inputs may represent a basic melody and an

embellished variant of the melody. Using asymmetrical penalties allows the user to specify that the deletion of tones from the embellished version is less costly than deletion of tones from the basic melody.

Since repetition is a common form of musical variation, **simil** allows the user to distinguish between repeated and non-repeated tokens. A repeated token is defined as one that is immediately preceded by an identical token. Thus, in deleting a sequence of identical symbols in String 1, say, all deletions except the first occurrence are R1 operations, whereas the deletion of the first occurrence is a D1 operation.

Note that the minimum theoretical edit-distance for any set of penalty weightings can be determined empirically by providing **simil** with source and template strings that share no symbols in common. For example, the source input may consist entirely of numbers, whereas the template input consists entirely of alphabetic characters. In the case where all edit operations are assigned a penalty of +1.0, the minimum quantitative similarity between two strings is 0.37.

Some user-defined weightings may give rise to peculiar results — such as negative costs — but **simil** does not forbid this. **Simil** generates warning messages if the weightings seem illogical; for example, if the cost of R1 is more than that of D1. In addition, **simil** will abort operation if the defined edit penalties transgress the triangular inequality rule (see Orpen & Huron, 1992). The default weighting for all operations is +1.0.

Below is a sample initialization file that defines the R1 substitution as having an edit penalty of 0.7, whereas the R2 substitution is given a penalty of 0.9. Edit penalties are defined by specifying the operation, followed by some spaces or tabs, followed by some real number. Since no other penalties are defined in this file, the remaining edit operations use the default edit penalty of +1.0. The user can effectively disable a given edit operation by defining an arbitrarily high edit penalty.

```
# This is a comment.
R1          0.7
R2          0.9
```

Raw edit-distance scores are normally unreliable estimates of similarity, unless the length of the template is considered. For example, 3 editing operations constitutes a rather modest change for a template consisting of 20 elements. However, 3 edit operations is significant for a template consisting of only 5 elements. As a result, in the default operation, **simil** scales the edit-distance scores according to the length of the comparison template. This ensures that all similarity values remain between 0 and 1.

Now that we better understand the operation of simil, let's return to our analysis of the harmonic data illustrated above. It might be argued that *changing* a chord function is more dissimilar than *repeating* a chord function. In the following `simil.rc` file, an increased penalty has been assigned for dissimilar substitution, and decreased penalties have been assigned for repetition.

```
S0   1.6
S1   0.7
S3   0.7
```

Repeating the above command with this new `simil.rc` file produces the following results:

```
(source      (template   (simil
input        input)      output)
**Harm       tonic       **simil
tonic        subdom      0.94
subdom       dom         0.91
subdom       tonic       0.87
dom                      0.45
dom                      0.41
tonic                    0.84
r                        0.68
secondary                0.42
dom                      .
mediant                  .
mediant                  .
*-                       *-
```

Notice that the similarity measure for the pattern (tonic, subdom, subdom, dom, dom, tonic) has increased from 0.91 to 0.94.

The **simil** command can be used to characterize innumerable types of similarity. Suppose, for example, that we wanted to identify similar fingering patterns in music for guitar. Consider the following work by Ferdinando Carulli:

```
!!!COM: Carulli, Ferdinando
!!!OTL: Larghetto, Opus 124, No. 23
!! For guitar.
**fret
*ICstr
*Iguitr
*AT:E2
*RT:0:5:10:15:19:24
*MM60
: : : : |0M :
=1
|0P : : |1bI : |0A
: : : |1bI : |0A
|0P : : |2bI : |2bA
: : : |4dI : |4eA
: : : |2bI : |2bA
=2
|0P : : |1bI : |0A
: : : |1bI : |0A
|0 : : |2bI : |2bA
: : : |4dI : |4eA
: : : |2bI : |2bA
=3
```

We might be interested in a fret-board fingering pattern that consists of the following successive finger combinations:

index finger
index finger
ring and little fingers
index finger

In order to search for similar fingering patterns, we need to eliminate all but the relevant information from our representation. In the `**fret` scheme, fret-board fingerings are indicated by the lower-case letters *a* to *e* (*a*=thumb, *b*=index finger, *c*=middle finger, etc.). The lower-case *n* is used to explicitly indicate no finger (i.e. open string(s)). We can prepare our input using the following **humsed** command. We delete all barlines, and then eliminate all characters other than the letters *a* to *e*. Any resulting empty lines we replace by the letter *n*.

```
grep -v ^= carulli | humsed 's/[^a-e]//g; s/^$/n/' carulli
```

The corresponding output would be as follows:

```
!!!COM: Carulli, Ferdinando
!!!OTL: Larghetto, Opus 124, No. 23
!! For guitar.
**fret
*ICstr
*Iguitr
*AT:E2
*RT:0:5:10:15:19:24
*MM60
n
b
b
bb
de
bb
b
b
bb
de
bb
```

The appropriate template file would contain the following finger successions:

```
b
b
de
b
```

The *accent* Command

Both the **correl** and **simil** tools presume that all data tokens are equally important. In the case of **correl**, each number is weighted equally in calculating the coefficient of correlation. In the case of **simil**, each data token has the same potential for disrupting the similarity measure.

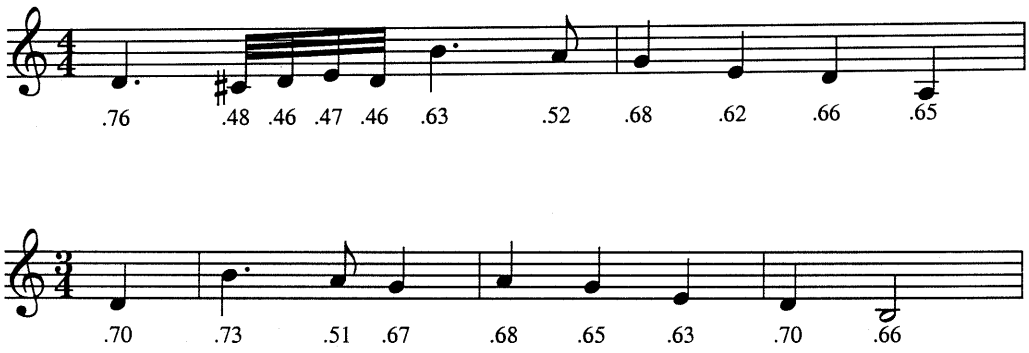
In musical circumstances, we are aware that not all notes are equally important. Some notes are more perceptually more noticeable. The effectiveness of both **correl** and **simil** can be increased significantly if we first "filter" our data — selecting only the most important items of data for consideration.

The **accent** command implements a sophisticated model of the perceptual salience or noticeability for various pitches. The command accepts only monophonic `**kern` input and outputs a spine containing numerical values estimating the noticeability of each note. Output accent values vary between 0 (minimum accent) and 1 (maximum accent). Input is limited to only a single `**kern` data spine.

The **accent** command takes into account seven factors: (1) the duration of notes (agogic stress), (2) the amount of melodic (or pitch-related) accent, (3) metric position, (4) position in scale-degree hierarchy, (5) primacy/recency contexts, (6) explicit accent/articulation marks, and (7) inner-voice or outer-voice position. No attempt is made to account for melodic expectancy, past experience, or other factors known to influence the perceptual salience of particular notes.

By way of illustration, consider the two passages shown in Example 25.1: from Wagner's *Rienzi* opera, and the Scottish folksong **My Bonnie**. Two sample outputs from **accent** are given below. In both examples the left-most spine shows the input, and the right-most spine shows the corresponding output:

Example 25.1. Richard Wagner, *Rienzi* Theme. Anon. *My Bonnie Lies Over the Ocean.*



```
!!!COM: Wagner, Richard
!!!OTL: Rienzi Overture
**kern          **accent
*M4/4           *M4/4
*D:             *D:
=1              =1
4.d             0.76 *
32c#            0.48
32d             0.46
32e             0.47
32d             0.46
4.b             0.63 *
8a              0.52
=2              =2
4g              0.68 *
4e              0.62 *
4d              0.66 *
4A              0.65 *
*-              *-
```

```
!!!OTL: My Bonnie Lies Over the Ocean
**kern          **accent
*M3/4           *M3/4
*G:             *G:
4d              0.705 *
=1              =1
4.b             0.729 *
8a              0.513
4g              0.671 *
=2              =2
4a              0.676 *
4g              0.652 *
4e              0.633 *
=3              =3
4d              0.696 *
2B              0.659 *
*-              *-
```

The similarity between these two passages is more evident when the perceptually more salient tones are considered alone. Using the `**accent` data, we might simplify one or both passages by extracting only those notes whose accent value exceeds some threshold. In the above examples, a threshold of 0.6 might be appropriate (marked with an asterisk). We can isolate these tones by using the **recode** and **yank** commands. First, we create an appropriate reassignment file for **recode**. In this case we have classified all notes as either primary, secondary, or tertiary:

```
>=0.6 primary
>=0.5 secondary
else  tertiary
```

Assuming this file is named `reassign`, we can pre-process our passage as follows:

```
recode -f reassign -s ^= -i '**accent' inputfile \
   | yank -m primary -r 0 | extract -i '**kern' > primary.krn
```

The file `primary.krn` contains only those notes having the highest estimated accent values. Using this file, we can continue processing using either a parametric (**correl**) or non-parametric (**simil**) similarity method.

Reprise

In this chapter we have introduced two types of similarity tools: **correl** and **simil**. For both tools, the criterion of similarity depends on the user's choice of input representation. For example, if the input represents fret-board finger patterns, then the similarity measures will reflect fret-board fingering similarity. Users need to choose carefully the type of pre-processing required to address the specific domain of interest.

In particular, we noted that the Humdrum **accent** provides a useful way of pre-processing passages so that only the structurally most important notes are considered during processing.

The **correl** command provides a way for measuring *parametric* similarity — where similarity is based on numerical resemblance. By contrast, the **simil** command provides a way for measuring *non-parametric* similarity: similar inputs are ones that require the least editing in order for one input to be made equivalent to the other. We saw that **simil** allows the user to define the edit penalties associated with different kinds of modifications. This allows the user to tailor the similarity measures to better suit the type of data being considered.

The tools described in this chapter complement the pattern searching tools (such as **patt, pattern** and **grep**) described earlier.

Chapter 26

Moving Signifiers Between Spines

In many types of tasks is it useful to be able to transfer signifiers from one spine to another. In this chapter we will discuss two commands. The **rend** command makes it possible to split characters in a single spine and distribute them across two or more spines. The **cleave** command does the reverse: it allows characters that are distributed across two or more spines to be gathered into a single new spine. These two commands provide opportunities to create Humdrum spines that contain precisely the information of interest to the user.

The *rend* Command

The **rend** command allows a Humdrum spine to be broken apart into two or more spines. Different pieces of information can be distributed to the individual output spines. Consider for example the following spine containing `**pitch` data:

```
**pitch
Ab3
F#4
C5
*-
```

The **rend** command might be used to structure this as three independent spines:

```
**octave        **note      **accidental
3               Ab          b
4               F#          #
5               C           .
*-              *-          *-
```

The operation of **rend** requires a reassignment file where each line contains an output exclusive interpretation, followed by a tab, followed by a regular expression. In the above case, the reassignment file (`reassign`) contained the following:

```
**octave        [0-9]
**note          [A-Gb#x]
**accidental    [b#x]
```

The first line tells **rend** that any signifiers matching the character-class 0-9 should be output in a

spine labelled `**octave`. The second line causes signifiers matching the upper-case letters A to G and the `b`, `#` and `x` signifiers to be output in a spine labelled `**note`. The third line causes signifiers matching just `b`, `#` and `x` to be output in a spine labelled `**accidental`.

The above output was generated by invoking the following command:

```
rend -i '**pitch' -f reassign inputfile
```

Note that the **-i** and **-f** options are mandatory. The **-i** option tells **rend** which input spines to process and the **-f** option tells **rend** the name of the file containing the spine-reassignments.

The **rend** command is typically paired with a subsequent **cleave** command.

The *cleave* Command

The **cleave** command amalgamates concurrent data tokens in two or more spines into a single data spine. In effect, **cleave** does the opposite of **rend.** By way of example, **cleave** can be used to transform the following:

```
**A         **B         **C
a           b           c
A           B           C
*-          *-          *-
```

into:

```
**new
abc
ABC
*-
```

Specifically, the above "cleaving" would be done using the following command:

```
cleave -i '**A,**B,**C' -o '**new' inputfile
```

Both the **-i** and **-o** options are mandatory. The **-i** option tells **cleave** which exclusive interpretations should be cleaved together. In the above case, we have provided a list of three types of data. The **-o** option tells **cleave** what to call the resulting cleaved spine. In this case, we've simply called the result `**new`.

Suppose that we would like to automatically add key-velocities to some `**MIDI` data that reflect the normal accents arising from the meter. For example, in 4/4 meter, we would like the first note in each measure to be strongest, the third beat to be next most strongest and so on. Recall that `**MIDI` data tokens consist of three elements: (1) the duration in MIDI clock ticks, (2) the key number (also on/off indication), and (3) the key velocity. In order to add accents, we need to change the key velocity values. The maximum MIDI key velocity value is 127; the minimum value is 0; and the default value is 64.

Consider the following hypothetical input file:

```
**kern
*M4/4
=1-
4c
8d
8e
8f
8g
8a
8b
=2
2cc
*-
```

The **metpos** command can be used to identify the metric position of various note onsets. Before using **metpos** however, we must use the **timebase** command to create an isorhythmic record structure. Since the shortest note is an eighth-note, the appropriate command is as follows.

```
timebase -t 8 scale > scale.tb
```

Using the timebased output, we can then invoke the **metpos** command:

```
metpos scale.tb > scale.met
```

The resulting output contains both the original input (on the left) and the metric position spine (on the right):

```
**kern    **metpos
*M4/4     *M4/4
*tb8      *tb8
=1-       =1-
4c        1
.         4
8d        3
8e        4
8f        2
8g        4
8a        3
8b        4
=2        =2
2cc       1
.         4
.         3
.         4
*-        *-
```

Let's now eliminate the null data tokens introduced by **timebase**. Using **humsed**, we delete each data record beginning with a period character:

```
humsed '/^\./d' scale.met > scale.tmp
```

Next, we can use the **recode** command to change the metric position values to appropriate MIDI key velocities. We might use the following reassignment file (named `accent`):

```
==1     100
==2     80
==3     60
==4     40
else    error
```

In applying **recode** we will take care to avoid processing measure numbers using the **-s** (skip) option:

```
recode -f accent -s ^= -i '**metpos' scale.tmp > scale.acc
```

The output will now appear as follows:

```
**kern   **metpos
*M4/4    *M4/4
*tb8     *tb8
=1-      =1-
4c       100
8d       60
8e       40
8f       80
8g       40
8a       60
8b       40
=2       =2
2cc      100
*-       *-
```

Now we can use the **midi** command to generate `**MIDI` data:

```
midi scale.acc > scale.mid
```

The result is given below:

```
**MIDI                  **metpos
*Ch1                    *
*M4/4                   *M4/4
*tb8                    *tb8
=1-                     =1-
72/60/64                100
72/-60/64 72/62/64      60
36/-62/64 36/64/64      40
36/-64/64 36/65/64      80
36/-65/64 36/67/64      40
36/-67/64 36/69/64      60
36/-69/64 36/71/64      40
```

```
=2                      =2
36/-71/64 36/72/64      100
144/-72/64              .
*-                      *-
```

Before using **cleave** to join the new key velocity values to the `**MIDI` data we need to delete the current key-down velocities. These are the values '64' preceding the tab character. The **humsed** command can be used as follows:

```
humsed 's/64<tab>/<tab>/' scale.mid > scale.tmp
```

The modified output will now be:

```
**MIDI                  **metpos
*Ch1                    *
*M4/4                   *M4/4
*tb8                    *tb8
=1-                     =1-
72/60/                  100
72/-60/64 72/62/        60
36/-62/64 36/64/        40
36/-64/64 36/65/        80
36/-65/64 36/67/        40
36/-67/64 36/69/        60
36/-69/64 36/71/        40
=2                      =2
36/-71/64 36/72/        100
144/-72/                .
*-                      *-
```

Finally, we use **cleave** to add the new key-down velocities.

```
cleave -i '**MIDI,**metpos' -o '**MIDI' scale.tmp > scale.mid
```

The final output is:

```
**MIDI
*
*M4/4
*tb8
=1-=1-
72/60/100
72/-60/64 72/62/60
36/-62/64 36/64/40
36/-64/64 36/65/80
36/-65/64 36/67/40
36/-67/64 36/69/60
36/-69/64 36/71/40
=2=2
36/-71/64 36/72/100
```

```
144/-72/
*-
```

Creating Mixed Representations

For some analytic tasks it is often useful to generate a special representation that combines all of the elements or types of data of interest to the researcher. For example, suppose we were working on a model of melodic organization that reduced melodies to three types of information: relative-duration context, gross pitch height, and scale step. Sample data tokens for our representation and their meanings are given in the following table. Notice that the order of signifiers is important:

token	**meaning**
`LSLHto`	long-short-long rhythm, high pitch, tonic
`LLSLsd`	long-long-short rhythm, low pitch, subdominant
`MLSMlt`	medium-long-short rhythm, medium pitch, leading tone
`r`	rest

In Chapter 22 we learned how to use **recode** to classify various numerical ranges and **humsed** to classify non-numeric data. We already know how to create the elements of our new representation.

The scale degree information can be created by using **deg** and **humsed** can be used to transform the signifiers as in the following `degree` file:

```
s/1.*/to/
s/2.*/st/
s/3.*/me/
s/4.*/sd/
s/5.*/do/
s/6.*/sm/
s/7.*/lt/
```

We can classify the pitch ranges into high, medium, and low using the **semits** command, followed by **recode**. For example, we could transform the `**semits` data using the following reassignment file:

```
<0      L
>16     H
>=0     M
else    r
```

Durations can be similarly classified into long (L), medium (M), and short (S) using the **dur** command, followed by recode.

```
>1.0     L
>0.5     M
>0       S
```

Using **context -n 3** we could then create contextual 'triples' so that data records contain three durations. Suppose also that we have used **sed** to change the names of the exclusive interpretations so they are more appropriate. As a result we have three spines that, when assembled together are organized as in Example 26.1

Example 26.1

```
**rhythm      **range     **scale-step
L S L         H           to
L L S         L           sd
M L S         M           lt
r             r           r
*-            *-          *-
```

We need to process the first spine with **humsed** again to eliminate the spaces in the multiple stops. The rhythm spine would be processed as follows:

```
humsed 's/ //g' rhythm > rhythm.new
```

We could assemble these spines using the **assemble** command:

```
assemble rhythm.new range scale.step > newfile
```

Finally we can use **cleave** to amalgamate all of the data into a single final spine.

```
cleave -i '**rhythm,**range,**scale-step' -o '**complex' \
   newfile > output
```

Having created our new representation, we can continue to process this new data with the various Humdrum tools. For example, we could generate inventories that answer questions such as "How often does a high subdominant note in a long-short-long rhythmic follow a low submediant in a long-long-short context?

A similar approach can be used to address other questions, such as do large leaps involving chromatically-altered tones tend to have a longer duration on the altered tone? Etc.

Reprise

In this chapter we have seen how **rend** and **cleave** can be used to take bits and pieces of signifiers from potentially many spines, and assemble a composite Humdrum spine that contains precisely the information of interest. Before amalgamating spines, you can use the **humsed** command to translate the characters/signifiers so that you use your preferred way of representing something.

Chapter 27

Text and Lyrics

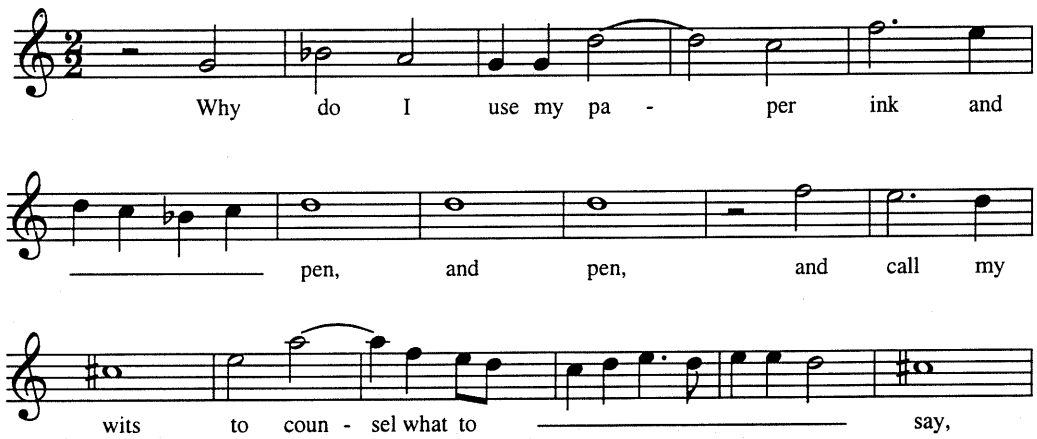
Musical texts include lyrics, librettos, stage directions, recitativo and other collections of words. It goes without saying that words provide important information related to semantics, imagery, similes, allusion, irony, parody, word painting, and emotion. In addition, words can add to the sonorous dimension of music, including rhyme schemes, word-rhythms, and phonetic and syllabic effects such as alliteration, vowel-coloration.

Humdrum provides three pre-defined representations pertinent to text or lyrics. The `**text` representation can be used to represent *words*; the `**silbe` representation can be used to represent *syllables*; and the `**IPA` representation can be used to represent *phonemes* (via the International Phonetic Alphabet). Discussion of the `**IPA` representation will be delayed until Chapter 34. In this chapter we will look at various representational and processing issues related to the manipulation of words and syllables.

The **text and **silbe Representations

Syllable- and word-oriented representations are illustrated in the following excerpt from a motet by Byrd (Example 27.1). The encoded Humdrum data includes three spines: `**text`, `**silbe` and `**kern`. Normally, only the `**silbe` and `**kern` data would be encoded — since the `**text` spine can be generated from the `**silbe` representation.

Example 27.1. From William Byrd, *Why Do I use my paper ink and pen.*



```
!!!COM: Byrd, William
!!!OTL: Why do I use my paper ink and pen.
**text      **silbe     **kern
*LEnglish   *LEnglish   *
=11         =11         =11
.           .           2r
Why         Why         2g
=12         =12         =12
do          do          2b-
I           I           2a
=13         =13         =13
use         use         4g
my          my          4g
paper       pa-         [2dd
=14         =14         =14
.           |           2dd]
.           -per        2cc
=15         =15         =15
ink         ink         2.ff
and         and         (4ee
=16         =16         =16
.           |           4dd
.           |           4cc
.           |           4b-
.           |           4cc
=17         =17         =17
pen,        pen,        1dd
=18         =18         =18
and         and         1dd
=19         =19         =19
pen,        pen,        1dd
=20         =20         =20
.           .           2r
and         and         2ff
=21         =21         =21
call        call        2.ee
my          my          4dd
=22         =22         =22
wits        wits        1cc#
=23         =23         =23
to          to          2ee
counsel     coun-       [2aa
=24         =24         =24
.           -sel        4aa]
what        what        4ff
to          to          (8ee
.           |           8dd
=25         =25         =25
```

```
.               |               4cc
.               |               4dd
.               |               4.ee
.               |               8dd
=26             =26             =26
.               |               4ee
.               |               4ee
.               |               2dd)
=27             =27             =27
say,            say,            1cc#
=28             =28             =28
*-              *-              *-
```

Note that all three representations in Example 27.1 make use of the common system for representing barlines. In the `**text` representation tokens represent individual words. In some scores, several words will be associated with a single moment (or pitch), as in the case of *recitativo* passages. Multi-word tokens are encoded as Humdrum multiple-stops with a space separating each word on a record.

In the `**silbe` representation tokens represent individual syllables. In `**silbe` the hyphen (-) is used explicitly to signify syllable boundaries and the tilde (~) is used to signify boundaries between hyphenated words (necessarily also a syllable boundary). In other words, four types of syllables are distinguished by `**silbe`: (1) a single-syllable word, (2) a word-initiating syllable, (3) a word-completing syllable, and (4) a mid-word syllable. The following table illustrates how these signifiers are used:

Table 26.1.

text	a single-syllable word
text-	a word-initiating syllable
-text	a word-completing syllable
-text-	a mid-word syllable
text~	a single-syllable word beginning a hyphenated multi-word
~text	a single-syllable word completing ahyphenated multi-word
~text~	a single-syllabe word continuing a hyphenated multi-word
~text-	a word-initating syllable continuing a hyphenated multi-word
-text~	a word-completing syllable — part of a hyphenated multi-word

Both the `**text` and `**silbe` representations are able to distinguish different tones of voice such as spoken voice, whispered voice, laughing voice, emotional voice, *Sprechstimme* and humming. In addition, there are signifiers for indicating untexted laughter and untexted sobs or crys. Some sample signifiers are shown in Table 26.2

Table 26.2.

`A-Z`	upper-case letters A to Z
`a-z`	lower-case letters a-z
`(`	open parenthesis
`)`	closed parenthesis
`{`	beginning of phrase
`}`	end of phrase
`%`	silence (rest) token (character by itself)
`M`	humming voice (character by itself)
`[`	beginning of spoken voice
`[[`	beginning of whisper
`]`	end of spoken voice
`]]`	end of whisper
`<`	beginning of *Sprechstimme*
`>`	*end of Sprechstimme*
`#`	beginning of laughing voice
`##`	end of laughing voice
`@`	laughter (no text)
`&`	sob or cry (no text)
`$`	beginning of emotional voice
`$$`	end of emotional voice
`*`	follows stressed word (`**test`) or stressed syllable (`**silbe`)

Signifiers common to `**text` and `**silbe`

The *text* Command

In most notated music, lyrics are written using a syllabic representation rather than a word-oriented representation. The `**silbe` representation is typically a better representation of the score than `**text`. However, for many analytic applications, words often prove to be more convenient. The Humdrum **text** command can be used to translate `**silbe` data to `**text` data. In general, syllabic information is useful for addressing questions related to rhythm and rhyme, whereas text information is more useful for addressing questions related to semantics, metaphor, word-painting, etc.

Invoking the **text** command is straightforward:

```
text inputfile > outputfile
```

A simple text-related task might be looking for occurrences of a particular word, such as the German "Liebe" (love). If the lyrics are encoded in the `**text` representation, then a simple **grep** will suffice:

```
grep -n 'Liebe' schubert
```

Recall that the **-n** option gives the line number of any occurrences found. If the input is encoded in the `**silbe` representation, then the output of **text** can be piped to **grep**:

```
extract -i '**silbe' schubert | text | grep -n 'Liebe'
```

Given a `**silbe` input, a inventory of words can be generated using **sort** and **uniq** in the usual way:

```
extract -i '**silbe' song | text | rid -GLId | sort | uniq
```

Frequently, it is useful to search for a group of words rather than individual words. Suppose we are looking for the phrase "white Pangur." The **context** command can be used to amalgamate words as multiple stops. If we are looking for a phrase consisting of just two words, we might use the **-n 2** option for **context**:

```
text barber | context -n 2 | grep -i 'white Pangur'
```

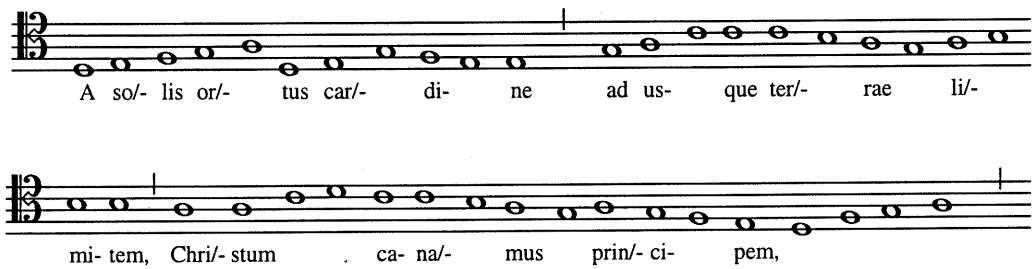
Alternatively, we might amalgamate words so they form sentences, or at least phrases. Puntuation marks provide a convenient marker for ending the amalgamation process carried out by **context**. In the following command, we have defined a regular expression with a character-class containing all of the puntuation marks. The output from this command will display all punctuated phrases (one per line) that contain the phrase "white Pangur."

```
text | context -e '[.,;?!]' | grep -i 'white Pangur'
```

The *fmt* Command

Another common task is simply to provide a readable text of the text or lyrics of a work. Given a `**text` representation, we can use the **rid** command to eliminate all records except non-null data records. This will result in a list of words — one word per line. UNIX provides a simple text formatter called **fmt** that will assemble words or lines into a block text where all output lines are roughly the same width. Consider the Gregorian chant *A Solis Ortus* from the *Liber Usualis* (shown in Example 27.2.)

Example 27.2. Beginning of chant *A Solis Ortus*.



The Latin text for this chant can be formatted as follows:

```
extract -i '**silbe' chant12 | text | rid -GLId | fmt -50
```

The **-50** option tells **fmt** to place no more than 50 characters per line. The default line-length is 72 characters. The above pipeline produces the following output:

```
A solis ortus cardine ad usque terrae limitem,
Christum canamus principem, natum Maria Virgine.
Beatus auctor saeculi servile corpus induit: ut
carne carnem liberans, ne perderet quos condidit.
Castae parentis viscera cae lestis intratgratia:
venter puellae bajulat secreta, quae non noverat.
Domus pudici pectoris tem plum repente fit Dei:
intacta nesciens virum, concepit alvo filium.
```

Another useful output would have the text arranged with one sentence or phrase on each line. As before we can use the **context** command with the **-e** option to amalgamate words, where each amalgamated line ends with a punctuation mark:

```
extract -i '**silbe' chant12 | text | context -e '[.,;:?!]' \
   | rid -GLId
```

The corresponding output is:

```
A solis ortus cardine ad usque terrae limitem,
Christum canamus principem,
natum Maria Virgine.
Beatus auctor saeculi servile corpus induit:
ut carne carnem liberans,
ne perderet quos condidit.
Castae parentis viscera cae lestis intratgratia:
venter puellae bajulat secreta,
quae non noverat.
Domus pudici pectoris tem plum repente fit Dei:
intacta nesciens virum,
concepit alvo filium.
```

Yet another way of arranging the text output would be to parse the text according to explicit phrase marks in the `**kern` data. This will require a little more work, but the result will be worthwhile. First, we will need to transfer the end-of-phrase signifier ('}') from the `**kern` spine to the `**silbe` spine. This transfer entails four steps. (1) Extract the monophonic `**kern` spine and eliminate all data signifiers except closing curly braces ('}'). Store the result in a temporary file:

```
extract -i '**kern' chant12 | humsed 's/[^}]*//; s/^$/./' \
   > temp1
```

Notice that **humsed** has been given two substitution commands. The first eliminates all data signifiers except the close curly brace. The second substitution transforms empty output lines to null data records by adding a single period.

(2) Extract the `**silbe` spine, translate it to `**text` and store the result in another temporary file:

```
extract -i '**silbe' chant12 | text > temp2
```

(3) Assemble the two temporary files together and use the **cleave** command to join the end-of-phrase marker to the syllable representation.

```
assemble temp1 temp2 | cleave -i '**kern,**text' \
   -o '**text' > temp3
```

With this cleaved data we can now use the **context** command to amalgamate phrase-related text. Finally, **rid** is used to eliminate everything but non-null data records.

```
context -o = -e } temp3 | rid -GLId
```

The result is as follows:

```
A solis ortus cardine }
ad usque terrae limitem, }
Christum canamus principem, }
natum Maria Virgine. }
Beatus auctor saeculi }
servile corpus induit: }
ut carne carnem liberans, }
ne perderet quos condidit. }
Castae parentis viscera }
cae lestis intratgratia: }
venter puellae bajulat }
secreta, quae non noverat. }
Domus pudici pectoris }
tem plum repente fit Dei: }
intacta nesciens virum, }
concepit alvo filium. }
```

We could clean up the output by using the **sed** command to remove the trailing closed curly brace. We simple add the following to the pipeline:

```
. . . | sed 's/}//'
```

You might have noticed that each of the above phrases seems to consist of eight syllables. We can confirm this by returning to the syllabic rather than word-oriented output. For the above command sequence, simply omit the **text** command and replace `**text` with `**silbe`. The revised script becomes:

```
extract -i '**kern' chant12 | humsed 's/[^}]*//; s/^$/./' \
   > temp1
extract -i '**silbe' chant12 > temp2
assemble temp1 temp2 | cleave -i '**kern,**silbe' \
   -o '**silbe' > temp3
context -o = -e } temp3 | rid -GLId | sed 's/}//'
```

The corresponding output is:

```
A so/- -lis or/- -tus car/- -di- -ne
ad us- -que ter/- -rae li/- -mi- -tem,
Chri/- -stum ca- -na/- -mus prin/- -ci- -pem,
na/- -tum Ma- -ri/- -a Vir/- -gi- -ne.
Be- -a/- -tus au/- -ctor sae/- -cu- -li
ser- -vi/- -le cor/- -pus in/- -du- -it:
ut car/- -ne car/- -nem li/- -be- -rans,
ne per/- -de- -ret quos con/- -di- -dit.
Ca/- -stae pa- -ren/- -tis vis/- -ce- -ra
cae/ le/- -stis in/- -trat- -gra/- -ti- -a:
ven/- -ter pu- -el/- -lae ba/- -ju- -lat
se- -cre/- -ta, quae non no/- -ve- -rat.
Do/- -mus pu- -di- -ci pe/- -cto- -ris
tem/ plum re- -pen/- -te fit De/- -i:
in- -ta/- -cta ne/- -sci- -ens vi/- -rum,
con- -ce/- -pit al/- -vo fi/- -li- -um.
```

If we are looking for vocal texts that exhibit a recurring rhythm, we might make a simple addition to the above script. Instead of outputting the actual syllables in each phrase, we would output a count of the number of syllables in each phrase. The standard **awk** utility allows us to write simple in-line programs. The following **awk** script simply outputs the number of fields (white-space separated text) in each input line:

```
awk '{print NF}'
```

If we add this to the end of our command sequence, then the output would simply be a sequence of numbers — where each number indicates the number of syllables in successive phrases. In the case of *O Solis Ortus* our output would consist of a series of 8s indicating that each phrase contains precisely eighth syllables.

By way of summary, we can generalize the above process so that syllable/phrase schemes can be generated for any syllable-related input. The following script counts the number of syllables in successive phrases for a single input file.

```
# SYLLABLE - count the number of syllables in each phrase
#
# Usage: syllable filename [ > outputfile]
#
extract -i '**kern' $1 | humsed 's/[^}]*//; s/^$/./' > temp1
extract -i '**silbe' $1 > temp2
assemble temp1 temp2 | cleave -i '**kern,**silbe' -o '**silbe' \
   | context -o = -e } | rid -GLId | sed 's/}//' | awk '{print NF}'
rm temp[12]
```

Variations on this theme abound. For example, if we wish to determine the number of syllables between successive punctuation marks, the following pipeline could be used:

```
extract -i '**silbe' | context -o = -e '[.,;:?!]' \
   | rid -GLId | awk '{print NF}'
```

Rhythmic Feet in Text

Another question related to rhythm is to identify rhythmic patterns. Once again, we might look at the chant *O Solis Ortus.* Below we have recoded the syllables in each phrase, where the value `0` indicates an unstressed syllable and `1` indicates a stressed syllable:

```
0 1 0 1 0 1 0 0
0 0 0 1 0 1 0 0
1 0 0 1 0 1 0 0
1 0 0 1 0 1 0 0
0 1 0 1 0 1 0 0
0 1 0 1 0 1 0 0
0 1 0 1 0 1 0 0
0 1 0 0 0 1 0 0
1 0 0 1 0 1 0 0
1 1 0 1 0 1 0 0
1 0 0 1 0 1 0 0
0 1 0 0 0 1 0 0
1 0 0 0 0 1 0 0
1 0 0 1 0 0 1 0
0 1 0 1 0 0 1 0
0 1 0 1 0 1 0 0
```

The above output was generated using the **humsed** command. Any syllable containing a trailing asterisk (`*`) is re-written as a '1', otherwise as a '0'.

```
. . . | humsed 's/[^ ][^ ]*\*/1/g; s/[^1][^1]*$/0/g'
```

With the above output, we can generate an inventory of phrase-related text-rhythms.

```
. . . | sort | uniq -c | sort
```

With the following results:

```
5    0 1 0 1 0 1 0 0
4    1 0 0 1 0 1 0 0
2    0 1 0 0 0 1 0 0
1    0 1 0 1 0 0 1 0
1    1 1 0 1 0 1 0 0
1    1 0 0 0 0 1 0 0
1    1 0 0 1 0 0 1 0
1    0 0 0 1 0 1 0 0
```

We can create a summary rhythmic pattern by adding together the values in each column — that is, counting the number of accented syllables that occur in each syllable position within the phrase. We can isolate each column using the UNIX **cut** command; **cut** is analogous to the Humdrum **extract** command. Fields are delineated by white space (tabs or spaces). For example, **cut -f 1** will isolate the first column of numbers. We can then pipe the results to the **stats** utility in order to calculate the numerical total. For example,

```
. . . | cut -f 1 | stats | grep 'total'
. . . | cut -f 2 | stats | grep 'total'
. . . | cut -f 3 | stats | grep 'total'
etc
```

For the chant *O Solis Ortus* the results are as follows:

```
7 9 0 13 0 14 2 0
```

This means that there are seven stressed syllables in the first syllable position of the phrase, nine stressed syllables in the second syllable position, and so on. These results suggest the following rhythmic structure: medium-strong-weak-strong-weak-strong-weak-weak. By way of conclusion, it appears that this work has a strongly rhythmic text structure — implying that this 'chant' might have been sung rhythmically.

Concordance

A traditional text-related reference tool is the *concordance.* Concordances allow users to look up a word, to see the word in the context of several preceding and following words, and provide detailed information about the location of the word in some repertory or corpus.

Suppose, for example, that we wanted to create a concordance for the lyrics in Samuel Barber's songs. We would like to create a file that has a structure such as shown in Table 26.3 below. The first column identifies the filename. The second column identifies the bar number in which the keyword occurs. The third column gives a five-word context where the middle word (in bold) identifies the keyword.

Table 26.3.

```
chant29   4   ut possim cantare, Alleluia: gaudebunt
chant29   7   mea, dum cantavero tibi: Alleluia,
chant27   1   Cantate Domino canticum novum Alleluia:
chant54   4   Cantate Domino canticum novum, quia
chant24   10  Cantate Domino canticum novum: quia
chant42   14  totus non capit orbis, in
chant47   5   et exaltavit caput ejus; et
chant12   1   solis ortus cardine ad usque
chant14   4   arrisit orto caritas: Maria, dives
chant12   7   induit: ut carne carnem liberans,
chant58   5   et in carne mea videbo
chant12   7   ut carne carnem liberans, ne
chant14   6   sola quae casto potes fovere
chant17   3   et discerne causam meam de
chant21   2   Dominus a cena, misit aquam
etc.
```

We would also like to provide a **grep**-like search tool so users can search for particular keywords.

The following script will generate our concordance file. For each file specified in the input, we ex-

tract the **silbe spine and store it. We then process this spine no less than three times. In the first pass, we translate from the **silbe to the **text representation, and generate a context of 5 words (**-n 5**) making sure to omit barlines (**-o =**). We also pad the amalgamated line with three null tokens (**-p 3**) so the context is centered near the third word in the sequence. In the second pass, we generate a new spine (**nums) that contains only bar numbers. The **ditto** command is used to ensure that every data record contains a bar number. To ensure that pick-up bars are numbered with the value 0, we've used **humsed** to replace any leading null-tokens with the number 0. In the third pass, we replace every data token with the name of the file. Finally, we assemble all three of these spines, eliminate everything but data records, and also eliminate lines that don't contain any text. All of this processing is carried out in a while-loop that cycles through all of the files provided when the command is invoked.

```
while [ $# -ne 0 ]
do
    extract -i '**silbe' $1 > temp1
    text temp1 | context -o = -n 5 -p 3 > temp2
    num -n = -a '**nums' temp1 | extract -i '**nums' \
        | ditto | humsed 's/./0/' > temp3
    humsed "s/.*/$1/" temp1 > temp4
    assemble temp4 temp3 temp2 | rid -GLId | sed '/.* .$/d'
    shift;
done
rm temp[1-4]
```

Having generated our concordance file, we can now create a simple tool that allows us to search for keywords. Suppose we kept our concordance information in a file called ~/home/concord/master. In essence, we'd like to create a command akin to **grep** — but one that searches this file solely according to the third word in the in the context. We cannot use **grep** directly since it will find all occurrences of a word no matter where it occures in the context. We need to tell **grep** to ignore all other data. The filename, bar number, and context fields are separated by tabs. We can ignore the first two fields by eliminating everything up to the last tab in the line. Since words are separated by blank space, the expression [^]+ will match a word not containg spaces. In short, the regular expression "^.*<*tab*>[^]+ [^]+ " will match everything up to the first tab, followed by two additional words. All we need to do is paste our keyword to the end of this expression.

Below is a simple one-line script for a command called **keyword.** The user simply types the command **keyword** followed by a regular expression that will allow him/her to search for a given word in context. Note that since we've used the extended regular expression character '+' — we must invoke **egrep** rather than **grep** in our script:

```
# KEYWORD - A script for searching a master concordance file
#
# Usage:  keyword <regular expression>
#
egrep "^.*   [^ ]+ [^ ]+ $1" ~/home/concord/master
```

Concordances can be used for a number of applications. One might use a concordance to help identify metaphor or image related words (such as "light," "darkness," etc.)

Simile

One of the most important poetic devices is the *simile* — where an analogy or metaphorical link is created between two things ("My love is like a red red rose.") In English, similes are often (though not always) signalled by the presence of the words "like" or "as."

A simple task involves searching for 'like' or 'as' in the lyrics of some input. For each occurrence of these words, suppose that we would like to output a line that places the word in context — specifically the preceding and following four words.

First we transform and isolate the text data using the **text** and **extract** commands:

```
text inputfile | extract -i '**text'
```

Since the input may contain multiple-stops, we might consider the precaution of ensuring no more than one word per data record. For this we can use **humsed**. Specifically, we can replace any spaces by a carriage return. Since the carriage return is interpreted by the shell as the instruction to begin executing a command, we need to escape it. Depending on the shell, the carriage return can be escaped in various ways. One way is to precede the carriage return by control-V (meaning "verbatim"). Another way is to type control-M rather than a carriage return. In the following command we have used the backslash to escape a control-M character:

```
text inputfile | extract -i '**text' | humsed 's/  */M/g' \
   | egrep -4 '^|(like)|(as)$'
```

Having ensured that there is no more than one word per line we can now search for a line contain *just* "like" or "as." The **-4** option for **egrep** causes any matched lines to be output with four preceding and four following lines of context. In addition, an extra line is added consisting of two dashes (`--`) to segregate each pattern output. That is, for each match, ten lines of output are typically given. In order to generate our final output, we need to transform the linear list of words into a horizontal list where each line represents a single match for "like" or "as."

The **context** command would enable us to do this. Unfortunately, however, the output from **egrep** fails to conform to the Humdrum syntax. In particular, adding `^\*` to the regular expression will fail to ensure a proper Humdrum output since preceding and following contextual lines will also be output.

The **hum** command is a special command that takes non-Humdrum input and adds sufficient interpretation records so as to make the input conform to the Humdrum syntax. Typically, this means simply adding a generic initial exclusive interpretation (`**A`) and a spine-path terminator (`*-`). If the input contains tabs, then appropriate spines will be added.

```
text inputfile | extract -i '**text' | humsed 's/  */M/g' \
   | egrep -4 '^|(like)|(as)$' | hum
```

Now we can make use of the **context** command. Each context ends with the double-dash delimiters generated by **egrep**. The **rid** command can be used to eliminate the interpretations added by **hum**.

```
text inputfile | extract -i '**text' | humsed 's/  */\^M/g' \
   | egrep -4 '^|(like)|(as)$' | hum | context -e '--' \
   | rid -Id
```

Word Painting

Word painting has a long history in music. There are innumerable examples where the music has somehow reflected the meaning of the vocal text. Suppose we wanted to determine whether words designating height (e.g., English "high," German "hoch," French "haute/haut") tend to coincide with high pitches.

A simple approach would be to extract those sonorities that coincide with any of the words high/hoch/haut and determine the average pitch. We can then contrast this average pitch with the average pitch for the repertory as a whole. Any significant difference might alert us to possible word painting.

First we translate any pitch data to `**semits` and any `**silbe` data to `**text`. We will also filter the outputs to ensure that only `**semits` and `**text` are present.

```
semits * | text | extract -i '**semits,**text'
```

Since a word may be sustained through more than one pitch, and a pitch may be intoned for more than one word, we should use the **ditto** command to ensure that null tokens are filled-in.

```
semits * | text | extract -i '**semits,**text' | ditto -s =
```

Next, we can use **egrep** to search for the words of interest:

```
semits * | text | extract -i '**semits,**text' | ditto -s = \
   | egrep -i '^\*|high|hoch|haut'
```

Notice the addition of the expression `^\*` in the search pattern. This expression will match any Humdrum interpretation records and so ensures that the output conforms to the Humdrum syntax. We can now isolate the `**semits` data and pass the output to **stats** in order to determine the average pitch for the words coinciding with the words high/hoch/haut:

```
semits * | text | extract -i '**semits,**text' | ditto -s = \
   | egrep -i '^\*|high|hoch|haut' | extract -i '**semits' |
   stats
```

The average pitch for the entire work can be determined as follows:

```
semits * | extract -i '**semits' | ditto -s = | rid -GLI \
   | stats
```

Emotionality

Musical texts often convey or portray a wide range of emotions. Some texts celebrate the ecstacy of love or lament the sorrow of loss. Yet other texts exhibit little emotional content. Suppose that we wanted to create a tool that would allow us to estimate the degree of emotional "charge" in the lyrics of any given vocal work. A simple approach might be to look for words that are commonly associated with high emotional content.

Table 26.4 shows a sample of six words from a study where 10 people were asked to rate the degree of emotionality associated with 100 English words. Participants rated each word on a scale from −10 to +10 where −10 indicates a maximum negative emotional rating and +10 indicates a maximum positive emotional rating. The values shown identify the average rating for all 10 participants.

Table 26.4. Average Emotionality Ratings for English Words.

begin	+3.8
river	+4.2
friend	+5.2
love	+8.6
hate	−9.7
detest	−9.8

Clearly, such a rating system might allow us to create a tool that would automatically search a large database and identify those vocal works whose lyrics are most emotionally charged. One way to generate a crude index of emotionality is to measure the average ratings for the ten most emotion-laden words in a given input.

The **humsed** command provides an appropriate place to start. In effect, we would take a table (such as Table 26.4) and use it to create a series of substitutions. Emotionally-charged words would be replaced by a numerical rating. Our **humsed** script would have the following form. Notice that the first substitution is used to eliminate punctuation marks.

```
s/[.,;:'`"!?]//g
s/begin/+3.8/
s/river/+4.2/
s/friend/+5.2/
s/love/+8.6/
s/hate/-9.7/
s/detest/-9.8/
/[^0-9+-]/s/.*/./
```

Also notice that the final command transforms any data records that contains anything other than a number to a null data token. In other words, words that are not present in the emotionality list are not rated.

In order to process our input, any syllabic text would first be translated to the `**text` representation, and all other spines discarded using **extract -i**.

```
text inputfile | extract -i '**text' ...
```

Then we would translate the words using our "emotionality" script, eliminate everything other than data records, and calculate the numerical statistics:

```
text inputfile | extract -i '**text' | humsed -f emotion \
    | rid -GLId | stats
```

In general, works whose lyrics express predominantly positive emotions ought to exhibit positive emotionality estimates. Similarly, works expressing predominantly negative emotions ought to exhibit negative emotionality estimates. Of course the process of averaging may be deceptive. Two sorts of problems may arise. First, a large number of fairly neutral words will tend to dilute an otherwise large positive or negative score. It may be preferable to observe the maximum positive and negative values. Alternatively, it may be appropriate to limit the average to (say) the ten most emotionally charged words. We can do this by sorting the numerical values and using the **head** and **tail** commands to select the highest or lowest values. In our revised processing, we use **sort -n** to sort the values in numerical order — placing the output in a temporary file. The UNIX **head** command allows us to access a specified number of lines at the beginning of a file: the option **-5** specifies the first five lines. Similarly, the UNIX **tail** command allows us to access a specified number of lines at the end of a file. The ten highest and lowest values are then concatenated together and piped to the **stats** command:

```
text inputfile | extract -i '**text' | humsed -f emotion \
    | rid -GLId | sort -n > temp
head -5 temp > lowest
tail -5 temp > highest
cat highest lowest | stats
```

A second problem with averaging together emotion rating values is that an emotionally-charged work might include a rough balance of passionate words expressing both positive and negative emotions. This might result in an average near zero and be mistaken for lyrics that exhibit little emotionality. The **stats** command outputs a variance measure that can be used to gauge the spread of the data. However, another way to address this problem is by ignoring the plus and minus signs in the input. That is, a rough index of emotionality — independent of whether the emotion is predominantly negative or positive would simply focus on the most emotionally charged words.

The plus and minus signs can be eliminated using a simple **humsed** substitution prior to numerical sorting:

```
humsed 's/[+-]//g'
```

Once again, we could use the **head** command to isolate the 10 or 20 most emotionally charged words.

Another variant of this approach might be to identify those words in a text which are most emotionally charged. Suppose we wanted to determine the location of the most emotionally charged word. A combination of **sort** and **grep** can be applied to this task. First we generate a spine containing the emotional-charge values taking care to eliminate the signs:

```
text inputfile | extract -i '**text' | humsed -f emotion \
    | humsed 's/[+-]//g' > charges
```

Next we assemble this new spine with the original input:

```
assemble charges inputfile
```

We can isolate data records using **rid** and then use **sort -n** to sort according to the numbers present in the first column. The most emotional charged word will be at the end of the file (largest number) so we can use **tail -1** to identify the word:

```
assemble charges inputfile | rid -GLId | sort -n | tail -1
```

Having established what word has been estimated as having the highest emotional-charge, we can then use **grep -n** to establish the location(s) of this word in the original input file.

Other Types of Language Use

Apart from emotionality, language tends to be used differently in different musical genres. The contrast between *aria* and *recitative* provides a classic example. The aria is intended to be a poetic reflection of a certain emotional state or reaction whereas the recitative moves the action along by focusing on concrete circumstances. Any number of variants on our "emotionality" processing can be conceived. For example, we might create another language index related to the degree of abstraction/concreteness for words. Words such as Verona, knife and Montague are comparatively concrete, whereas words such as feud, love and tragedy are more conceptual or abstract. We might expect to be able to observe such differences in recitative versus aria texts.

Similarly, differences in language use can be found in folk and popular music. In the case of the folk *ballad*, a detailed story unfolds. Differences in language use may be correlated with the `!!!AGN` reference record used to identify genres.

Reprise

In this chapter we have introduced two text-related representations: `**text` and `**silbe`. We have examined the **text** command (which translates from `**silbe` to `**text`). We have also been exposed to the UNIX **fmt** command (a simple text formatter), the **cut** command (similar to **extract -f**), and the **head** and **tail** commands.

In Chapter 34 we will examine further representations and processes related to phonetic data.

Chapter 28

Dynamics

In this chapter we introduce three pre-defined representations pertaining to musical dynamics. One representation (**dyn) represents dynamic markings as they appear visually in a printed score. A second representation (**dynam) represents a rationalized interpretation of the notated dynamic markings in a score. A third scheme (**dB) represents continuous dynamic levels in decibels.

The ***dynam* and ***dyn* Representations

Musical scores commonly contain dynamic markings that include both written text (such as "*subito forte*" and "*dimin.*") and graphic representations (such as hairpin or wedge-shaped crescendo markings). Unfortunately, traditional dynamic markings are often confusing or ambiguous. Consider, for example, the following sequence of dynamic markings from a Beethoven piano sonata:

pp ***cresc.*** ***cresc.*** ***p*** ***cresc.*** ***pp***

What are we to make of these markings? Does the music gradually *crescendo* from *pianissimo* to *piano*? Does this initial *crescendo* occur in two distinct phases or does the repetition of the term "*cresc.*" merely indicate a continuation of a single *crescendo*? Does this *crescendo* move to a dynamic level above *piano* and abruptly reduce to *piano*? Does the final *crescendo* begin at a *piano* level and get louder — followed by a relatively abrupt reduction to *pianissimo*? Or does the final *crescendo* begin below *piano* and gradually reach *pianissimo*? Such ambiguities are rampant in printed musical scores. We can examine the accompanying musical context to help us resolve questions of interpretation, but computers are unable to bring such sophistication to the task.

Humdrum provides two pre-defined representations for score-related dynamic markings. One representation (`**dyn`) attempts to represent the dynamic markings as they appear in a visual rendering of a score. That is, `**dyn` represents the visual or orthographic information. A second representation (`**dynam`) provides a "rationalized" or canonical means for interpreting score-related dynamic indications. Users will want to choose one or the other representation depending on the analytic task being pursued.

In the first instance, `**dynam` uses standardized data tokens to represent particular dynamic levels. Table 28.1 shows the standard representations for `**dynam`. For example, the token `pp` rep-

resents the concept of pianissimo, even if the visual rendering may be ***pp*** or *pianissimo* or *pianiss.*, etc.

Table 28.1.

p	piano
pp	pianissimo
ppp	triple piano
pppp	quadruple piano, [etc]
f	forte
ff	fortissimo
fff	triple forte
ffff	quadruple forte, [etc]
mp	mezzo-piano
mf	mezzo-forte
s	subito (suddenly), e.g. spp (*subito pianissimo*), sf (*subito forte*)
z	sforzando = fp (forte-piano)
<	begin crescendo
>	begin diminuendo
(	continuing crescendo
)	continuing diminuendo
[	end crescendo
]	end diminuendo
X	explicit interpretation (not indicated in the score)
x	published interpretation (indicated in the score, often in parentheses)
r	rest (silence)
v	notated accent or stress

In the case of crescendo and diminuendo markings, `**dynam` requires an explicit interpretation of where the dynamic marking begins and ends. The beginning of a crescendo is indicated by the less-than sign (<). The end of the crescendo is marked by the open square bracket (`]`). Between the beginning and end points, *continuation signifiers* are encoded. For crescendos, continuations are indicated using the open parenthesis; for diminuendos, continuations are indicated using the closed parenthesis.

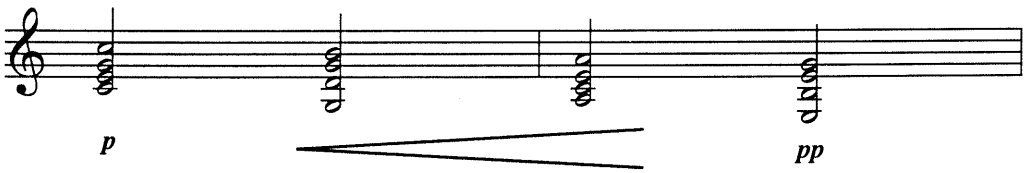
In the `**dynam` representation, no distinction is made for various ways a composer might indicate a crescendo or a diminuendo. For example, it doesn't matter whether a diminuendo is notated as *dim.*, *dimin.*, *diminuendo*, *decres.*, *decresc.*, *decresendo*, *calando*, *morendo*, *se perdant*, *cédéz*, *gradually quieter*, or via a hairpin or wedge graphic diminuendo. All are represented by > ...) ... `]`.

The `**dynam` representation also requires explicit resolution of possibly ambiguous dynamic markings. In many cases, the user will be required to add dynamic markings that are only implicit in the original score. Interpreted dynamics are preceded by the upper-case letter `X`, so an interpreted diminuendo will be represented by `X>` ... `X)` ... `X]`.

Often published editions will include dynamic markings that have been introduced by the editor. In scholarly publications these editorialisms are indicated in parentheses or square brackets. Such interpreted dynamics are preceded by the lower-case letter `x`.

The use of the **dynam representation is illustrated in Example 28.1.

Example 28.1



This example might be encoded as follows:

```
**dynam   **kern
*staff1   *staff1
=         =
p         2c 2e 2g 2cc
<         .
(         2G 2d 2g 2b
=         =
(         2A 2c 2e 2a
[         .
pp        2E 2B 2e 2g
=         =
*-        *-
```

The **dynam encoding is interpreted as follows: the level begins *piano* with a crescendo beginning prior to the second chord; the crescendo continues until after the third chord and then the level abruptly drops to *pianissimo* with the onset of the fourth chord. Notice that dynamic markings are "read from left-to-right"; that is, we presume that the crescendo begins *piano* and that the *pianissimo* is an abrupt reduction in level, rather than presuming that the crescendo builds to the *pianissimo* level and so there is an abrupt reduction in level after the initial *piano* just before the cresendo begins. In short, a crescendo (or diminuendo) marking is always assumed to increase (or decrease) the dynamic level from the preceding indication.

If appropriate, a user can render implicit dynamic shading explicitly. For example, a user might choose to re-code Example 28.1 as either

```
**dynam   **kern
*staff1   *staff1
=         =
p         2c 2e 2g 2cc
<         2G 2d 2g 2b
=         =
[         2A 2c 2e 2a
X>        .
X]        .
pp        2E 2B 2e 2g
=         =
*-        *-
```

or

```
**dynam   **kern
*staff1   *staff1
=         =
p         2c 2e 2g 2cc
Xppp      .
<         2G 2d 2g 2b
=         =
(         2A 2c 2e 2a
[         .
pp        2E 2B 2e 2g
=         =
*-        *-
```

Notice that null data records may be inserted as necessary to clarify the moment of dynamic change. The `**dynam` representation makes use of the common system for representing barlines.

The ***dyn* Representation**

The `**dyn` representation provides a method for representing the orthographic appearance of notated dynamic markings. Unlike `**dynam`, the `**dyn` representation distinguishes between different ways of identifying a dynamic marking. For example, *dim.*, *dimin.*, *diminuendo*, *decres.*, *decresc.*, *decresendo*, are all regarded as different from each other. Composers often have idiosyncratic ways of writing dynamic markings. As a result, the specific terms used may have repercussions, for example, in resolving cases of disputed composership. In some circumstances, it is thought that individual composers distinguish the terms in their own minds. For example, a composer might use *decrescendo* as a general term to indicate a temporary descending dynamic shape, whereas *diminuendo* might have a more specific meaning of a 'dying away' or 'fade-out' gesture.

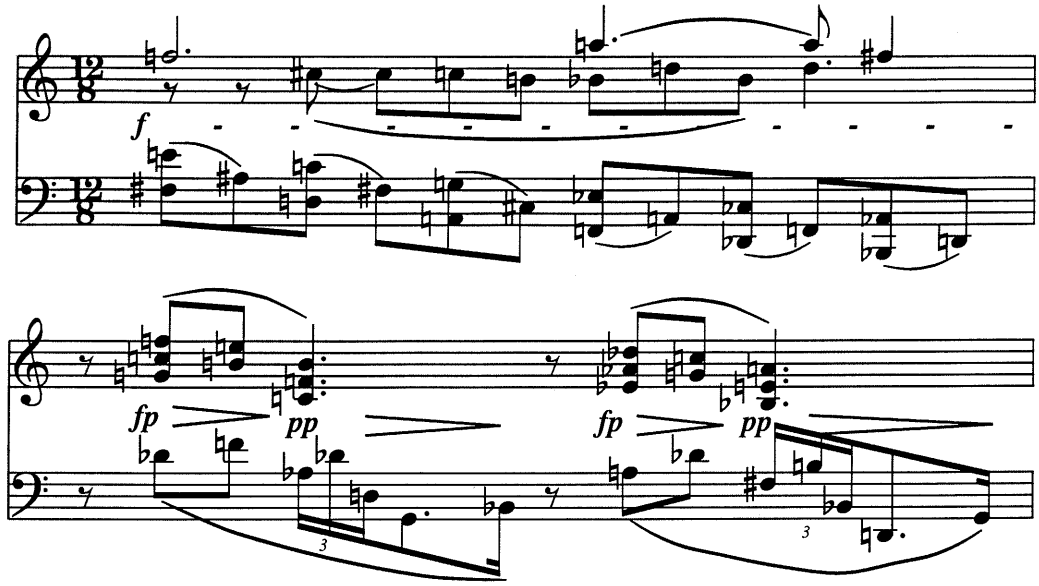
In the **dyn representation the horizontal position of dynamic markings is indicated in quarter-durations with respect to the previous barline. This number appears prior to the dynamic signifier, hence `4.1f` means a *forte* (e.g., *f*) marking just after the horizontal position of the fourth quarter in the measure. The vertical position of dynamic markings is indicated with respect to the middle line of a corresponding staff; this number appears in curly braces.

<	begin wedge-graphic crescendo marking
>	begin wedge-graphic diminuendo marking
[	terminate wedge-graphic crescendo marking
]	terminate wedge-graphic diminuendo marking
(	continuing wedge-graphic crescendo
)	continuing wedge-graphic diminuendo
{...}	vertical position (in staff-line steps from mid-line)
#...	size of marking (in staff-line steps)
:number:	density of dashed lines in strokes per quarter-duration
/.../	wedge opening size (in staff-line steps)
r	rest (silence)

H marking appears in square brackets

By way of illustration, consider Example 28.2.

Example 28.2: Arnold Schoenberg, *Three Piano Pieces, Op. 11, No. 2*, excerpt.



Using the **dyn representation, Example 28.2 might be encoded as follows:

```
**dyn          **kern         **kern         **kern
*staff1/2      *staff2        *staff1        *staff1
*              *clefF4        *clefG2        *clefG2
*              *M12/8         *M12/8         *M12/8
=              =              =              =
0.8f{-4}       8F# 8en        8r             2.ffn
.              8A#            8r             .
.              8Dn 8cn        [8cc#          .
.              8F#            8cc#]          .
.              8AAn 8Gn       8ccn           .
.              8C#            8bn            .
.              8FFn 8E-       8b-            [4.aan
.              8AAn           8ddn           .
.              8DD- 8C-       8b-            .
.              8FFn           4.dd           8aa]
.              8BBB- 8AA-     .              4ff#
.              8DDn           .              .
*              *              *v             *v
=              =              =
.              8r             8r
1.4fp{-4.5}    {8d-           {8gn 8ccn 8ffn
1.6>{-4.3}/1.5/.              .
)              8fn            8bn 8een
```

```
2.4]{-4.3}      .           .
2.5pp{-5}       24A-        4.cn 4.fn 4.b}
.               24d-        .
2.8>/1.4/       24Dn        .
)               8.GG        .
4.2]            16BB-}      .
.               8r          8r
4.4fp{-4.5}     {8An        {8e- 8a- 8dd-
4.6>/1.5/       .           .
)               8d-         8gn 8ccn
5.2]            .           .
5.4pp{-4.5}     24F#        4.B- 4.en 4.a}
5.7>/1.4/       24Bn        .
)               24BB-       .
)               8.DD        .
6.8]            16GG}       .
=               =           =
*-              *-          *-
```

The `*staff1/2` tandem interpretation indicates that the dynamic markings pertain to both staffs 1 and 2, however all vertical `**dyn` distance measures are encoded with respect to staff 1. (Reversing the numerical order — `*staff2/1` — would cause all distances to be measured with respect to staff 2.) The token `0.8f{-4}` means that the signifier *f* is located 0.8 quarter-duration spaces from the beginning of the bar and 4 staff-line steps below the center line of staff 1. The token `1.6>{-4.3}/1.5/` means that a wedge diminuendo marking begins 1.6 quarter-durations from the beginning of the bar; the size of the opening of the wedge is 1.5 staff-line steps wide and the center of the opening is located 4.3 staff-line steps below the center line for staff 1. The token `2.4]{-4.3}` means that a wedge diminuendo marking ends 2.4 quarter-durations from the beginning of the bar; the tip of the wedge converges at a point 4.3 staff-line steps below the center line for staff 1. Changing this value allows tilted wedges to be represented.

The ***dB* Representation

The `**dB` representation provides a way to represent intensity in decibels. Decibels can be expressed in relative or absolute terms. Absolute values are represented according to sound pressure level (SPL). An absolute representation is indicated by the presence of the `*SPL` tandem interpretation. Zero decibels (SPL) corresponds roughly to the quietest sound detectable under ideal circumstances. A quiet room is roughly 40 dB in intensity; a conversation produces roughly 70 dB, a vacuum cleaner produces roughly 80 dB, a noisy factory produces roughly 90 dB, and a passing loud motorcycle generates roughly 100 dB (SPL).

The `**dB` representation provides a convenient way to represent sound intensity in a numerical form. A numerical representation allows us to carry out a variety of calculations and comparisons.

The *db* Command

The **db** command translates dynamic markings to dynamic level expressed in decibels; specifically, **db** translates from the `**dynam` representation to `**dB` representation. By default, **db** uses the following mapping:

dynamic level	dB SPL
fffffff	115
ffffff	110
fffff	105
ffff	100
fff	90
ff	80
f	75
mf	70
mp	65
p	60
pp	55
ppp	50
pppp	45
ppppp	40
pppppp	35
ppppppp	30
v	+5

Notice the presence of the *accent* signifier (`v`); the assigned value (`+5`) means that any encoded accents will receive a decibel level 5 dB higher than the basic sound pressure level at that point in the score. For example, an explicitly accented note occurring in a *fortissimo* passage will be assigned a value of 85 dB SPL.

Users can define other mappings by using the **-f** option for **db**. With **-f** the user provides a filename that contains the non-default mapping values. The format for this file is the same as that shown in the above table. Each table entry specifies a dynamic marking, followed by a tab, followed by a numerical value.

In the case of crescendo and diminuendo markings, the **db** command attempts to interpolate a series of values between any preceding and subsequent dynamic markings. The following example shows a pianissimo marking at the beginning of measure 5; a crescendo marking spans all of measure 6, and a mezzo-forte marking appears in measure 7. The right-most spine shows the corresponding output generated by the **db** command. It shows an interpolation between the two dynamic levels.

```
**dynam    **dB
*SPL       *
=5         =5
pp         55
.          .
```

```
.           .
.           .
=6          =6
<           58
(           61
.           .
(           64
]           67
=7          =7
.           .
mf          70
.           .
```

The interpolation begins with the crescendo indicator and increments for each continuation signifier (i.e., the open parentheses). Interpolations are linear and continue up to the crescendo termination signifier. The size of the increment value depends on starting and ending dynamic levels as well as the number of crescendo-continuation signifiers. In the above case four pertinent crescendo signifiers separate the pianissimo and mezzo-forte markings; each of these records has been incremented by 3 decibels. Where necessary, decimal values are output. Notice that null tokens (such as those in the middle of measure 6) are ignored in the calculation.

Processing Dynamic Information

The `**dB` representation can be used to assist a number of tasks related to musical dynamics. Suppose, for example, that we want to compare the average overall dynamic levels for two arabesques:

```
extract -i '**dynam' arabesque1 | db | rid -GLId | stats
extract -i '**dynam' arabesque2 | db | rid -GLId | stats
```

Similarly, we might compare the overall dynamic levels between two sections of a single work. Perhaps we wish to know whether the exposition is on average louder than the development section:

```
yank -s Exposition -r 1 symphony3 | extract -i '**dynam' \
    | db | rid -GLId | stats
yank -s Development -r 1 symphony3 | extract -i '**dynam' \
    | db | rid -GLId | stats
```

Does a work tend to begin quietly and end loudly, or vice versa? Here we might compare the first 10 measures with the final 10 measures. Notice the use of **ditto** to increase the number of values participating in the calculation of the average dynamic level:

```
yank -n = -r 1-10 janacek | extract -i '**dynam' \
    | ditto -s = | db | rid -GLId | stats
yank -n = -r '$-10-$' janacek | extract -i '**dynam' \
    | ditto -s = | db | rid -GLId | stats
```

Suppose we want to determine whether there is an association between dynamic levels and pitch height for Klezmer music. That is, does the music tend to be quieter for lower pitches and louder for higher pitches? A straightforward way to determine this is to compare dynamic level with pitch height — represented in semitones (**semits). The **correl** command can then be used to measure Pearson's coefficient of correlation. If there is a relationship between pitch height and dynamic level then the correlation should be positive.

```
semits klezmer | correl -s ^= -m
```

This command assumes an input consisting of two spines — one pitch-related and a **dB spine. The **-s** option for **correl** is used to skip barlines so bar numbers aren't included in the calculation. The **-m** option for **correl** disables the "matched pairs" criterion. Normally, if a number is found in one spine but not the other then **correl** will complain and terminate. With the **-m** option, each encoded pitch need not have a corresponding dynamic level indication and vice versa.

Similarly, we could use this same approach to determine whether there is a relationship between duration and dynamic level. Are longer notes more likely to be louder in Klezmer music?

```
dur klezmer | correl -s ^= -m
```

A variation on this procedure might be to restrict the comparison over a specified pitch range. For example, one might think that higher pitches tend to be louder but that lower pitches are neither softer nor louder than usual. In order to test this view we can use the **recode** command to reassign "low" pitches to a single value. By way of illustration, the reassignment might presume that below G4 (semits=7) there is no relationship between pitch height and dynamic level. We might recode all values lower than 7 to a unique string (such as 'XXX') and then use **grep -v** to eliminate these notes from a subsequent correlation:

```
extract -i '**kern' klezmer | semits recode > temp1
extract -i '**dB' klezmer > temp2
assemble temp1 temp2 | grep -v 'XXX' | correl -s ^= -m
```

Terraced Dynamics

Suppose we want to identify whether various works exhibit "terraced" or "graduated" dynamics. In the case of terraced dynamics, we would expect to see many relatively abrupt dynamic contrasts, such as alternations between *forte* and *piano*. There are several ways of approaching this question. One approach might translate **dynam data to **dB data and then calculate the average (or maximum) changes in dynamic level. If a work contains many crescendos and diminuendos markings, then most of the changes in **dB values will be small. Conversely, alternations between contrasting dynamic levels will cause the average decibel differences to be larger. The **xdelta** command can be used to calculate the changes in dynamic level. Notice that it is important to avoid using the **ditto** command since repeated dynamic level values will cause the average dynamic difference to approach zero.

```
extract -i '**dynam' haendel | db | xdelta -a -s = | rid -d \
    | stats
```

Another approach to this problem might be to count the number of dynamic contrasts, avoiding the use of the **db** command. In the following pipeline, we use **context** to generate pairs of dynamic markings, and then use **grep** to count the number of alternations between ***f*** and ***p***.

```
extract -i '**dynam' haendel | grep -v '[][()=rX]' | rid -d \
   | context -n 2 | grep -c 'f p'
extract -i '**dynam' haendel | grep -v '[][()=rX]' | rid -d \
   | context -n 2 | grep -c 'p f'
```

Dynamic Swells

Conceptually, crescendos and diminuendos can be paired to form one of two dynamic gestures. A "swell" gesture consists of a crescendo followed by a diminuendo. Conversely, a "dip" gesture would consist of a diminuendo followed by a crescendo. Musical intuition would suggest that swell gestures are more common than dip gestures. We could test this view as follows:

```
extract -i '**dynam' grieg | grep -v '[][()=rX]' | rid -d \
   | context -n 2 | grep -c '< >'
extract -i '**dynam' grieg | grep -v '[][()=rX]' | rid -d \
   | context -n 2 | grep -c '> <'
```

MIDI Dynamics

Dynamic level data is not always easily available. One possible source is to translate MIDI key-velocity data to an estimated decibel value. Actual sound pressure levels will depend on the timbre of the MIDI sounds, the specific pitch played, and the volume on the output amplifier. Nevertheless, a rough estimate of sound pressure level may be useful for various analytic tasks. Recall that in the **MIDI representation, key-velocity data is encoded as the final number in three-number tokens where numbers are separated by slashes. The first value in the triplet is elapsed clock ticks and the second value is the MIDI key number (positive for key-on events, negative for key-off events). By way of reminder, the following example shows three **kern notes with a corresponding **MIDI representation.

```
**kern   **MIDI
*        *Ch1
4c       72/60/64
4d       72/-60/64 72/62/64
4e       72/-62/64 72/64/64
.        72/-64/64
*-       *-
```

In order to translate to a **dB representation, we must first isolate the key velocity values for key-on events. The following **humsed** command simply eliminates all data up to (and including) the last slash character:

```
extract -i '**MIDI' mono_input | humsed 's/.*\///'
```

This will leave us with just the key-down velocity data. Let's suppose that the following rough decibel equivalents are established:

key velocity	approximate dB SPL
127	85
100	80
90	77
80	74
70	70
60	65
50	60
40	53
30	44
20	32
10	21
1	10
0	0

An appropriate reassignment file for **recode** would begin as follows:

```
>=127   85
>=100   80
>=90    77
>=80    74
>=70    70
etc.
```

The completed translation would be accomplished by the following pipeline:

```
extract -i '**MIDI' mono_input | humsed 's/.*\///' \
    | recode -f reassign | sed 's/**MIDI/**dB/'
```

Notice the use of the **sed** command to replace the **MIDI interpretation by a **dB interpretation.

Reprise

In this chapter we have introduced three representations related to musical dynamics. The `**dyn` representation allows us to encode dynamic markings as they appear visually in a printed score. Unfortunately, traditional notated dynamic markings are often confusing or ambiguous. In order to facilitate some types of analytic processing it is useful to generate a more rationalized interpretation of the dynamics of a work. The `**dynam` representation provides a canonical scheme for representing basic notated dynamic markings where ambiguities are resolved by explicitly interpreting the meaning of dynamic markings. A third scheme (**dB) provides a scheme for representing continuous dynamic levels in decibels. We have seen that the **db** command (which translates from `**dynam` to `**dB`) allows us to pose and answer a variety of questions related to the dynamic organization of music.

Chapter 29

Differences and Commonalities

In Chapter 25 we introduced the problem of similarity via the Humdrum **simil** and **correl** commands. This chapter revisits the problem of similarity by focussing on differences and commonalities. Specifically, this chapter introduces three additional tools, the UNIX **cmp**, **diff** and **comm** commands. Although these commands are less sophisticated than the **simil** and **correl** commands, they nevertheless provide convenient tools for quickly determining the relationship between two or more inputs.

Comparing Files Using *cmp*

The **cmp** command does a character-by-character comparison and indicates whether or not two files are identical.

```
cmp file1 file2
```

If the two files differ, **cmp** generates a message indicating the first point where the two files differ. E.g.,

```
file1 file2 differ: char 4, line 10
```

If the two files are identical, **cmp** simply outputs nothing ("silence is golden").

Sometimes files differ in ways that may be uninteresting. For example, we may suspect that a single work has been attributed to two different composers. The encoded files may differ only in that the `!!!COM:` reference records are different. We can pre-process the files using **rid** in order to determine whether the scores are otherwise identical.

```
rid -G file1 > temp1
rid -G file2 > temp2
cmp temp1 temp2
```

Of course one of the works might be transposed with respect to the other. We can circumvent this problem by translating the data to some key-independent representation such as `solfa` or `deg`:

```
rid -GL file1 | solfa > temp1
```

```
rid -GL file2 | solfa > temp2
cmp temp1 temp2
```

Two songs might have different melodies but employ the same lyrics. We can test whether they are identical by extracting and comparing any text-related spines. Since there may be differences due to melismas, we might also use **rid -d** to eliminate null data records.

```
extract -i '**silbe' file 1 | rid -GLd > temp1
extract -i '**silbe' file 2 | rid -GLd > temp2
cmp temp1 temp2
```

Similarly, two works might have identical harmonies:

```
extract -i '**harm' file 1 | rid -GLd > temp1
extract -i '**harm' file 2 | rid -GLd > temp2
cmp temp1 temp2
```

By further reducing the inputs, we can focus on quite specific elements, such as whether two songs have the same rhythm. In the following script, we first eliminate bar numbers, and then eliminate all data except for durations and barlines.

```
extract -i '**kern' file 1 | humsed '/=/s/[0-9]//; \
   s/[^0-9.=]//g' | rid -GLd > temp1
extract -i '**kern' file 1 | humsed '/=/s/[0-9]//; \
   s/[^0-9.=]//g' | rid -GLd > temp2
cmp temp1 temp2
```

For some tasks, we might focus on just a handful of records. For example, we might ask whether two works have the same changes of key.

```
grep '^*[a-gA-G][#-]*:'  file 1 > temp1
grep '^*[a-gA-G][#-]*:'  file 2 > temp2
cmp temp1 temp2
```

In the extreme case, we might compare just a single line of information. For example, we might identify whether two works have identical instrumentation:

```
grep '^!!!AIN:'  file 1 > temp1
grep '^!!!AIN:'  file 2 > temp2
cmp temp1 temp2
```

Comparing Files Using *diff*

The problem with **cmp** is that it is unable to distinguish whether the difference between two files is profound or superficial. A useful alternative to the **cmp** command is the UNIX **diff** command. The **diff** command attempts to determine the minimum set of changes needed to convert one file to another file. The output from **diff** entails editing commands reminiscent of the **ed** text editor. For example, two latin texts that differ at line 40, might generate the following output:

```
40c40
< es quiambulas
---
> es quisedes
```

Let's consider again the question of whether two works have essentially the same lyrics. Many otherwise similar texts might differ in trivial ways. For example, texts may differ in punctuation or in the use of upper- and lower-case characters. The **diff** command provides a **-i** option that ignores distinctions between upper- and lower-case characters. Punctuation marks can be eliminated by adding a suitable **humsed** filter.

```
extract -i '**silbe' file1 | text | humsed 's/[^a-zA-Z ]//g' \
    | rid -GLId > temp1
extract -i '**silbe' file2 | text | humsed 's/[^a-zA-Z ]//g' \
    | rid -GLId > temp2
diff -i file1 file2
```

Every time **diff** encounters a difference between the two files, it will output several lines identify the location of the difference and showing the conflicting lines in the two files. The **diff** command is line-oriented. Two lines need only differ by a single character in order for **diff** to generate an output.

When there are more than a dozen or so differences, the output becomes cumbersome to read. A useful alternative is to avoid looking at the raw output from **diff**; instead, we might simply count the number of lines of output (using **wc -l**). When compared with the total length of the input, the number of output lines can provide a rough estimate of the magnitude of the differences between the two files. A suitable revision to the last line of the above script would be:

```
diff -i file1 file2 | wc -l
```

One problem with this approach is that it assumes that we know which two files we want to compare. A more common problem is looking for *any* work that is somewhat similar to some given work. We can automate this task by embedding the above script in a loop so that the comparison (second) file cycles through a series of possibilities. A simple **while** loop will enable us to do this. Since our script may process a large number of scores, we ought to format our output for ease of reading. The **echo** command in our script outputs each filename in turn with the a count of the number of output lines generated by **diff**.

```
extract -i '**silbe' $1 | text | humsed 's/[^a-zA-Z ]//g' \
    | rid -GLId > temp1
shift
while [ $# -ne 0 ]
do
    extract -i '**silbe' $1 | text | humsed 's/[^a-zA-Z ]//g' \
        | rid -GLId > temp2
    CHANGES=`diff -i temp1 temp2 | wc -l`
    echo $1 ":  " $CHANGES
    shift
done
```

```
rm temp[12]
```

Of course this same approach may be applied to other musical aspects apart from musical texts. For example, with suitable changes in the processing, one could identify works that have similar rhythms, melodic contours, harmonies, rhyme schemes, and so on.

Comparing Inventories — The *comm* Command

The **diff** command is sensitive to the order of data. Suppose that texts for two songs differ only in that one song reverses the order of verses 3 and 4. Comparing the "wrong" verses will tend to exaggerate what are really minor differences between the two songs. In addition, the above approach is too sensitive to word or phrase repetition. Many works — especially polyphonic vocal works — use extensive repetitions (e.g., "on the bank, on the bank, on the bank of the river"). Short texts (such as for the *Kyrie* of the Latin mass) are especially prone to use highly distinctive repetition. How can we tell whether one work has pretty much the same lyrics as another?

Fortunately, most texts tend to have unique word inventories. Although words may be repeated or re-ordered, phrases interrupted, and verses re-arranged, the basic vocabulary for similar texts are often much the same. A useful technique is to focus on the similarity of the word inventories. In the following script, we simply create a list of words used in both the original and comparison files.

```
extract -i '**silbe' file1 | text | humsed 's/[.,;:!?]//g' \
    | rid -GLId | tr A-Z a-z | sort -d > inventory1
extract -i '**silbe' file2 | humsed 's/[.,;:!?]//g' | tr A-Z
a-z | text \
    | rid -GLId | sort | uniq -c | sort -nr > inventory2
```

Suppose that our two vocabulary inventories appear as follows:

Inventory 1:	**Inventory 2:**
domine	a
et	coronasti
eum	domine
filio	et
gloria	eum
in	filio
jerusalem	gloria
orietur	honore
patri	manuum
sancto	oper
spiritui	patri
super	sancto
te	spiritui
videbitur	super
	tuarum

Notice that a number of words are present in both texts, such as *domine*, *et*, *eum*, *filio*, and so on. Identifying the common vocabulary items is easily done by the UNIX **comm** command; **comm** compares two sorted files and identifies which lines are shared in common and which lines are unique to one file or the other.

The **comm** command outputs three columns: the first column identifies only those lines that are present in the first file, the second column identifies only those lines that are present in the second file, and the third column identifies those lines that are present in both files. In the case of our two Latin texts, the command:

```
comm inventory1 inventory2
```

will produce the following output. The first and second columns identify words unique to `inventory1` and `inventory1`, respectively. The third column identifies the common lines:

```
                a
                coronasti
                                domine
                                et
                                eum
                                filio
                                gloria
                honore
in
jerusalem
                manuum
                oper
orietur
                                patri
                                sancto
                                spiritui
                                super
te
                tuarum
videbitur
```

In the above case, five words are unique to `inventory1`, six words are unique to `inventory2` and nine words are common to both.

The **comm** command provides numbered options that suppress specified columns. For example, the command **comm -13** will suppress columns one and three (outputing column two). (Empty lines are also suppressed with these options.) A convenient measure of similarity is to express the shared vocabulary items as a percentage of the total combined vocabularies. We can do this using the word-count command, **wc**. The first command counts the total number of words and the second command counts the total number of shared words:

```
comm inventory1 inventory2 | wc -l
comm -3 inventory1 inventory2 | wc -l
```

An important point about **comm** is that the order of materials is important in the input files. If the word *filio* occurs near the beginning of `inventory1` but near the end of `inventory2` then **comm** will not consider the record common to both files. This is the reason why we used an alphabetical sort (**sort -d**) in our original processing.

On the other hand, there are sometimes good reasons to order the vocabulary lists non-alphabetically. For example, suppose we created our inventories according to the frequency of occurrence of the words. That is, suppose we use **uniq -c | sort -nr** to generate a vocabulary list ordered by how common each word is. Our inventory files might now appear as follows:

Inventory 1:

```
3   et
2   te
2   gloria
1   videbitur
1   super
1   spiritui
1   sancto
1   patri
1   orietur
1   jerusalem
1   in
1   filio
1   eum
1   domine
```

Inventory 2:

```
4   et
2   gloria
2   eum
1   tuarum
1   super
1   spiritui
1   sancto
1   patri
1   oper
1   manuum
1   honore
1   filio
1   domine
1   coronasti
1   a
```

Comparing these two inventories will produce little in common due to the presence of the numbers. For example, the records "`3 et`" and "`4 et`" will be deemed entirely different. However, we can eliminate the numbers using an appropriate **sed** command leaving us with vocabulary lists that are ordered according to the frequency of occurrence of the words. If we apply the

comm command to these lists then the commonality measures will be sensitive to the relative frequency of words within the vocabularies.

Reprise

In this chapter we have introduced the UNIX **cmp**, **diff** and **comm** commands. The **cmp** command determines whether two files as are the same or different. The **diff** command identifies how two files differ. The **comm** command identifies which (sorted) lines two files share in common; **comm** also allows us to identify which lines are unique to just one of the files.

The value of these tools is amplified when the inputs are pre-processed to eliminate unwanted or distracting data, and when post-processing is done (using **wc**) to estimate the magnitude of the differences or commonalities.

Together with the **simil** and **correl** commands discussed in Chapter 25, these five tools provide a variety of means for characterizing differences, commonalities, and similarities.

Chapter 30

MIDI Input Tools

The Humdrum Toolkit provides two tools for inputting MIDI data. In this chapter we briefly introduce the **record** and **encode** commands. These tools provide ways for capturing MIDI input and translating them to representations that conform to the Humdrum syntax. The **record** command translates a live or computer-generated MIDI performance to the `**MIDI` representation. The **encode** command provides an interactive editor that translates MIDI events to any pre-defined or user-defined Humdrum representation.

Note that the Humdrum **record** and **encode** commands are currently available only for the DOS operating system. These commands can be used only with computers that have MIDI capable hardware.

The *record* Command

The **record** command captures a stream of input MIDI data and translates this data into the Humdrum `**MIDI` representation described in Chapter 7. The input data is obtained from a MIDI instrument such as a keyboard synthesizer.

Recording commences as soon as the command is invoked and recording ceases when any key is pressed — with the exception of the space bar. Pressing the space bar causes a `**MIDI` barline token to be output. Measure numbers are incremented automatically beginning with measure 1.

Only MIDI key-press activity (including after-touch) information is recorded. MIDI "system-exclusive" instructions and other non-key-press data are not recorded.

Each MIDI channel is represented using a separate Humdrum spine. New spines are added automatically during the recording in response to MIDI activity appearing in any MIDI channel. Once a MIDI channel becomes active, the corresponding Humdrum spine continues to be output until the recording is terminated.

The recorded output is normally directed to a file as in the following:

```
record > filippa
```

The *encode* Command

The **encode** command provides an interactive editor for capturing Humdrum data from a MIDI input, such as a keyboard synthesizer. MIDI events are mapped to user-defined signifiers so **encode** can be used to enter data directly into a particular representation such as **kern, **fret, **solfg, etc. Since the mapping of MIDI events to Humdrum data tokens is arbitrary, users can enter data using a representation design by the user.

The **encode** command is limited to encoding information one spine at a time. A typical use of **encode** is to encode individual musical parts or voices using a representation like **kern. A full score is generated from the individual parts using the **timebase** and **assemble** commands.

The **encode** command implements a full-screen interactive editor similar to the **vi** text editor. When invoked, the screen is divided into three display regions, including a *status window*, a *command window*, and a larger *text window*. The *status window* displays various items of basic information about the file being edited. The **command window** allows the user to execute general-purpose commands to manipulate the data. The **text window** is used to display, encode and edit the encoded Humdrum text.

When the **encode** command is invoked, its operation is determined by a configuration file (that may be written or edited by the user). This configuration file contains a series of definitions that map MIDI events to output strings. For example, the instruction

```
KEY 60 middle-C
```

assigns the key-on event for MIDI key #60 to the string `middle-C`. Each time key #60 is depressed, the string `middle-C` will appear in the text window.

Such mappings can be made for each individual MIDI key. In addition, the user may define mappings for *key velocity*. For example, the following instruction in the configuration file will map any key-velocities between 90 and 127 MIDI units to the apostrophe character (the `**kern` signifier for a staccato note):

```
VEL 90 127 '
```

A third class of mapping instructions relates to the elapsed time between MIDI key onsets — "delta time" or DEL. Consider, for example, the following configuration instructions:

```
DEL 48 80 8
DEL 81 112 8.
DEL 113 160 4
DEL 161 224 4.
DEL 225 320 2
```

These instructions divide the elapsed time between key onsets into five ranges. When the elapsed time lies between 48 and 80 clock ticks, the string "8" is output. When the elapsed time lies between 81 and 112 clock ticks, the string "8." is output. And so on. This allows the durations to be classified as either an eighth note, a dotted eighth note, a quarter note, a dotted quarter note, or a half note. Once again, the user is free to map events to any arbitrary output string and to arrange

the ranges and number of classes as needed.

The Humdrum Toolkit provides a large selection of predefined configuration files for use with **encode**. Depending on the configuration, the input may be mapped to a particular representation such as **kern. For example, Humdrum provides a configuration file that is optimized for encoding lute tablatures using the **fret representation. Other configuration files are optimized for particular keys. For example, one may select a configuration file that interprets the MIDI events in the key of C# minor; in this case, playing the pitch C will result in a default encoding of B#.

The **encode** command provides many additional features that facilitate encoding Humdrum data from a MIDI input device. These include setting metronome values, assigning the beat, rearranging the order of signifiers, making global and local substitutions, replaying an interpreted input, defining buffers and string constants, and so on. Using the configuration files, users can tailor the **encode** editor to suit specific needs and skills.

Reprise

In this chapter we have briefly identified two tools for capturing MIDI-related input: **encode** and **record.** These tools allow MIDI data to be translated to a Humdrum format. Further information regarding these tools is given in the *Humdrum Toolkit Reference Manual.*

Chapter 31

Repertories and Links

In initiating a research project, we often begin by selecting a suitable repertory for study. A common approach is to focus on a particular composer, period, style, or culture. For example, a researcher might focus on 17th century canons, or on Beethoven's compositions prior to Opus 20, or on Ojibway songs transcribed in the 1900s. Depending on the research task, the user may wish to locate works that conform to highly complex criteria — such as solo Baroque flute works written in compound meters, or slow Russian symphonic movements that don't include any wind instruments and are written in minor keys.

In this chapter, we discuss how to search entire file-systems for Humdrum scores that conform to user-specified criteria. With large musical databases, automated methods for locating specific repertories become important. As we will see, in most cases, one or two commands are all that is necessary to assemble a repertory list of works conforming to complex selection criteria.

The *find* Command

The UNIX **find** command traverses through a file hierarchy, and finds all files that match certain conditions. The **find** command takes the following syntax:

```
find <PATH> <OPTIONS> <ACTIONS>
```

The *PATH* is a directory from which the search *commences*. All files in the specified directory are examined including all files in the subdirectories, sub-subdirectories, and so on.

The path

```
/
```

means the "root" directory containing all files on a computer system. Even single-user systems are apt to have several thousand files subsumed under the root directory.

The path

```
/scores
```

means all files under the `scores` directory, whereas

```
/scores/bach
```

means all files under the `scores/bach` directory. The period character:

```
.
```

tells **find** to commence searching from the current directory.

Since **find** searches all files under the given path, its operation may be quite slow when there are thousands of files to search. It's wise to restrict the search by choosing a reasonable starting point. For example, specifying the path `/scores/bach/chorales` may save a great deal of time compared with the path `/scores`. Although we won't discuss them in this book, the **find** command provides a number of options that help to restrict the depth of searches or otherwise "prune" the search. When first trying **find** it's a good idea to limit the searches to small segments of the file system.

When searching through the specific *PATH*, **find** is able to carry out a wide variety of possible tests on each file. One simple action is to test whether the file-name conforms to a given regular expression. Consider, for example, the goal of identifying all files representing pitch-class (`**pc`) information. The Humdrum convention is to identify these files by adding the `.pc` extension to the filename — such as `opus24.pc`. The following **find** command will traverse through the `/scores` directory (and all sub-directories) searching for files that contain the `pc` file extension:

```
find /scores -name *.pc
```

The above command uses the **-name** option followed by the appropriate regular expression. This command is unusual in that it has no explicit *action*. In such cases, the implied action is to print or display the name of all files whose names match the regular expression.

Note that regular expressions may be literal strings. This means we can locate a specific named file. For example, the following command will locate all files named `findme`:

```
find / -name findme
```

An example of an explicit *ACTION* might be to delete files conforming to a particular criterion. For example, the following command searches the `/scores` path for files whose names contain the `.tmp` extension. Any matching file is then deleted using the UNIX **rm** command:

```
find /scores -name *.tmp -exec rm "{}" ";"
```

This command illustrates a number of features of the **find** command. The search begins from the *path* `/scores`. The *option* `-name *.tmp` identifies the search condition. The `-exec` flag identifies the *action* as that of executing a command. The arguments between `-exec` and the semi-colon are treated as the command to be executed. Each time a file is found with the `.tmp` extension, the `-exec` action is executed. The paired curly braces {} have a special significance to **find.** The braces are replaced by the filename found to match the regular expression. For example, if the first file found is named `xyz.tmp` the braces will be replaced by the string `xyz.tmp`.

The quotation marks around the braces and around the semi-colon are necessary in order to prevent the UNIX shell interpreter from substituting inappropriate information before passing the command line to **rm**.

Note that the **-name** option defaults to filenames only; it does not apply to directory names. The above command is equivalent to the more explicit form:

```
find /scores -type f -name *.tmp -exec rm "{}" ";"
```

The **-type** option can be used to match regular files (**f**), directories (**d**), or network files (**n**). By way of example, the following command deletes all directories whose names have the `.tmp` extension.

```
find /scores -type d -name *.tmp -exec rmdir "{}" ";"
```

Content Searching

For most music research applications, we are interested in identifying files on the basis of their contents. That is, we'd like to know what's inside the file before we take any action.

The **grep** command is especially useful in determining whether certain items of information are present in a file. For example, the following command identifies all files in the path `/scores` that contain passages in 7/8 meter:

```
find /scores -type f -exec grep -l '\*M7/8' "{}" ";"
```

Recall that the **-l** option for **grep** causes the output to consist only of names of files that contain the sought regular expression. Note that the **-type f** option has been specified in order to ensure that the **grep** command is only executed for files.

The structure of the above command can be used to search for all sorts of pertinent musical information. For example, recall that the `*IC` tandem interpretation is used to encode instrument classes such as strings, voice, percussion, etc. The following command searches all files in the path `scores` and generates a list of those files that encode scores containing one or more brass instruments:

```
find scores -type f -exec grep -l '\*ICbras' "{}" ";"
```

The following command identifies all files in the path `/scores`, that contain passages in the key of C major:

```
find /scores -type f -exec grep -l '\*C:' "{}" ";"
```

The following command identifies all files in the path `/scores`, that contain passages in any minor key:

```
find /scores -type f -exec grep -l '\*[a-g][#-]*:' "{}" ";"
```

Humdrum reference records are ideal targets for such searches since reference records encode information such as the composer's name, composer's dates, title of work, date of composition, movement number, instrumentation, meter classification, and so on. For example, the following command identifies all files in the path `/scores` that are composed by Franck:

```
find /scores -type f -exec grep -l '!!!COM.*Franck' "{}" ";"
```

The following command identifies all files in the path `/scores` that are written in compound meters:

```
find /scores -type f -exec grep -l '!!!AMT.*compound' "{}" ";"
```

The following command identifies all files beginning from the current directory that are rondos:

```
find . -exec grep -il '!!!AFR.*rondo' "{}" ";"
```

Recall that the **-i** option for **grep** makes the pattern-match insensitive to upper- or lower-case.

The following command identifies all files in the path `non-western` that have been designated as having heterophonic textures:

```
find non-western -exec grep -il '!!!AST.*heterophony' "{}" ";"
```

In the path `/scores/jazz`, we might want to identify all files that contain the style-designation "bebop:"

```
find /scores/jazz -exec grep -il '!!!AST.*bebop' "{}" ";"
```

The following command identifies all files in the path `18th-century`, that include French horns and oboes:

```
find 18th-century -exec grep -il '!!!AIN.*cor.*oboe' "{}" ";"
```

Of course, more complex regular expressions can be also be defined. For example, the following command identifies all works composed between 1805 and 1809:

```
find / -exec grep -l '!!!ODT.*180[5-9]' "{}" ";"
```

There is no restriction on the complexity of the regular expression. The following command identifies all works composed between 1812 and 1840:

```
find / -exec egrep -l '!!!ODT.*18(1[2-9])|([23][0-9])|(40)' \
   "{}" ";"
```

Often the **find** command can be used to answer research questions more directly. Suppose we wanted to determine whether German drinking songs more likely to be in triple meter. There are over four thousand German folksongs encoded in Helmut Schaffrath's *Essen Folksong Collection.* These works contain genre-related tags encoded as "AGN" reference records. One of the genres distinguished is "Trinklied" (drinking song).

In order to answer our question, we need to search the file system for all works that have the "Trinklied" designation, and then generate an inventory of meter classifications (available in "AMT" records).

```
find /scores -type f -exec grep -l '!!AGN.*Trinklied' "{}" \
   ";" | grep '!!!AMT.*' | sort | uniq -c
```

For the entire database, the output is as follows:

```
  1 !!!AMT: compound duple
  4 !!!AMT: irregular
 14 !!!AMT: simple quadruple
  5 !!!AMT: simple triple
```

There are just 24 drinking songs in the Essen collection and only five are in triple meters. The proportion of drinking songs in triple meters turns out to be no different than the distribution of triple meters in general for German folksongs. In other words, according to the *Essen Folksong Collection*, it is not the case that German drinking songs are more likely to be in triple meters.

Using *find* with the *xargs* Command

As we saw in Chapter 10, the **xargs** command can be used to propagate file names from command to command within a pipeline. Using **xargs** in conjunction with **find** provides a powerful means for finding works that conform to highly complex criteria. For example, the following command identifies all files in the path `/corelli` that contain a change of meter signature:

```
find /corelli -type f -name '*' | xargs grep -c '^\*M[0-9]' \
   | grep -v ':[01]$'
```

The output specifies each filename followed by a colon, followed by the number of meter signatures in the corresponding file. For example, in the following output, the third movement from Opus 1, No. 5 by Corelli is identified as containing 6 meter signatures at different points in the score:

```
/corelli/opus1n5c.krn:6
/corelli/opus1n9a.krn:3
/corelli/opus1n9b.krn:2
/corelli/opus1n9d.krn:2
```

Similarly, the following command identifies all works that contain a change of key signature:

```
find /scores -type f -name '*' | xargs grep -c '^\*k\[' \
   | grep -v ':[01]$'
```

As a further example of the use of **xargs**, consider the following extension of the above pipeline. The **grep -v** command causes only those files containing more than one key signature to be passed. The **sed** command eliminates the colon and the number appended to the filenames. The ensuing **grep -c** counts the number of meter signatures in each file. The final **grep -v** passes only those

filenames containing 2 or more meter signatures.

```
find / -type f -name '*' | xargs grep -c '^\*k\[' | \
   grep -v ':[01]$' | sed 's/:.*$//' | \
   xargs grep -c '^\*M[0-9]' | grep -v ':[01]$'
```

In summary, the above pipeline identifies all scores that contain both a change of key signature as well as a change of meter signature.

The **xargs** command can also be used to process a list of files — where the list has been stored in a file. For example, suppose we used the **find** command to locate all scores in compound meters written for woodwind quintet:

```
find . -name '*' | xargs grep -l '!!!AMT:.*compound' \
| xargs grep -l '!!!AIN: clars cor fagot flt oboe' > scorelist
```

The resulting list of files can be used for further processing. For example, we might search these files for any scores containing changes of key:

```
cat scorelist | xargs grep -c '^\*[A-Ga-g][#-]*:' | grep -v
':[01]$'
```

The output identifies all scores in compound meters written for woodwind quintet that contain changes of key.

Repertories As File Links

Rather than applying commands to files stored in a list, it is often helpful to have all of the files accessible in one location. That is, we might create a directory containing only those score-files that meet our selection criteria. it is often helpful to have all of the files accessible in one location. We might simply make copies of the files in a special directory. However, UNIX systems make it possible to create "links" to files in other directories without having to make duplicate copies of already existing files.

Suppose you wanted to make a directory of all scores containing vocal parts. The following command creates a file (`vocalfiles`) listing all files in the path `/scores` that contain one or more vocal parts:

```
find /scores -exec grep -l '!!!AIN.*vox' "{}" ";" > vocalfiles
```

The contents of `vocalfiles` might look like the following:

```
/scores/bach/cantatas/cant140.krn
/scores/bach/chorales/chor217.krn
/scores/bach/chorales/midi/chor368.hmd
etc.
```

We can create an appropriate new directory using the **mkdir** command.

```
mkdir vocal
```

Next, edit the file containing the list of filenames as follows. Insert **ln -s** prior to each filename, and append the directory name `vocal` at the end of each line.

```
ln -s /scores/bach/cantatas/cant140.krn vocal
ln -s /scores/bach/chorales/chor217.krn vocal
ln -s /scores/bach/chorales/midi/chor368.hmd vocal
etc.
```

(The **-s** option for **ln** is used to create a so-called "symbolic" link.)

Using the **chmod** command, we can make this file executable, and then we can execute it:

```
chmod +x vocalfiles
 ./vocalfiles
```

We now have a new directory whose files contain scores with vocal parts.

Reprise

The **find** command provides a convenient way to traverse through an entire file-system looking for files that conform to specific criteria. In musicological tasks, the **find** command is especially well suited to assembling a repertory of scores that exhibit some characteristic(s) of interest. Multiple selection criteria can be accommodated by using one or more pipes in conjunction with the **grep** command.

For convenience, it is often helpful to create a new directory that holds all of works selected for a study repertory. On UNIX systems, file "links" can be created, so that there is no need to make multiple copies of the same score. This means that several concurrent directory structures can be created without duplicating files. For example, a given score may be accessed in one directory structure via *composer*, in another directory via *instrumentation*, in a third directory via *genre*, and so on.

Chapter 32

The Shell (IV)

In research applications, it is impossible to anticipate all the types of manipulations we might want to carry out. For some tasks, we will need to write our own software to carry out specific operations of interest. Fortunately, many specialized tasks require only a brief program to achieve the goal. The Humdrum tools can be used in conjunction with user-developed software to carry out specific tasks.

Many users will already have some programming ability and will be able to apply this knowledge using their preferred programming language. For those users who have less programming background, it may be useful to learn some basic programming skills. While the shell provides a useful programming environment, for more complex tasks, it is better to use one of the many good programming languages.

For data manipulation tasks comparable to those described in this book, the most appropriate programming language include **perl** and **awk**. The **awk** programming language is especially useful for text processing, retrieving, transforming, reducing, and validating text data. The **perl** programming language provides even more extensive capabilities, but requires a somewhat greater effort to learn. For research-oriented programming, **perl** is the programming language of choice. However, for this brief introduction we will describe features of the **awk** programming language. Awk is a so-called "scripted" language. It is easy to learn but nevertheless quite powerful.

The *awk* Programming Language

Awk programs can be executed from the shell command line. A simple program is the following:

```
awk '{print "hello"}'
```

The **awk** command invokes the awk program interpreter. The material within the single quotes is the actual program. Once the program is started, it is is executed once each time you type the carriage return or ENTER key. To stop the program, simply type control-D (on UNIX systems) or control-Z (on DOS systems).

In the default configuration, an awk program will be executed once for each line of input. If no input file is specified, then "standard input" is assumed. That is, input will come from either data arriving through a pipeline, or data typed at the keyboard.

Automatic Parsing of Input Data

Each line of input data is automatically assigned to the awk variable `$0`. This means that the command

```
awk '{print $0}'
```

will simply echoe each line of input as the output. Similarly, the following command will print each line of input preceded by a colon and a space:

```
awk '{print ": " $0}'
```

For any input line, awk also automatically parses the data into individual tokens or fields. A token is deemed to be any sequence of characters that is separated from other tokens by any blank space such as spaces or tabs. The first data token is automatically assigned to an awk variable `$1`. The second data token is assigned to the variable `$2`, and so on. For example, suppose a program encountered the following input line:

```
243xyz 3    29    #%$    **      Ullyses 234-034
```

The variables would be automatically assigned as follows:

```
$1 = 234xyz
$2 = 3
$2 = 29
$4 = #%$
$5 = **
$6 = Ullyses
$7 = 234-034
```

Given this input, the command

```
awk '{print $2 + $3}'
```

will print the sum of `$2` and `$3`, namely 32.

Arithmetic Operations

Suppose that we have two `**semits` spines as input and we would like to print the semitone difference between the two parts for each sonority. Typically, the higher part is placed in the rightmost spine, so it makes most sense to subtract $1 from $2. Negative numbers mean that the nominally lower part has crossed above the nominally higher part:

```
awk '{print $1 - $2}'
```

In addition to addition and subtraction, other possible arithemtic operators include the slash (/) for division, the asterisk (*) for multiplication, the caret (^) for exponentiation, and the percent sign (%) for modulo arithmetic. Parentheses can be used to clarify the order of operations. For exam-

ple, the following command prints the product of the first and second tokens ($1 * $2) divided by the third token raised to the fourth token power:

```
awk '{print ($1 * $2) / ($3^$4)}'
```

As we have already seen, character strings can also be included in print statements. For example, we might want to print the first and third input tokens separated by a tab:

```
awk '{print $1 "\t" $3}'
```

Conditional Statements

Often we'd like to avoid processing certain records. For example, we might wish to avoid processing barlines. The awk **if** statement can be used to restrict the operation to particular circumstances. Consider the following awk program:

```
awk '{if ($0 !~/^=/) print $1 - $2}'
```

The **if** condition is given in parentheses. The string given between the slashes (`/^=/`) is a regular expression: in this case, it identifies any equals sign that occurs at the beginning of an input line. The tilde means "match" and the exclamation mark means "not". Hence the program means: if the entire line (`$0`) does not match (`!~`) an equals sign occuring at the beginning of the line (`/^=/`), then print the value of the first token minus the value of the second token (`print $1 - $2`).

Awk also provide an **else** condition. The syntax is:

```
if (condition)
    [then] {do something}
    else {do something else instead}
```

For Humdrum inputs, we may want to avoid processing comments and interpretations. Whenever we encounter a comment or interpretation, we might simply echo the input record in the output:

```
awk '{if($0 ~/^[*!]/) {print $0} else {print $1 - $2}}'
```

Sometimes we might simply want to do nothing at all when we encounter a comment or interpretation:

```
awk '{if($0 ~/^[*!]/) {} else {print $1 - $2}}'
```

Recall that input tokens in awk are separated by any blank space such as spaces or tabs. This means that a Humdrum multiple-stop will be treated as containing two or more tokens. We can avoid this situation by explicitly telling awk to assign the "field separate" (FS) to the tab character. For example, the following program prints the value in the third spine of a Humdrum input. Without reassigning the field separator, the third token might be the third element of a multiple-stop in the first spine, or the second element of a multiple-stop appearing in the second spine.

```
awk '{FS="\t"; print $3}'
```

Notice the use of the semicolon to separate individual instructions.

Assigning Variables

Within an awk program, the user can assign and manipulate variables that store particular values. Variables may hold numerical values or they may hold character strings. In the following examples, the value 178 is assigned to the variable 'A'; the value 2.2 is assigned to the variable 'number'; and the character string "Dear Gail" is assigned to the variable 'salutation':

```
A=178
number = 2.2
salutation = "Dear Gail"
```

Named variables can be used for various arithmetic operations. For example:

```
A=178+18
number = 2.2 + A
number_squared = number ^ 2
```

Manipulating Character Strings

Variables holding character strings can be concatenated together. In the following example, after the first three assignments, the variable saluation will contain the character string "Dear Craig":

```
opening = "Dear"
space = " "
name = "Craig"
salutation = opening space name
```

Awk provides a number of built-in functions for manipulating text. One function (**gsub**) carries out global substitutions. The syntax is:

```
gsub("target-string","replacement-string",variable)
```

For example, the following instruction changes all occurrences of X to Y in a variable named string:

```
gsub("X","Y",string)
```

Suppose that we wanted to increment all measure numbers by 1. Let's presume our input contains only a single spine. First we test for the presence of the equal sign at the beginning of the input record. If the input is not a barline, then we simple print the line in the output. Otherwise we: (1) assign the input to the variable barline, (2) eliminate all non-numeric characters using gsub, (3) add one to the remaining numeric value, and (4) output the new number preceded by the equal

sign:

```
awk '{
     if ($0 !~/^=/) {print $0}
     else {
          barline = $1
          gsub("[^0-9]","",barline)
          barline = barline + 1
          print "=" barline
          }
     }'
```

Notice that we are at liberty to add spaces, tabs, and newlines in order to improve the readability of our program.

The *for* Loop

Often we would like to repeat a process for several concurrent spines. For example, suppose we had four spines of **solfa data and we want to output the total number of leading-tones for each sonority. Awk provides a **for** instruction that allows us to cycle through a series of values. The **for**-loop construction has the following syntax:

```
for (initial-value; condition-for-continuing; increment-action)
     {do something repeatedly}
```

In the case of counting the number of leading-tones for each of four spines, our program would be as follows:

```
awk '{
     count = 0
     for (i=1; i<=4; i++)
          {if ($i ~/ti/) count++}
     print count
}'
```

The initial value for the for-loop is 1 (`i=1`); each time the loop is executed the value of `i` is incremented by 1 (`i++`); and the loop continues executing as long as `i` is less-than or equal to 4 (`i<=4`). The value `$i` will take successive values so that the loop will test whether each of $1, $2, $3 and $4 match the regular expression `/ti/`. For each match, the variable `count` is incremented by 1. Finally, the value of count is printed. The count is set to zero each time the program is run (that is, for each line of input).

It would be nice if our program could adapt to inputs containing any number of spines. For each line of input, awk automatically identifies the number of input tokens or fields and stores the value in the varible `NF`. Simply replacing the number 4 by `NF` will achieve our goal. In our revised program we have also added some comments to clarify our code. Like the shell, awk comments consist of material following the octothorpe character (#):

```
awk '{
     # A program to count occurrences of the leading-tone.
     count = 0
     for (i=1; i<=NF; i++)
          {if ($i ~/ti/) count++}
     print count
}'
```

A problem with the above script is that it will attempt to count occurrences of `ti` in Humdrum comments, interpretations, and barlines. We can improve our program by echoing these in the output without processing them. Another refinement makes use of the awk **next** instruction. Whenever a **next** statement is encountered, the program immediately moves on to the next input line and begins processing again from the start of the program.

```
awk '{
     # A program to count occurrences of the leading-tone.
     count = 0
     if ($0 ~/^[!*=]/) {print $0; next}
     for (i=1; i<=NF; i++)
          {if ($i ~/ti/) count++}
     print count
}'
```

Although our output data will consist of a single column (spine) of numbers, it is possible that an input will contain more than one interpretation — and so cause the output to fail to conform to the Humdrum syntax. Rather than simply echoing any interpretation records, we might ensure that only a single interpretation is generated for the output. First, we might look for exclusive interpretations (beginning `**`) and output a suitable interpretation of our own (e.g., `**leading-tones`). In the case of tandem interpretations (beginning with only a single asterisk), we could output a single null interpretation. Similarly, when we encounter a barline, we might ensure that only one barline token is output. Finally, we should remain vigilant for spine-path terminators (`*-`) and ensure that our output is similarly properly terminated. The revised program is as follows:

```
awk '{
     # A program to count occurrences of the leading-tone.
     count = 0
     if ($0 ~/^**/) {print "**leading-tones"; next}
     if ($0 ~/^*-/) {print "*-"; next}
     if ($0 ~/^*[^*]/) {print "*"; next}
     if ($0 ~/^!!/) {print $0; next}
     if ($0 ~/^=/) {print $1; next}
     {
     for (i=1; i<=NF; i++)
          {if ($i ~/ti/) count++}
     print count
}'
```

Of course there are many other features of the awk programming language that we have not de-

scribed here. These features include associative arrays, built-in variables, string-processing functions, user-defined functions, system calls, begin and end blocks, other control-flow statements, and pipes and file manipulations.

Reprise

In this chapter we have introduced some features of the **awk** pattern/action language. A programming language, like **awk** or **perl** can be used to transform data in highly specific and specialized ways. The power of Humdrum is significantly enhanced when users are able to create their own specialized filters.

Chapter 33

Word Sounds

In addition to the meanings conveyed by words, words also provide distinctive sounds that can prove to be musically useful. Poets and composers often arrange or choose texts so that the sequence of sounds create alliteration, onomatopoeia, rhythm, rhyme and other sonorous effects. This chapter introduces the **IPA scheme for representing speech sounds. This representation provides a companion to the **text and **silbe representations discussed in Chapter 27. Various sonorous processes are illustrated.

The **IPA Representation

The Humdrum **IPA scheme provides a way to represent the International Phonetic Alphabet. The **IPA scheme is based on the transliteration scheme developed by linguist Evan Kirshenbaum. The scheme is suitable for representing the basic phonemes found in most of the world's languages. The table below summarizes the **IPA mappings for various phonemes.

@	**schwa**†; as in (unaccented) *banana, collide, alone or (accented) humdrum*
V	**schwa** (IPA symbol: —); as in the British pronunciation of *hut*
R	**R**‡; as in *burn, operation, dirt, urgent*
&	short **a** (IPA symbol: æ); as in *mat, map, mad, gag, snap, patch*
A	**a** (IPA symbol: *a*); as in *bother, cot*, and, with most American speakers, *father, cart*
a	**â**; *father* as pronounced by speakers who do not rhyme it with *bother.*
E	short **e** (IPA symbol: ɛ or ɛ); as in *get, bed, peck, edge*
i	long **e** (IPA symbol: e); as in *beat, greed, evenly, easy*
I	short **i** (IPA symbol: ɪ or *ɪ*); as in *tip, banish, active*
o	**o** as in *oboe, trombone, banjo*
O	**ô** (IPA symbol: o or upside-down 'c'); as in *law, all, shawm*
W	**oe** digraph (IPA symbol: œ); as in the French *boeuf*, German *Holle*
u	**u**; as in *rule, youth, few, ooze*
U	**û** (IPA symbol: *ʊ* or ʊ or *ω*); as in *pull, wood, book*
y	**ue**; as in the German *fullen, hubsch*, or French *rue*
vowel~	following a vowel* indicates a vowel or diphthong pronounced with open nasal passages; as in the French "un bon vin blanc" (`W~ bo~ va~ blA~`)

b	**b** (IPA symbol: b or c); as in *beam, cabin, rob*
d	**d**; as in *deed, dulcimer, ader*
f	**f**; as in *fugue, staff, forte*
g	**g**; as in *guitar, fagot, gig*
h	**h**; as in *hear, ahead, horn*
k	**k**; as in *cook, take, score, ache*
x	**K** (IPA symbol: k); as in the German *ich, Buch*
l	**l**; as in *libretto, Lully, pool*
m	**m**; as in *music, limb, hymn*
n	**n**; as in *no, instrument, blown*
N	**eng** (IPA symbol: 'n' with a tail); as in *sing, fingering, ink*
p	**p**; as in *piano, beeper, lip*
r	**r**; as in *reed, organ, car*
s	**s**; as in *soprano, cymbal, source, bass*
S	**sh** ["esh"] (IPA symbol: ʃ); as in *sharp, crescendo, special, percussion*
t	**t**; as in *tempo, tie, attacca, minuet*
T	**th** ["thorn"] (IPA symbol: *θ*); as in *thin, path, ether*
D	**th** ["eth"] (IPA symbol: *ð*) *as in then, rhythm, smooth*
v	**v**; as in *voice, vivace, live*
w	**w**; as in *we, away*
j	**j**; as in *yes, Johann, cue, onion*
z	**z**; as in *zone, raise, xylophone, jazz*
Z	**zh** ["yogh"§]; as in *measure, vision, azure*
consonant-	following a consonant (l-, n-, m-, or N-)** indicates a consonant preceded by a *schwa* that is pronounced as an independent syllable; as in *battle, mitten, eaten*
consonant;	following a consonant,†† indicates that the front of the tongue is positioned as in the beginning of the word 'yard'
ˆ	preceding phoneme is palatalized
´	primary stress (should precede stressed sound)
,	secondary stress (should precede stressed sound)
%	silence signifier

Summary of ****IPA** *Signifiers*

† The IPA *schwa* is notated as an upside-down 'e'.

‡ The IPA symbol consists of a *schwa* with a hook.

§ The IPA *yogh* is written like a flat-topped numeral '3' that has been lowered in height.

* In IPA such vowels are marked by the presence of a tilde above the vowel.

** In IPA such consonants are marked by the presence of a vertical bar below the consonant.

†† The IPA symbol consists of a superscript letter 'j' either following or hooked beneath the consonant.

Humdrum does not provide a tool for translating from **text or **silbe representations to **IPA. However, there are a number of commercial text-to-phoneme translators available for most common languages.

Alliteration

A common sonorous use of words is found in alliteration where several successive words commence with the same sound. A famous example of alliteration is found at the beginning of William Shakespeare's *Tempest*:

```
**text   **IPA
Full     ful
fathom   f&D@m
five     fAiv
thy      DAi
father   fADR
lies     lAiz
*-       *-
```

Given an **IPA input, occurrences of alliteration can be found by first isolating the initial phoneme for each word using **humsed**. This task requires some additional knowledge about using **humsed**. Both **sed** and **humsed** provide a "back reference" construction that allows users to manipulate a matched expression without knowing the precise matched sequence of characters. The expression to be matched is indicated via parentheses preceded by back-slash characters, i.e., `\(` and `\)`. Several such expressions can be defined and each successive expression is internally labelled with an integer beginning with 1. The marked expression can then be "back-referenced" by using the integer label preceded by a back-slash. Hence, `\1` refers to the first referenced expression. Consider, by way of illustration, the following command:

```
sed 's/^\(.\).*/\1/'
```

In this command, the referenced expression consists of the period (match any single character). Notice that this expression is back-referenced in the replacement string — `\1`. In other words, this **sed** command carries out the following operation: find a single character at the beginning of a line followed by zero or more characters. Replace this entire string by just the first character in the line.

Let's now use this back-reference technique in our alliteration search. First we extract the **IPA spine and use **humsed** to eliminate all but the first character in each data record:

```
extract -i '**IPA' Tempest | humsed 's/^\(.\).*/\1/'
```

The result is:

```
**IPA
f
f
f
D
f
l
*-
```

We can now amalgamate successive initial phonemes by using the **context** command. Suppose we are interested in identifying alliterations where three or more words begin with the same initial phoneme. For this, we would use the **-n 3** option for **context**. Having amalgamated three phonemes on each data record we can use **humsed** to eliminate the spaces between the multiple stops:

```
extract -i '**IPA' Tempest | humsed 's/\(.\).*/\1/' \
    | context -n 3 | humsed 's/ //g'
```

The revised output is:

```
**IPA
fff
ffD
fDf
Dfl
.
.
*-
```

Now we need to identify any data records that contain three identical sigifiers. Once again, we can use the back-reference feature for **humsed**.

```
extract -i '**IPA' Tempest | humsed 's/\(.\).*/\1/' \
    | context -n 3 | humsed 's/ //g'; s/\(.\)\1\1/allit: \1/'
```

The resulting output is:

```
**IPA
allit: f
ffD
fDf
Dfl
.
.
*-
```

Let's add one further refinement which illustrates yet another feature provided by **sed** and **humsed**. Recall that operations such as substitions (`s`) and deletions (`d`) can be preceded by a regular expression that limits the operation only to those lines that match the expression. For example, the command `sed '/=/s/[0-9]//g'` will eliminate all numbers found on lines containing an equals sign. The leading regular expression can be followed by an exclamation mark which reverses the sense of the action. For example, the command `sed '/=/!s/[0-9]//g'` will eliminate all numbers *except* those found on lines containin an equals sign.

This feature can be usefully applied in our alliteration task to eliminate all other data in our spine except alliteration markers. Our final revised pipeline transforms non-alliteration data to null tokens:

```
extract -i '**IPA' Tempest | humsed 's/\(.\).*/\1/' \
    | context -n 3 | humsed 's/ //g'; s/\(.\)\1\1/allit: \1/; \
    /allit/!s/.*/./'
```

The final output is:

```
**IPA
allit: f
.
.
.
.
.
*-
```

Classifying Phonemes

Linguists have devised innumerable ways for classifing phonemes. For example, phenomes such as *f*, *s*, *sh*, *th*, *v*, etc. are classified as fricatives. Bi-labial plosives include the *p* and *b* sounds. The *m* and *n* sounds are classified as nasals. And so on.

For some tasks, it is often useful to reduce the phonemes to phonetic classes. For example, in our example from Shakespeare's *Tempest*, the 'th' in 'thy' is part of the fricative alliteration.

In Chapter 22 we saw how **humsed** can be used to classify things. A simple reassignment script can be defined which collapses the various phonemes into a smaller set of phonetic classes. For example, a suitable script might contain the following assignments:

```
s/[bdtk]/<PLOS>/g
s/[mn]/<NASL>/g
s/[fsSTDv]/<FRIC>/g
etc.
```

Classifying phonemes in this way will allow us to broaden our searches for alliterative passages.

Properties of Vowels

Vowels are particular important in music since notes can be sustained only by increasing the duration of the vowels. In speech, more time is occupied by vowel sounds than by consonants. In vocal music, an even greater proportion of the time is taken up by vowels. This means that the quality of the vowels can have a marked impact on the overall timbre or tone color of a work. For example, a long note sustained with an 'ee' can sound very different from the same note sung with a sustained 'ah'.

Like consonants, vowels can also be classified in many ways. One common classifying dimension is the front/back distinction. The vowel 'oo' is a front position vowel whereas the vowel 'ah' is a back position vowel. Similarly, vowels can be classified according to the high/low distinction.

The vowel 'ee' is a high vowel whereas the vowel 'oh' is a low vowel.

A preponderance of high vowels is often associated with sarcasm, irony or humor. Taunting sounds made by children ("nya, nya ...") commonly use high vowels mixed with nasals. Similarly, high/nasal vocal sounds are often used by comedians and actors to produce a 'funny' voice.

Suppose we want to test the idea that a certain piece in a Gilbert and Sullivan operetta exhibits a preponderance of high vowels. We might begin by creating a re-assignment file where the estimated height of each vowel is given a value between 1 (low) and 10 (high). We can estimate the overall average vowel height for a piece by averaging these values together. The basic pipeline will extract the pertinent **IPA spine, eliminate all non-vowel phonemes, add spaces between each vowel, and then assign estimated heights to each vowel. Finally, non-data records are eliminated using **rid** and the data values averaged using the **stats** command:

```
extract -i '**IPA' Penzance | humsed 's/[^@VR&AaEiIoOWuUy]//'\
    | humsed 's/./& /g; s/  / /; s/ $//' | humsed -f vowel.map\
    | rid -GLId | stats
```

This procedure can be repeated for several movements or pieces to provide a contrast for the piece of interest.

Vowel Coloration

When translating vocal texts from one language to another, it is often difficult for translators to preserve the vowel coloration. A vocal work can be considerably maligned if a prominent (high/long) note is changed from an 'ah' sound to an 'ee' sound.

Suppose we want to determine which of several English translations of a song by Schubert best preserves the vowel coloration. As above, let's limit our notion of coloration to vowel height. (Of course any other similarity mapping or dimension can be used.) As in our Gilbert and Sullivan example, we could simply compare the overall vowel height for the original Schubert song with each of the translations. However, not all notes are equally important. In the first instance, the vowels on longer sustained notes will be more noticeable than the vowels attending shorter notes. A simple remedy is to use the **timebase** command to expand the input so that longer notes are proportionally more influential in our measure of overall vowel height. We can use **ditto** to repeat sustained vowels:

```
timebase -t 16 Schubert | extract -i '**IPA' \
    | humsed 's/[^@VR&AaEiIoOWuUy]//' | humsed 's/./& /g; \
    s/  / /; s/ $//' | humsed -f vowel.map | rid -GLId | stats
```

Since translators have plenty of other issues to consider when translating a vocal text, we might focus our comparisons solely on a small collection of especially important notes. We might for example use a longer value for **timebase**. Alternatively, we might use the Humdrum **accent** command (described in Chapter 25) to identify notes have a particularly high noticeability.

Rhymes and Rhyme Schemes

Rhymes are common poetic devices throughout the world's cultures. Rhymes involve the use of similar or identical word-final phonemes. Typically, rhymes are based on the final phonemes of phrase-terminating words, but rhymes commonly occur in mid-phrase and other positions in poetry from various cultures. Consider the rhymes in the following traditional nonsense verse:

> We're all in the dumps,
> For diamonds are trumps,
> The kittens are gone to St. Paul's
> The babies are bit,
> The moon's in a fit
> And the houses are built without walls.
> -Anon.

Suppose we want to automatically identify the rhyme scheme for this (or some other) text. Our first order of business is to identify phrase-terminating points. Let's assume we already have some phrase indicators (via curly braces { }). Our input might begin as follows:

```
**text   **IPA
We're    {wRr
all      Al
in       In
the      D@
dumps,   d@mps}
etc.
```

Using **extract**, **context** and **rid** we can isolate each poetic phrase:

```
extract -i '**IPA poem | context -b { -e } | rid -GLId
```

The result is as follows:

```
{wRr Al In D@ d@mps}
{foR dAim@nds Ar tr@mps}
{D@ kIt@ns Ar gAn tu seint pAUls}
{D@ beibiz Ar bIt}
{D@ munz In @ fIt}
{&nd D@ h&uz@z Ar bIlt wITAut wAUls}
```

The rhyming portion of words typically consist of a final vowel plus any subsequent consonants. We can isolate these phonemes using **sed**. Notice our use of back reference to preserve the final phonemes:

```
... | sed 's/.*\([@VR&AaEiIoOWuUy][^@VR&AaEiIoOWuUy]*}$\)/\1/'
```

The resulting output is:

```
@mps}
@mps}
Uls}
It}
It}
Uls}
```

A little further processing can remove the closing braces using **sed**, and eliminate the duplicate lines using **sort** and **uniq**.

```
... | humsed 's/}//' | sort | uniq
```

The output can then be changed into a set of substitutions for a **humsed** script. A suitable file would contain the following substitutions:

```
s/.*@mps$/A/
s/.*It$/B/
s/.*Uls$/C/
s/.*/./
```

This script will label all words ending with "umps" to 'A'. Word ending with "its" will be labelled 'B', and so on. All other words will be output as null tokens. Using this script, a suitable pipeline for processing our original file would be as follows:

```
extract -i '**IPA poem | context -b { -e } | humsed 's/}//' \
    | humsed -f rhyme | rid -GLId
```

The corresponding output would indicate the rhyme scheme for this poem:

```
A
A
B
C
C
B
```

Note that the entire analytic procedure can be placed in a shell script and applied to any input containing **IPA text. The following script adds a number of refinements.

```
# RHYME
#
# This script determines the rhyme scheme for an input file containing
# an **IPA spine.  This script assumes that the input contains curly
# braces indicating phrase endings.
#
# USAGE:  rhyme <filename>

extract -i '**IPA' $1 | extract -f 1 | context -b { -e } | rid -GLId \
   | sed 's/.*\([@VR&AaEiIoOWuUy][^@VR&AaEiIoOWuUy]*}$\)/\1/' \
   | sort | uniq | sed 's/^/s\/.*/; s/$/\/XXX\//' \
```

```
    | awk 'BEGIN {alphabet[1]="A"; alphabet[2]="B"; alphabet[3]="C";
       alphabet[4]="D"; alphabet[5]="E"; alphabet[6]="F";
       alphabet[7]="G"; alphabet[8]="H"; alphabet[9]="I";
       alphabet[10]="J"; alphabet[11]="K"; alphabet[12]="L";
       alphabet[13]="M"; alphabet[14]="N"; alphabet[15]="O";
       alphabet[16]="P"; alphabet[17]="Q"; alphabet[18]="R";
       alphabet[19]="S"; alphabet[20]="T"; alphabet[21]="U";
       alphabet[22]="V"; alphabet[23]="W"; alphabet[24]="X";
       alphabet[25]="Y"; alphabet[26]="Z"; alphabet[27]="ERROR"}
       {temp=$0
       gsub("XXX",alphabet[NR],temp)
       print temp
       }' > rhyme.sed.$$
extract -i '**IPA' $1 | extract -f 1 | context -b { -e } | rid -GLId \
    | sed -f rhyme.sed.$$
rm rhyme.sed.$$
```

Reprise

By focusing on phonetic signifiers, the **IPA representation provides opportunities for analyzing many sonorous aspects of vocal sounds — including alliteration, vowel coloration, rhyme, and other effects. Although we did not illustrate it in this chapter, the **IPA representation can be used in conjunction witht the **silbe representation to characterize complex aspects of rhythm and rhyme in vocal texts.

Chapter 34

Serial Processing

Humdrum provides a handful of specialized tools for serial and serial-inspired analytic processing. In this chapter we introduce the **reihe**, **pcset**, **iv**, **pf** and **nf** commands These commands reveal their greatest power when used in conjunction with Humdrum tools we have already encountered — such as **context**, **humsed** and **patt**.

The chapter culminates with a script that automatically identifies 12-tone row variants in complex orchestral scores. The general approach is instructive for applications beyond serial analysis.

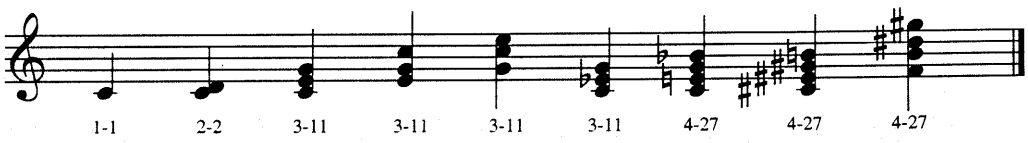
Pitch-Class Representation

In set theoretic applications it is common to use pitch-class representations. The **pc** command can be used to transform pitch-related representations (such as **pitch, **freq and **kern) to a conventional pitch-class notation where pitch-class C is represented by the value zero. With the **-a** option, **pc** will generate outputs where the pc values '10' and '11' are rendered by the alphabetic characters 'A' and 'B' respectively. Using the alpha-numeric pc representation is recommended; it proves to be especially convenient for searching tasks since, otherwise, the characters 1 and 0 do not uniquely specify a single pitch-class type.

The *pcset* Command

A common set theoretic task is identifying occurrences of various pitch-class set forms. Figure 34.1 identifies the set forms for several sample vertical sonorities. These forms are identified using standard numerical designations (see Forte, 1973; Rahn, 1980). Set forms are insensitive to transposition, pitch-inversion, and pitch spelling so all major and minor triads are identified as pitch-class set 3-11. Similarly, the dominant seventh chord and 'Tristan' chord are similarly related by inversion so both are identified as pc set 4-27.

Figure 34.1. Examples of PC set forms.



The **pcset** command identifies pitch-class sets from **pc or **semits input. Illustrated below are the corresponding **kern, **pc and **pcset representations for Example 34.1.

```
**kern          **pc        **pcset
4c              0           1-1
4c 4d           0 2         2-2
4c 4e 4g        0 4 7       3-11
4e 4g 4cc       4 7 0       3-11
4g 4cc 4ee      7 0 4       3-11
4c 4e- 4g       0 3 7       3-11
4c 4e 4g 4b-    0 4 7 10    4-27
4c# 4e# 4g# 4b  1 5 8 11    4-27
4f 4b 4dd# 4gg# 5 11 3 8    4-27
*-              *-          *-
```

Suppose we wanted to identify the pc sets for successive vertical sonorities in the first movement of Webern's Opus 24 concerto. First, we translate the input to a pitch-class representation, and then we apply the **pcset** command:

```
pc opus24 | pcset
```

Of course this command will only identify the set forms for pitches that have concurrent attacks. If any pitch is sustained, **pcset** won't know that some null tokens indicate sustained pitch activity. We can rectify this by using the **ditto** command (Chapter 15) to fill-out the null tokens:

```
pc opus24 | ditto -s ^= | pcset
```

If we wanted, we could assemble the resulting **pcset spine to the original input. This would allow us to search for particular patterns that are coordinated with certain pitch-class sets. For example, we might be interested in comparing the pitch-class sets that coincide with the beginnings of slurs/phrases versus those pitch-class sets coinciding with the ends of slurs/phrases. First we generate the **pcset spine:

```
pc opus24 | ditto -s ^= | pcset > opus24.pcs
```

Then we assemble this spine to the original input score:

```
assemble opus24 opus24.pcs > opus24.all
```

Now we can search for data records containing phrase ('{}') or slur ('()') markers. Using **yank -m ... -r 0** rather than **grep** assures that the output retains the Humdrum syntax (see Chapter 12). Maintaining the Humdrum syntax will allow us to use **extract** to isolate just the **pcset data. Finally, we create an inventory of the pc sets. The process is repeated — once for beginning slurs/phrases, and once for ends of slurs/phrases.

```
yank -m '[{(]' -r 0 opus24.all | extract -i '**pcset' \
    | rid -GLId | sort | uniq -c
yank -m '[)}]' -r 0 opus24.all | extract -i '**pcset' \
    | rid -GLId | sort | uniq -c
```

Two pitch-class set inventories will be generated: one inventory for the beginnings of phrases/slurs and one for phrase/slur endings.

Incidentally, the **pcset** command supports a **-c** option that can be used to generate the set *complement* rather than the principal set form.

Prime Form and Normal Form

The 3-11 set form designates both the major and minor chords (since they are symmetrical). In order to distinguish symmetrical forms, it is sometimes useful to represent pitch-class sets using either *prime form* (the **pf** command) or *normal form* (the **nf** command).

Suppose we wanted to count the proportion of phrase endings in music by Alban Berg where the phrase ends on either a major or minor chord. First, we locate all works composed by Berg:

```
BERG=`find /scores -type f -exec grep -l '!!!COM.*Berg,' "{}" ";"`
```

Let's put all the Berg works in a single temporary file:

```
cat $BERG > AllBerg
```

Next we generate the normal set forms:

```
pc AllBerg | ditto -s ^= | nf > AllBerg.nf
```

Assemble the **nf spine with the original scores:

```
assemble AllBerg AllBerg.nf > AllBerg.all
```

Now we're ready to count the number of phrases that match the appropriate patterns. First, count the total number of phrases:

```
grep -c '}' AllBerg.all
```

Count the number of phrases that end with a major chord:

```
grep -c '}.*\t(047)' AllBerg.all
```

And count the number of phrases that end with a minor chord:

```
grep -c '}.*\t(037)' AllBerg.all
```

Interval Vectors Using the *iv* Command

Interval vectors identify the frequency of occurrence of various interval-classes for a given pitch-class set. The **iv** command generates the six-element interval vector for any of several types of inputs — including semitones (**semits), pitch-class (**pc), normal form (**nf), prime form

(**pf), and pitch-class set (**pcset). The following example shows several different pitch-class sets, their corresponding pitch-class sets and (right-most spine), the associated interval vector.

```
**pc        **pcset    **name             **iv
0           1-1        tone               <000000>
0 2         2-2        major second       <010000>
0 3 7       3-11       minor triad        <001110>
0 4 7       3-11       major triad        <001110>
0 4 7 10    4-27       dominant seventh   <012111>
1 5 8 11    4-27       dominant seventh   <012111>
*-          *-         *-                 *-
```

Suppose we wanted to determine whether Arnold Schoenberg tended to use simultaneities that have more semitone (interval-class 1) relations and fewer tritone (interval-class 6) relations. As before, we might translate his scores to pitch-class notation, fill-out the sonorities using **ditto**, and then determine the associated interval vectors for each sonority. Interval vectors without semitone relations will have a zero in the first vector position (i.e., <0.....>) whereas interval vectors without tritone relations will have a zero in the last position (i.e., <.....0>).

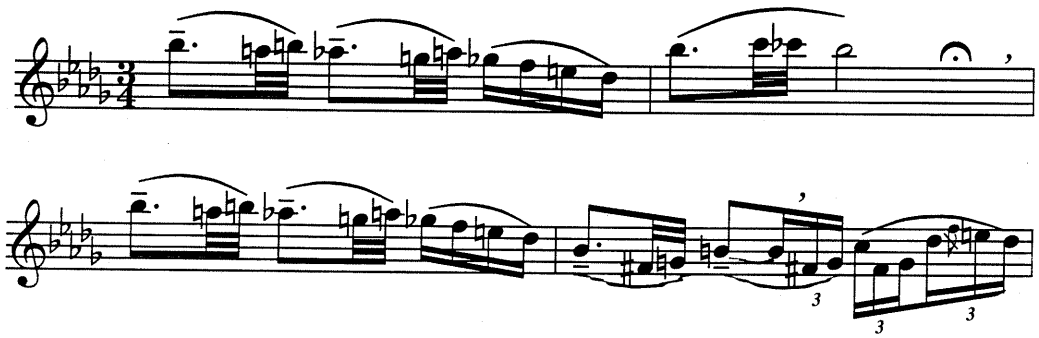
```
pc schoenberg* | ditto -s ^= | iv | grep -c '<0.....>'
pc schoenberg* | ditto -s ^= | iv | grep -c '<.....0>'
```

Segmentation Using the *context* Command

So far, we have processed only "vertical" sets of concurrent pitches. In set-theory analyses, there are many other important ways of "segmenting" the musical pitches into pitch-class sets. As we saw in Chapter 19, the **context** command provides a useful way of grouping together successive data tokens.

Suppose, for example, we wanted to analyze set forms in Claude Debussy's *Syrinx* for solo flute. The opening measures are shown in Example 34.1.

Example 34.1. From Claude Debussy, *Syrinx* for flute.



There are a number of ways we might want to try segmenting the melodic line. One possibility is to regard slurs or phrases as indicating appropriate groups. Recall that the **-b** and **-e** options for **context** are used to specify regular expressions that match the beginning and end (respectively) of the context group: We can invoke an appropriate **context** command, translate the output to a pitch-

class representation, and then use the **pcset** command to identify the set names:

```
context -b '[{(]' -e '[})]' syrinx | pc | pcset
```

Perhaps we might consider gathering groups of three successive notes together, and then generating an inventory of the set forms associated with such a segmentation:

```
context -n 3 -o '[=r]' syrinx | pc | pcset | rid -GLId \
    | sort | uniq -c
```

Another possibility is to treat rests as segmentation boundaries.

```
context -e r syrinx | pc | pcset
```

When a work consists of more than one instrument or part, useful segmentations can be made by extracting each instrument individually, using **context** to generate musically-pertinent sets, and then assembling all of the **pcset spines into a single file.

The *reihe* Command

Twelve-tone music raises additional analysis issues. Variants of a tone-row can be generated using the **reihe** command. Given some input, **reihe** will output a user-specified transformation. Options are provided for prime transpositions (**-P** option), for inversions (**-I** option), for retrogrades (**-R** option) and for retrograde-inversions (**-RI** option).

Inputs do not have to be 12-tone rows. The 5-tone row used in Igor Stravinsky's "Dirge-Canons" from *In Memoriam Dylan Thomas* is as follows:

```
**pc
2
3
6
5
4
*-
```

The following command will generate a prime transposition of the tone-row so that it begins on pitch-class 6:

```
reihe -P 6 memoriam
```

The result is:

```
**pc
6
7
10
```

```
9
8
*-
```

Generating the inversion beginning at pitch-class 2 would be carried out using the following command.

```
reihe -a -I 2 memoriam
```

The **-a** option causes the values '10' and '11' to be rendered alphabetically as 'A' and 'B'.

The **reihe** command also provides a *shift* operation (**-S**) that is useful for shifting the serial order of data tokens forward or backward. Consider the following command:

```
reihe -S -1 memoriam
```

This shifts all of the data tokens back one position so the data begins with the second value in the input, and the first value is moved to the end:

```
**pc
3
6
5
4
2
*-
```

The shift option for **reihe** can be used to shift *any* type of data — not just pitches of pitch-classes. For example, one might use the shift option to rotationally permute dynamic markings, text, durations, articulation marks, or any other type of Humdrum data. In Chapter 38 we will see how the shift option for **reihe** can be effectively used in many applications apart from serial analysis.

Generating a Set Matrix

The first step in automated row-finding is to generate a set matrix of all the set variants. Typically, the user begins with a hypothesized tone row. Suppose the tone-row was stored in a file called `primerow`. From this we can generate the entire set matrix. There is a Humdrum **matrix** command that automatically generates a set matrix, but let's create our own script to see how this can be done.

The following script uses the **reihe** command to generate each set form. Each form is stored in a separate file with names such as `I8` and `RI3`. There are two noteworthy features to this script. Notice that the alphanumeric system (**-a** option) is used — so the values 'A' and 'B' are used rather than '10' and '11'; this will facilitate searching. Also notice that our script provides an option that allows us to specify *partial* rows: that is, we can store (say) only the first 5 notes in each tone row file. This feature will also prove useful when doing an automatic search.

```
#####################################################################
#                              MATRIX
# This script generates a tone-row matrix for a specified prime row.
# The -n option is used to specify the number of pitches to be out-
# put in each row-file (e.g. the first 7 pitches of a 12-tone row).
#
# Usage: matrix -n N primerowfile
#
if [ "x$1" != "x-n" ]
then
     echo "-n option must be specified."
     echo "USAGE:   matrix -n number primerowfile"
     exit
fi
if [ ! -f $3 ]
then
     echo "File $3 not found."
     echo "USAGE:   matrix -n number row-file"
     exit
fi
# Generate the primes, inversions, retrograde, etc:

X=0
while [ $X -ne 12 ]
do
     reihe -a -P $X $3 | rid -GLId | head -$2 > P$X
     reihe -a -I $X $3 | rid -GLId | head -$2 > I$X
     reihe -a -R $X $3 | rid -GLId | head -$2 > R$X
     reihe -a -RI $X $3 | rid -GLId | head -$2 > RI$X
     let X=$X+1
done
```

For any given input, the above script produces 48 short files named P0, P1, ... I0, I1 ... R0, R1 ... RI10, RI11.

Locating and Identifying Tone-Rows

Each of the row variant files can be used as a template for the **patt** command (see Chapter 21). The following "rowfind" script shows how the Humdrum tools can be coordinated to carry out an automatic search and identification of tone row variants for some score.

The first part of the script simply checks to ensure that all of the row variant files are present:

```
#####################################################################
#                              ROWFIND
#
# This script carries a preliminary tone-row search in a specified
# score.  It assumes that a complete set of set-variant files exists
# in the current directory, named P0-P11, I0-I11, R0-R11, and RI0-RI11.
#
# This script puts a file named "analysis" which may be assembled
```

```
# with the original input file.
#
# Invoke:
#       rowfind scorefile
#
# Check that the specified input file exists:
if [ ! -f $1 ]
then
     echo "rowfind: ERROR: Input score file $1 not found."
     exit
fi
# Also check that the row-variant files exist:
X=11
while [ $X -ne -1 ]
do
     if [ ! -f P$X ]
     then
          echo "rowfind: ERROR: Row file P$X not found."
          exit
     fi
     if [ ! -f I$X ]
     then
          echo "rowfind: ERROR: Row file I$X not found."
          exit
     fi
     if [ ! -f R$X ]
     then
          echo "rowfind: ERROR: Row file R$X not found."
          exit
     fi
     if [ ! -f RI$X ]
     then
          echo "rowfind: ERROR: Row file RI$X not found."
          exit
     fi
     let X=$X-1
done
```

The following two lines of the script prepare the input score for searching. Specifically, the score is transformed to pitch-class notation (using **pc**) and then all rests are changed to null tokens using the **humsed** command. Notice the use of the **-a** option for **pc** in order to use the alpha-numeric pitch-class representation.

```
pc -at $1 > temp.pc
humsed 's/r/./g' temp.pc > score.tmp
```

The main searching task is done by **patt**. The **patt** command is executed 48 times — once for each row variant. The **-t** (tag) option is used so that a **patt output is generated. Each time a match is made the appropriate name (e.g. P4) is output in the spine. The **-s** option is used to skip barlines and null data records when matching patterns. The **-m** option invokes the multi-record matching mode — which allows **patt** to recognize row statements where several nominally successive pitches are collapsed into a vertical chord:

```
# Search for instances of each tone-row variant.
X=0
while [ $X -ne 12 ]
do
   patt -s '=|^\.(\t\.)*$' -f P$X -m score.tmp -t P$X \
      | extract -i '**patt' > P$X.pat
   patt -s '=|^\.(\t\.)*$' -f I$X -m score.tmp -t I$X \
      | extract -i '**patt' > I$X.pat
   patt -s '=|^\.(\t\.)*$' -f R$X -m score.tmp -t R$X \
      | extract -i '**patt' > R$X.pat
   patt -s '=|^\.(\t\.)*$' -f RI$X -m score.tmp -t RI$X \
      | extract -i '**patt' > RI$X.pat
   let X=$X+1
done
```

Each of the above 48 **patt** searches resulted in a separate temporary output file. It would be convenient to reduce all 48 **patt spines into a single aggregate spine. This can be done using the **assemble** and **cleave** commands:

```
# Now we have a lot of files to assemble:
assemble P*.pat > prime.pat
cleave -d ' ' -i '**patt' -o '**rows' prime.pat > analysis.1

assemble I*.pat > inversion.pat
cleave -d ' ' -i '**patt' -o '**rows' inversion.pat > analysis.2

assemble R*.pat > retro.pat
cleave -d ' ' -i '**patt' -o '**rows' retro.pat > analysis.3

assemble RI*.pat > retroinv.pat
cleave -d ' ' -i '**patt' -o '**rows' retroinv.pat > analysis.4

assemble analysis.[1-4] > temp
cleave -d ' ' -i '**rows' -o '**rows' temp > analysis.out

# Finally, clean up some temporary files:

rm [PRI][0-9].pat [PRI]1[01].pat RI[0-9]*.pat temp.pat
rm analysis.[1-4] temp temp.pc score.tmp
```

There are a few subtleties and problems that deserve mention about our **rowfind** script. In general, shorter patterns are easier to find than longer patterns. Since row statements tend to be unique after the first 4 or 5 notes, it is preferable to clip the row patterns used as templates. Reducing the length of the templates can lead to "false hits" — but these tend to be infrequent and are easily recognized.

Applied to an entire multi-part score, **rowfind** may miss concurrent row statements due to interposed notes appearing in an irrelevant instrument or part. This problem can be avoided by first extracting individual parts and running **rowfind** on each part separately. (The results can then be amalgamated using **assemble** and **cleave**.) On the other hand, searching instruments separately can mean that row statements crossing between instruments may be missed. This problem can be addressed by extracting pairs and groups of instruments and analyzing them together.

For a complex work like Webern's Opus 24 Concerto, this strategy of analyzing both individual instruments and groups of instruments works very well.

Reprise

In this chapter we have discussed several tools related to set theory analysis. These include the **pc** (pitch-class) command, the **pcset** command (for identifying set-forms), and the **reihe** command (for generating set variants).

We have seen how general tools like **context** can be used to carry out segmentation of some score. Similarly, we have seen how the **patt** command can be used to identify tone-row statements.

Two scripts were described in this chapter: **matrix** and **rowfind.** These demonstrated how the tools may be coordinated to carry out various automated processes.

This chapter has only scratched the surface regarding the types of pertinent serial-related manipulations that might be pursued. For example, much more sophisticated approaches to segmentation can be created by using some of the layer techniques described in the next Chapter. Similarly, the pattern searches could easily be expanded to look at other parameters typical of "complete serialism" — such as durations, dynamics, articulation marks, and so on.

Chapter 35

Layers

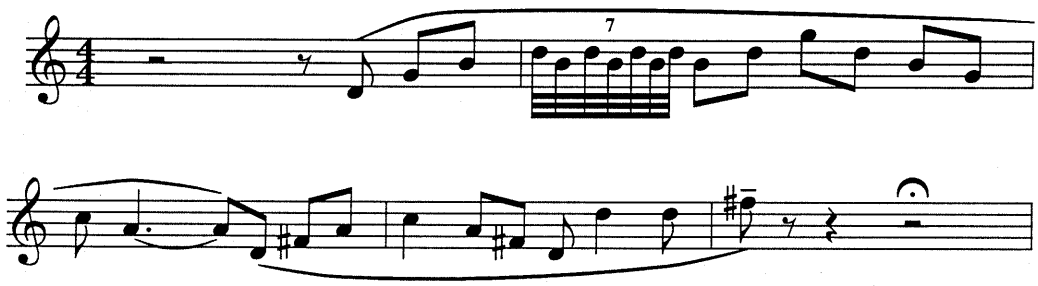
In Chapters 11 and 15 we examined different kinds of intervals, including both harmonic and melodic intervals. A number of different types of intervals were distinguished and we learned how to calculate such intervals. One type of melodic interval mentioned in Chapter 11 is the *distance interval* — an interval between pitches which are separated by intervening musical materials. In this chapter we consider more sophisticated ways of determining distance intervals. These types of intervals are the foundation of various notions of hierarchies or "layers" of pitch analysis.

This chapter also visits a related issue of implied harmony. Many melodic passages outline clear harmonic progressions which are also implicated in layer-related analyses.

Implied Harmony

Example 35.1 shows a two-phrase trumpet solo from Aaron Copland's *El Salon Mexico*. Harmonic progressions may be evident only when arpeggiated figures are collapsed. In this case, an implicit harmony may is evident where a G major chord is followed by a D dominant seventh chord. The barlines provide convenient ways of parsing the harmonies.

Example 35.1 Aaron Copland, *El Salon Mexico*.



A **kern encoding of the passage is given below:

```
!!!COM: Copland, A.
!!!OTL: El Salon Mexico
```

```
**kern
*Itromp
*clefG2
*k[]
*M4/4
=29
2r
8r
{8d
8g
8b
=30
28dd
28b
28dd
28b
28dd
28b
28dd
8b
8dd
8gg
8dd
8b
8g
=31
8cc
[4.a
8a]}
{8d
8f#
8a
=32
4cc
8a
8f#
8d
4dd
8dd
=33
8ff#}~
8r
4r
2r;
=
*-
```

We can collapse the arpeggiated chords using the **context** command:

```
context -b = -o = copland
```

Identify the chords is facilitated by using the pitch-class (**pc) representation described in Chapter 34.

```
context -b = -o = copland | pc -a | rid -d
```

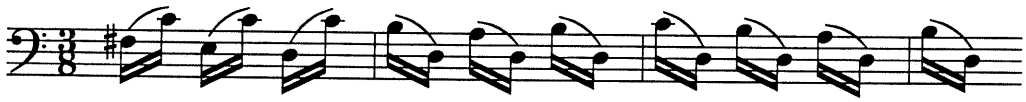
The corresponding output is:

```
!!!COM: Copland, A.
!!!OTL: El Salon Mexico
**pc
*Itromp
*clefG2
*k[]
*M4/4
r r 2 7 B
2 B 2 B 2 B 2 B 2 7 2 B 7
0 9 9 2 6 9
0 9 6 2 2 2
6 r r r
*-
```

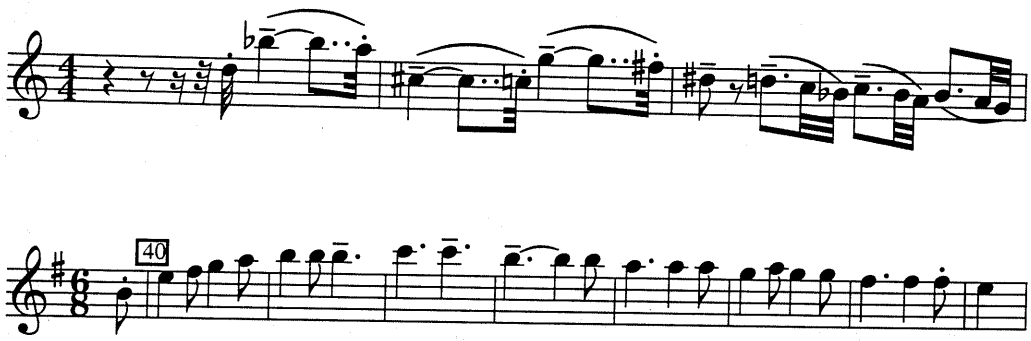
In order to identify these as G major and D dominant chords it would be convenient to reduce the sets to (2,7,B) and (0,2,6,9) respectively. For this task, we can use a The following awk script eliminates repeated tokens within a record: (huniq: We might call this script **huniq** since it acts like a horizontal version of the **uniq** command:

```
awk '{
     # A script to eliminate repeated tokens within a record.
     if ($0 ~ /^[!*]/) {print $0; next}
     else
          { array[$1] = line = $1
          for (i=2; i<=NF; i++)
               {
               if (array[$i] == "") {array[$i]=$i; line = line
               " " $i}
               }
          print line
          for (i in array) delete array[i]
          }
     }' $1
```

Example 35.1 J.S. Bach, "Gigue" from *Suite No. 3* for solo 'cello (excerpt).



```
**kern
*M3/8
=88
(16F#
16c)
(16E
16c)
(16D
16c)
=89
(16B
16D)
(16A
16D)
(16B
16D)
=90
(16c
16D)
(16B
16D)
(16A
16D)
=91
(16B
16D)
```



Chapter 36

Sound and Spectra

Music is a sonic art and no analytic toolkit would be complete with considering the representation and manipulation of sound-related information. In this chapter we introduce some special-purpose tools related to sound analysis, sound synthesis, and auditory perception. We have already encountered the `**freq` and `**cents` representations in Chapter 4. Much of this chapter will center on the `**spect` representation. Three tools will be discussed in connection with `**spect`: the **spect**, **mask** and **sdiss** commands. **Spect** accesses a database of analyzed instrument tones to generate harmonic spectra for all notes for various orchestral instruments over their complete ranges; **mask** can be used to modify a spectrum so that masked frequencies are attenuated in a manner that simulates human hearing; **sdiss** characterizes the degree of sensory dissonance for arbitrary sonorities.

In addition, we will consider how Humdrum data can be used in conjunction with non-Humdrum tools — such as digital sound editors, spectral analysis tools, and general signal processing software, In particular, we will discuss the **kern2cs** command which generates score data for the popular *Csound* digital sound synthesis language.

The ***spect* Representation

A useful predefined sound-related representations that in Humdrum is the `**spect` scheme. The `**spect` representation is used to represent successive acoustic spectra. Each data record represents a complete spectrum specified as a set of concurrent discrete frequency components. Each frequency component in the spectrum is represented by a pair of numerical values separated by a semicolon (;). These paired values encode the frequency and amplitude for a single spectral component. Frequency values are positive values representing *hertz*. Amplitude values are positive values representing the sound pressure level in decibels (dB SPL). Most sonorities consist of more than one pure tone component, so `**spect` data records typically encode a number of multiple stops.

Example 36.1 shows a sample document containing five spectra and a barline. The first data record encodes an ambient spectrum ("silence") represented by the upper-case letter 'A'. Following this are two spectra, each consisting of three spectral components: the first spectrum consists of a 261 Hz tone at 47 dB SPL, as well as frequencies at 523 Hz and 785 Hz at 57 dB SPL and 35 dB SPL, respectively. Following the barline are two data records that represent two different amalgamations of the preceding two three-component spectra. Notice that these two spectra are

identical; only the order of the components differs. In the CR**spect representation there is no special requirement that the spectral components be encoded in any particular order. However, it is often convenient to have the components assembled from left to right in ascending frequency order. This can be achieved by passing an input through the **spect** command.

Example 36.1

```
**spect
A
261;47 523;57 785;35
330;57 659;35 989;27
=1
261;47 523;57 785;35 330;57 659;35 989;27
261;47 330;57 523;57 659;35 785;35 989;27
```

The SHARC Database and *spect* Command

More commonly, the **spect** command is used to generate a `**spect` (acoustic spectral data) output from a `**semits` score input. The **spect** command recognizes instrument tandem interpretations (e.g., *Iclarinet) and fetches a corresponding spectral data file (e.g., clarinet.spe). These files are derived from the SHARC database of music instrument spectra created by Gregory Sandell (1991). The file contains precise spectral measurements for recordings of the instrument playing each note throughout the instrument's range. Suppose that the input to **spect** contains the note F#5 for oboe (i.e., semits value 18). Then **spect** will retrieve the spectral information for a recording of an oboe playing F#5 and add it to the composite spectrum for the particular sonority.

If more than one instrument is playing concurrently, then **spect** will generate an output record representing the aggregate of all the spectral components generated by all of the sounding instruments.

The SHARC database includes most orchestral instruments including piccolo, E-flat clarinet, contrabassoon, etc. The database also includes selected Medieval instruments such as the soprano crumhorn and alto shawm. Timbres for string instruments are distinguished according to different playing methods including arco, vibrato, non-vibrato, pizzicato, mute, and martello.

The *mask* Command

Masking is the tendency for sounds to obscure one another. In many cases, masking may cause a sound to become completely inaudible. This means that the physical presence of a frequency component in a spectrum does not necessarily mean that the component is audible to the listener. Especially in the case of complex orchestral sonorities, many of the acoustically present components are irrelevant for human listeners. A clear demonstration of this effect is evident in digital sound recordings that have been processed using the JPEG compression scheme. JPEG-encoded audio is indistinguishable from the original uncompressed audio, yet the scheme eliminates those aspects of the sound which are masked.

The Humdrum **mask** command implements a common masking algorithm. It accepts as input any

`**spect` data, and for each sonority modifies the spectrum so that masked frequencies are attenuated accordingly. So-called "forward" and "backward" masking are not taken into account in this utlity. No options are provided, and the command is invoked as follows:

`mask` <*inputfile*> > <*outputfile*>

Both the output and input to the **mask** command are `**spect` representations. A tandem interpretation (`*masked` is added to the output to indicate that the sonorities already reflect the influence of masking. Some sound utilities (such as the **sdiss** command) already take into account the effects of masking, and so an input containing the `*masked` interpretation will cause an error to be generated. Similarly, the **mask** command itself will generate an error if the input sonorities have already been modified using the **mask** command.

The *sdiss* Command

A great deal of research has been carried out over the centuries concerning the nature of consonance and dissonance. This complex subject remains something of an enigma. The perception of consonance or dissonance is known to be affected by a number of factors, including past musical experience and cultural milieu. Perceptions of dissonance are even known to be influenced by the personality of the listener.

Research by Donald Greenwood, Reiner Plomp, Wim Levelt, and others has established that one aspect of dissonance perception is related to the physiology of the ear. This aspect of dissonance is referred to as low-level or *sensory dissonance*.

The **sdiss** command implements a measurement method for sensory dissonance described by Kameoka and Kuriyagawa (1969a/b). (The Humdrum **sdiss** command itself was written by Keith Mashinter.) The **sdiss** command characterizes the degree of sensory dissonance for successive vertical sonorities or acoustical moments. The command accepts as input one or more `**spect` spines and produces a single `**sdiss` spine as output. For each `**spect` data record, **sdiss** produces a single numerical value representing the aggregate sensory dissonance. The greater the output value, the greater the dissonance.

Example 36.2 illustrates some sample inputs and outputs for **sdiss**. The left-most spine provides double-stops for `**kern` data for violin. The middle spine provides corresponding `**spect` data using the **spect** command. The right-most spine shows the result of passing the `**spect` data through the **sdiss** command.

Example 36.2

```
**kern   **spect **sdiss
*Ivioln  *Ivioln *Ivioln
4c 4e    .       .
4G 4d    .       .
4f 4g    .       .
4e 4g    .       .
*-       *-      *-
```

Note that sensory dissonance is known to be influenced by the number of complex tones in the sonority. That is, three-note sonorities are virtually always more dissonant than three-note sonorities, etc. However, it is known that increasing the number of notes in a chord can sometimes reduce the perceived dissonance. For example, the dyad of a major seventh generally sounds more dissonant than the major-major-seventh (four-note) chord. Consequently, it is problematic to compare sensory dissonance values for sonorities consisting of different numbers of complex tones. Further problems with the Kameoka and Kuriyagawa measurement method are described in Mashinter (1995).

Connecting Humdrum with Csound — the *kern2cs* Command

Apart from generating and processing acoustic spectra, it is often convenient to be able to listen to the data. Generating sounds from descriptions of acoustic spectra cannot be done using MIDI synthesizers. The sounds can be heard only by doing direct computer sound synthesis using an audio-rate digital-to-analog converter. A number of popular computer sound synthesis languages exist, such as *Csound* developed by Barry Vercoe (1993). Most of these languages are inspired by the *Music 5* language developed by Max Mathews (1969).

Typically, these languages divide the task of sound synthesis into two representations called the *score* and the *orchestra*. The *orchestra* is an executable program, whereas the *score* is a set of note- or event-related data that is "performed" by the orchestra. Typically, the *score* consists of a series of note-records where each data record defines several attributes for a single note. Common attributes include the frequency (or pitch), amplitude, duration, onset time, attack/decay envelope, spectral content, etc. Example 36.3 shows a sample Csound score corresponding to the opening measures of a Mozart clarinet trio.

Example 36.3 W.A. Mozart *Clarinet Quinete.*

```
; W.A. Mozart, Second trio from Clarinet Quintet
f1 0 512 10 5 3 1 ; three harmonics in waveform table
t0 96; tempo of 96 beats per minute

; Instrument #1
; inst      time duration slur  pitch vol  stac
i1.01       0.000  0.500   1    9.00  0.2  1.0   ; measure 1
i1.01       0.500  0.500   3    9.04  0.2  1.0
i1.01       1.000  0.500   3    9.07  0.2  1.0
i1.01       1.500  0.500   3    9.04  0.2  1.0
i1.01       2.000  1.000   3   10.00  0.2  1.0
i1.01       3.000  0.500   3    9.07  0.2  1.0   ; measure 2
i1.01       3.500  0.500   3    9.04  0.2  1.0
i1.01       4.000  0.500   3    9.02  0.2  1.0
i1.01       4.500  0.500   3    9.05  0.2  1.0
i1.01       5.000  1.000   2    9.09  0.2  1.0

; Instrument #2
i2.01       2.000  1.000   0    8.09  0.2  1.0
i2.01       3.000  1.000   0    8.09  0.2  1.0   ; measure 2
i2.01       5.000  1.000   0    8.09  0.2  1.0

; Instrument #3
i3.01       2.000  1.000   0    8.04  0.2  1.0
i3.01       3.000  1.000   0    8.04  0.2  1.0   ; measure 2
i3.01       5.000  1.000   0    8.06  0.2  1.0

; Instrument #4
i4.01       2.000  1.000   0    8.01  0.2  1.0
i4.01       3.000  1.000   0    8.01  0.2  1.0   ; measure 2
i4.01       5.000  1.000   0    7.11  0.2  1.0†
```

Csound is able to generate traditional 16-bit digital audio output. It can also be used to generate AIFF files (audio information file format) for greater portability. *Csound* provides several other utilities for sound analysis, including Fourier analysis and linear predictive coding.

Sound Analysis

Humdrum does not provide any sound analysis tools *per se*. As we noted, *Csound* provides utilities for Fourier analysis and linear predictive coding. A wealth of software exists for sound analysis, loudness estimation, mixing, sound card input/output, CD audio input/output, and other sound-related applications. (See Tranter, 1996 for a sampling of such multimedia applications.) Other analysis methods are available through general-purpose signal analysis software such as *matlab*, *mathematica*, and *maple*. Custom software has also been written by Humdrum users, such as Kyle Dawkins Humdrum synchronization of CD audio disks.

Sound synthesis and analysis software has a rapid rate of development. Users should consult recent audio and multi-media resources for up-to-date information.

Reprise

In this chapter we have seen that Humdrum score-related data can be transformed into spectral information using the **spect** command. This allows us to reconstitute a score as a sequence of sonorous spectra — which might be used for studies in timbre or orchestration. The **mask** tool can be used to revise a spectral description so that it reflects how listeners hear rather than the actual acoustical information present. The **sdiss** command can be used to characterize successive spectra in terms of the estimated sensory dissonance.

We have also seen that Humdrum data can be connected to other sound-related software, such as *Csound*. Since Humdrum data consists of simple ASCII text, it is generally easy to write filters that allow the data to be imported to a wide variety of existing sound analysis and synthesis software.

Chapter 37

Electronic Editing

The preparation of encoded musical materials for processing is an essential part of computer-assisted musicology. Without electronic data to process, none of the manipulations described in this book would be possible. The encoding and editing of musical data has been an activity that has occupied various scholars since the earliest forays into computer-assisted musicology (e.g., Bronson, 1959).

In this chapter we identify and describe some of the tasks and issues involved in preparing electronic musical documents in the Humdrum format. In general, the following discussion pertains to the production of electronic documents representing musical score information. However, the basic procedures are applicable to any kind of data — from sound recordings to historical choreographies of ballets.

The Process of Electronic Editing

Electronic editing entails translating one representation into another representation. In many cases, an electronic edition is prepared from existing source documents — such as printed scores or manuscripts. In other cases, an electronic edition may be prepared from another electronic form — such as MIDI. When translating between two formats, it is important to be familiar with both representations. Descriptions of a number of popular electronic music formats may be found in Eleanor Selfridge-Field's encyclopedic *Handbook of Musical Codes* (1997).

The principal processes involved in electronic editing include identifying possible sources and alternative versions, selecting and encoding the material, proofing the material (using both automated and manual methods), adding reference and editorial information, writing appropriate research notes, generating possible analytic information, and resolving issues related to copyright, distribution, and data integrity.

Establishing the Goal

Encoding and editing music is a time-consuming and labor-intensive process. Before starting to work on a electronic edition, it is appropriate to ask: what is the goal? What purpose will be served by the edition? Is the objective to provide a performance copy? A historically accurate document? A document suitable for analysis? If so, what sort of analysis?

In general, it is wise to avoid trying to cater to all needs at once. There is nothing wrong with creating a narrowly-defined document that serves a modest goal. For example, if our intention is to pursue a study of rhythm, there is little need to encode data that is extraneous to this goal. Accordingly we might omit stem-direction, beaming, textual underlay, and even pitch information.

Note that it is always possible to incorporate additional information at a later date. Additional information can be inserted into a Humdrum encoding by using the **assemble** and **cleave** commands. In general, it is important that the encoding of electronic documents not consume all of a researcher's efforts and resources. The most common problem that beset early projects in computational musicology was that researchers rarely got past the stage of inputting data. Early researchers typically ran out of time, money or enthusiasm before they could turn to *using* the materials they had input.

Documenting Encoded Data

More important than generating a complete and accurate database is generating complete and accurate documentation so that future users of the information understand the limitations of the materials. Whether or not the electronic edition is "complete," the materials are almost useless without proper documentation about what information is present, what information has been omitted, what sources were used, and what interpretations have been made. As long as the electronic materials are well documented, users can make effective use of whatever you create.

An ethnomusicologist may have inaccurately transcribed pitches, but even approximate pitch information can be used for a study of pitch contour. Of course, it is better to have clean data than messay data, but even messy data can be highly useful if the nature of the mess is well understood by researchers. In fact, most historical information is a mess.

Concretely, Humdrum provides several reference records that allow electronic editors to identify what information has been encoded and what information has been omitted or interpreted.

Sources

In light of the editorial goal, you can procede to select source materials for encoding. In many cases, this will involve selecting printed musical scores. Begin by spending time with the materials. What special problems or challenges are raised? What sort of Humdrum representation would be suitable for the proposed edition? Are there special notational symbols that suggest a new representation should be designed? Can special representational problems be accommodated within an existing Humdrum representation? How would users be made aware of extensions or modifications to the representation?

In the case of musical scores, there is rarely a single source or *Urtext*. In most cases, multiple sources are available. How do you want to approach the problem of variant sources? One approach is to select a single source and remain true to it. A second approach is to identify the principal variant sources and encode each independently. A third approach is to encode several variants within a single document and provide Humdrum "versions" that allow users to select particular "readings" from a single file. A fourth approach is to create your own "critical" edition — based on a close re-examination of the original materials.

An important consideration will be the copyright status of the materials you examine. Some materials are old enough that their copyrights have expired. Encoding older materials raises two difficulties. First, older sources are typically not the best quality editions available. Second, access to printed versions of older sources may be difficult for other scholars using your data. Some publishers (such as Dover Publications) specialize in reprinting musical scores whose copyrights have lapsed. Although these sources may not be the best available, they are often convenient because other scholars can easily purchase the modern Dover reprints that correspond to your electronic edition.

The best materials are usually under copyright. If you encode this material without permission, you will never be able to distribute the encodings — even on a non-profit basis. Most modern publishers profess an interest in electronic publishing, but (at the time of this writing) few publishers have any pertinent expertise, and fewer yet are willing to encourage or allow the preparation of electronic documents from their publication catalogue by third parties. It is likely to take decades before the financial and legal issues are satisfactorily addressed.

In some cases, you will be translating from one electronic format to another — such as from MIDI to `**kern`. Once again, be conscious of any copyright issues that are raised. Few experiences are more discouraging than discovering that your work cannot be distributed because you failed to consider seriously the copyright issues involved.

Selecting a Sample from Some Repertory

Quite often, there are insufficient resources to encode an entire corpus. Many repertories are simply too large to consider creating an exhaustive electronic edition. Nevertheless, you might choose to encode a subset or selection of works from the given corpus.

A common expectation is that studying a sample subset of works will inform us about the corpus as a whole. That is, it is commonly expected that studying a number of works by, say, Offenbach, the scholar may be able to identify general characteristics of Offenbach's writing. In order for this assumption to be valid, statisticians tell us that the sample of works must be a *representative* sample — free of all sorts of possible selection biases.

From a research perspective, the best sample is a *random sample.* Suppose, for example, that there are 2,000 items in a particular repertory, but you are able to encode just 100 items. You might be tempted to select the first 100 items, or to select a subset of items that share some common feature that makes the collection seem "coherent."

However, picking and choosing what to encode will prevent researchers from being able to draw general conclusions about the repertory as a whole. For example, selecting the first 100 items will typically bias the sample to early works in a composer's career. Selecting 100 random items provides a very good statistical sample of a population of 2,000 items.

Unfortunately, since random sampling has little precedence in traditional humanities research, many scholars are resistent to the idea. The value of random sampling has been established beyond a doubt by statisticians. This is not the place to rehearse the detailed arguments. Simply take my word for it: if you can't encode a complete corpus, the very best solution is to select a random sample.

In making such a random sample, it is essential to resist the temptation to select a "random" sample "by eye." Establish a truly random procedure (such as flipping coins or using a random number table) and methodically follow the procedure.

Incidentally, it is common to run out of resources before completely encoding the selected materials. As a result, you may end up encoding only half or two-thirds of the projected materials. If you began encoding the materials in (say) chronological order, then the resulting database will be biased toward the early works of the repertory. In order to avoid introducing an unwanted bias, it is also prudent to encode the selected materials in a random order.

Encoding

Once you have established your materials and have decided on the type of encoding, you can go ahead and begin encoding the documents in random order. Use whatever resources are available to you. These might include scanning software, MIDI performance capture, or the Humdrum **encode** command. Begin by encoding a sample section or sections. Spend some time determining ways to increase your productivity.

As you encode the selected materials, editorial problems or questions will inevitably arise. As you gain experience, you may realize that earlier encoding practices were not the best. You may want to return to these problems and encode them in a different manner. Be sure to keep notes — either pencil marks on a page, or local comments in a file — so that you can easily revisit these problem sites later. Again, it is valuable to encode works in random order in order to avoid possible confounds arising from editorial experience. That is, you don't want a scholar's conclusions about differences between early works and late works to be merely an artifact of the electronic editor's increasing experience.

Typically, it is more efficient to encode individual parts and then assemble all parts into a single full score.

Transposing Instruments

In the case of the `**kern` representation, all parts are represented at concert pitch. It is typically easier to encode the parts as written and then transpose the result using the Humdrum **trans** command. For example, material for B-flat trumpet or B-flat clarinet can be transposed using the following command:

```
trans -d -1 -c -2
```

In the case of clarinet in A, a suitable transposition would be:

```
trans -d -2 -c -3
```

The **trans** command adds a transposition interpretation to the output in order to identify that the material has been shifted. In the `**kern` representation, transposed instruments must be explicitly identified using a special "transposing-instrument interpretation" (see *Humdrum Reference Manual* — Section 3 for details). A suitable interpretation can be created by adding the upper-

case letter 'I' prior to the 'T' in the appropriate tandem interpretation. In the case of a horn in F for example, the transpostion interpretation would be modified from

```
*Trd-4c-7
```

to:

```
*ITrd-4c-7
```

Instrument Identification

Humdrum provides standardized instrumentation indicators. Three different types of indication are appropriate: (1) the instrument name as indicated in the source, (2) standardized instrument name, and (3) instrument class. Standardized instrument names can be found in Appendix II. For example, the standard indicator for "harpsichord" is `*Icemba`.

Standardized instrument class designators include `*ICklav` for keyboard instruments and `*ICidio` for percussion instruments, etc., and instrument grouping designators — such as `*IGripn` for *ripieno* instruments and `*IGacmp` for accompaniment instruments. These instrument class designators can also be found in Appendix II.

In addition, the original instrument name (as found in the score) should also be encoded as a Humdrum local comment.

Leading Barlines

Humdrum tools prefer to have explicit information indicating the beginning of the first measure. If a file does not begin with an anacrusis ("pickup") then it is appropriate to encode an "invisible" first barline. For a hypothetical file containing five spines, we would need to insert the following line just before the first note(s) in the work:

```
=1-   =1-   =1-   =1-   =1-
```

Recall that the common system for representing barlines makes a distinction between the logical *function* of a barline and it's visual or *orthographic* appearance. For example, the common system for barlines distinguishes between double barlines whose function is to indicate the end of a work or movement, and double barlines that simply delineate sections within the course of a work or movement. It is possible for a barline at the end of the work to be "functionally" a double barline, yet appear visually as a single barline.

Functional double barlines are encoded with a double equals sign (==) whether or not they are visually rendered as double barlines. *Functional single barlines* are encoded with a single equals sign (=) whether or not they are visually rendered as single barlines.

The specific visual appearance may be encoded following the equals sign(s). The vertical line (|) represents a 'thin' line and the exclamation mark (!) represents a 'thick' line. A typical final double bar would be encoded:

```
==|!
```

Most mid-movement double bars are encoded with two thin lines and so would be encoded:

```
=||
```

A common encoding error is to render mid-movement double barlines as *functional* rather than *orthographic* double-bars.

Ornamentation

The `**kern` representation makes a distinction between whole-tone and semitone trills and mordents. Typically, each ornament must be examined manually and the correct code selected.

In some cases, the size of the trill or mordent will be ambiguous and so some sort of editorial decision will be necessary. One possibility is to add the kern 'x' signifier immediately following the 'T' or 't'. This indicates that the trill size is an "editorial interpretation."

The `**kern` representation treats appoggiaturas in a special way. In general, `**kern` is oriented to representing things in a manner closer to how they sound. Consequently, appoggiaturas are encoded as they would be logically performed. For example, a quarter-note preceded by an appoggiatura (small note) would be performed as two eighth-notes. Similarly, a dotted quarter-note preceded by an appoggiatura would be performed as a quarter-note followed by an eighth-note.

All appoggiaturas must be re-encoded in a way that reflects their likely performance. At the same time, the two notes forming the appoggiatura must be marked in the kern representation: the initial note of the appoggiatura is marked by the upper-case letter '`P`' and the final (second) note of the appoggiatura is marked by a lower-case letter '`p`'.

Editing Sections

It is helpful to break-up large works/movements into smaller sections that can be labelled. In a binary work, for example, it may be useful to label the 'A' and 'B' sections. In a sonata-allegro work, it may be useful to label the introduction, exposition, development, recapitulation, etc. Some works include explicitly notated labels. These labels may be traditional, e.g. "Coda," or they may reflect programatic descriptions, such as the section entitled *Il canto degl'uccelli* [The song of the birds] in Vivaldi's *The Four Seasons.*

Where appropriate, suitable section labels should be created and encoded using the Humdrum Section Label designator. Remember that section labels can include the space character:

```
*>1st Theme
```

If you include section labels, you must also include a Humdrum "Expansion List" to indicate how the sections are connected. The Humdrum **thru** command causes a through-composed version of a file to be generated according to the expansion list. For example, an expansion list for a simple binary work may be encoded as:

```
*>[A,B]
```

Remember that expansion lists ought to be encoded prior to the first section label.

Whenever a work/movement includes repeats or Da Capos, section labels and expansion lists must be encoded. In some cases, there is more than one way of interpreting how to realize the repeats. The most "conventional" realization should be encoded with the *unnamed expansion list.* This will specify the default expansion using the Humdrum **thru** command. Suppose for example, that you are encoding a typical minuet and trio. The conventional performance practice involves repeating all sections of both the minuet and trio, but then avoiding the repeats in the minuet following the Da Capo. A suitable expansion list might be:

```
*>[Minuet,Minuet,Trio,Trio,,Minuet]
```

An alternative expansion list might be encoded as follows (notice the expansion-list-label *ossia*):

```
*>ossia[Minuet,Minuet,Trio,Trio,,Minuet,Minuet]
```

Editorialisms in the ***kern* Representation

Humdrum provides several ways of encoding editorialisms. These include editorial footnotes, local comments, global comments, interpretation data, *sic* and *ossia* designations, version labels, sectional labels, and expansion lists.

The `**kern` representation provides several special-purpose signifiers to help make explicit various classes of editorial amendments, interpretations, or commentaries. Five types of editorial signifiers are available: (1) *sic* (information is encoded literally, but is questionable) signified by the `Y` character; (2) *invisible symbol* (Unprinted note, rest or barline, but logically implied) signified by the `y` character; (3) *editorial interpretation*, (a "modest" editorial act of interpretation — such as the interpretation of accidentals in *musica ficta*) signified by the `x` character; (4) *editorial intervention* (a "significant" editorial intervention) signified by the `X` character; (5) *footnote* (accompanying local or global comment provides a text commentary pertaining to specified data token) signified by `?`.

One of the most onerous impositions of the `**kern` representation is the requirement that the music be interpreted into a coherent spine organization. Why not avoid interpreting the voicings?

The answer to this question is that editorial interventions are often essential clarifications that make a document useable. Without voicing information, users would be unable to calculate melodic intervals, for example. Without melodic intervals, it may be impossible to search for themes, motives, and other patterns. Editorial interpretations are not simply unwarranted obfuscations. This does not mean that interpretations are "correct" and so it may be necessary to provide several alternative or plausible interpretations of an artifact.

One of the advantages of computers is that it is possible for documents to undergo continuous revision. In research, it is common for documents to be reinterpreted, annotated, or recast in light of newly found documents.

The kern 'x' signifies an "editorial interpretation" — that the immediately preceding signifier is interpreted. The kern 'xx' also signifies an editorial interpretation where the immediately preceding data token is interpreted. The kern 'X' signifies an "editorial intervention" — that the immediately preceding signifier is an editorial addition. The kern 'XX' also signifies an editorial intervention where the immediately preceding data token is an editorial addition. The kern 'y' designates a invisible symbol — such as an unprinted note or rest that is logically implied. The kern 'Y' signifies an editorial *sic* marking — that the information is encoded literally, but is questionable. The kern '?' signifies an editorial footnote where the immediately preceding signifier has an accompanying editorial footnote (located in a comment record). The kern '??' signifies an editorial footnote where the immediately preceding data token has an accompanying editorial footnote (located in a comment record).

Adding Reference Information

Reference information must be added to each file. This information provides "library-type" information about the composer, date of composition, place of composition, copyright notice, etc.

As many reference records should be added as possible since these are immensely useful to Humdrum users. Essential reference records include the following:

!!!COM:	composer's name
!!!CDT:	composer's dates
!!!OTL:	title (in original language)
!!!OMV:	movement number (if appropriate)
!!!OPS:	opus number (if appropriate)
!!!ODT:	date of composition
!!!OPC:	place of composition
!!!YEP:	publisher of electronic edition
!!!YEC:	date & owner of electronic copyright
!!!YER:	date electronic edition released
!!!YEM:	copyright message
!!!YEN:	country of copyright
!!!EED:	electronic editor
!!!ENC:	encoder of document
!!!EEV:	electronic edition version
!!!ELF:	file number, e.g. 1 or 4 (1/4)
!!!VTS:	checksum validation number (see below)
!!!AMT:	metric classification
!!!AIN:	instrumentation

Where appropriate, the following reference records should also be included:

!!!CNT:	composer's nationality
!!!XEN:	title (English translation)
!!!OPR:	title of larger (or parent) work
!!!ODE:	dedication
!!!OCY:	country of composition

`!!!PPR:`	first publisher
`!!!PDT:`	date first published
`!!!PPP:`	place first published
`!!!SCT:`	scholarly catalogue name & number
`!!!SMA:`	manuscript acknowledgement
`!!!AFR:`	form of work
`!!!AGN:`	genre of work
`!!!AST:`	style of period

In general, place essential reference records at the beginning of a document. These will include the composer, title of the work, etc. Less important reference records should be placed at the end of the file. Minimizing the number of reference records at the beginning of a file makes it more convenient for users looking at the contents of a Humdrum file.

Refer to the *Humdrum Reference Manual* for further information about the types and format for different reference records.

Proof-reading Materials

Once you have encoded your document, you should create a error-checking strategy. The Humdrum **humdrum** command can be used to identify whether the final encoded output conforms to the Humdrum syntax:

```
humdrum full.krn
```

Use the Humdrum **proof -k** command to identify any syntactical errors in any encoded `**kern` data:

```
proof -k full.krn
```

One of the best ways to ensure that musical data makes sense is to listen to it. The Humdrum **midi** and **perform** commands can be used to listen to your data.

```
midi -c full.krn | perform
```

The **perform** command allows you to *pause* (press the space bar), to *move* to a particular measure (type a measure number followed by enter), to increase (type <) or decrease (type >) the *tempo,* and to *return* to the beginning of the score (type enter). There are many other functions within the **perform** command; refer to the *Humdrum Reference Manual* — section 4 for further details.

Data Integrity Using the VTS Checksum Record.

When using electronic documents, it is often useful to modify the document for some purpose. After a while, the user will become confused about the status of a document. Is this the original distribution file? Did I make some modification to this file that I've forgotten about? Has someone tampered with this data?

Humdrum provides a means for ensuring that a particular file is what it purports to be. The **veritas** command provides a formal means for verifying that a given Humdrum file is identical to the original distribution file and has not been modified in some way.

The **veritas** command works by looking for a VTS reference record in the file. It then calculates a "checksum" for the file (excluding the VTS record itself) and compares this value with the encoded VTS value. If these values differ, a warning is issued that the file has been modified in some way.

Once you are certain that an encoded Humdrum file is completely finished, you should calculate a "checksum" value to be encoded in a Humdrum "VTS" reference record.

In order to calculate the checksum value for a given file, use the following command:

```
cksum final.file > temp
```

Open the original file and move to the bottom of the document. Then read in the calculated checksum value. Finally, insert the '`!!!VTS: `' reference record designator.

You can check that everything is fine by invoking the **veritas** command:

```
veritas final.file
```

The command will complain only if the VTS checksum value does not correspond to the computed checksum for the file. Finally, be sure to include the checksum value in an index or README file for the distribution. This provides a public venue for users to determine whether the VTS record itself has not been modified.

Preparing a Distribution

Finally, you may want to prepare the material you have encoded for public distribution. Rename the score files and collect them into a coherent repertory. If your data is encoded in the `**kern` format, be sure to use the `.krn` file extension. Place all resulting Humdrum files in a single directory.

Create a `README` file similar to others in Humdrum data distributions. The file should contain a repertory title, a brief paragraph describing the historical background for the works, a paragraph describing the personnel involved in the production, a copyright and license notice, and a table of contents. Avoid tabs in this file, and ensure that no line is greater than 80-characters in length.

It is wise to also add a `LICENSE` file that reiterates whatever licensing agreement is entailed for the distributed data.

Electronic Citation

Electronic editions of music might be cited in printed or other documents by including the

following information. The "author" (e.g. **!!!COM:**), the "title" — either original title (**!!!OTL:**) or translated title (**!!!XEN:**). The editor (**!!!EED:**), published (**!!!YEP:**), date of publication and copyright owner (**!!!YED:**), and electronic version (**EEV:**). In addition, a full citation ought to include the validation checksum (**!!!VTS:**). The validation number will allow others to verify that a particular electronic document is precisely the one cited. A sample citation to an electronic document might be:

> Franz Liszt, Hungarian Rhapsody No. 8 in F-sharp minor (solo piano). Amsterdam: Rijkaard Software Publishers, 1994; H. Vorisek (Ed.), Electronic edition version 2.1, checksum 891678772.

Reprise

In this chapter we have reviewed the principal issues involved in preparing electronic music documents in Humdrum.

Chapter 38

Systematic Musicology

Much of music research centers on the task of describing things. A researcher might offer a description of "House" style, or Wagner's orchestration, commonalities in themes by Respighi, or a Balinese variation technique. Good descriptions are based on the identification of *features*. A "feature" is a notable or characteristic part of something — something that helps to distinguish one thing from another thing (or one group of things from another group of things).

Not all truthful descriptions qualify as distinctive features. Imagine that you were robbed by someone and were later asked by the police to provide a description of your assailant. Suppose you began your description by saying that the robber had a nose, a mouth, and two eyes. These facts would undoubtedly be true, but the police would be rightly dismayed by your description for the simple reason that the facts fail to distinguish your assilant from billions of potential suspects.

When characterizing a musical work, genre, style, or composer, it is important, not simply to make truthful observations. It is also important to identify those characteristics that distinguish the work, genre, style, or composer from others. If the goal of an analytic description is to convey what is unique or characteristic of a given object or class of objects, then good features must embody or define some of that distinctiveness. That is, the research must guard against describing something that is commonplace.

In this chapter, we will describe several methods to help researchers determine whether a presumed feature is distinctive. In essence, we will ask the following question: "How likely is it that this is a feature of some larger class of objects — like the class of all musical works?" If the feature is commonly found in many situations, then we are not justified in claiming that the feature is unique to a particular situation.

In order to determine whether a proposed feature is distinctive of a work, we need to compare the incidence of occurrence with a body of works where we wouldn't expect the feature to be so common. In systematic musicology, we say we are looking for a comparison or *control* sample. There are four basic approaches to establishing a control sample:

- comparison repertory
- randomizing

- counter-balancing
- autophase procedure

Comparison Repertory

By a "comparison repertory" we mean a group of works, that we hypothesize do not show the same distinctive feature — although in other respects the works are similar.

Suppose a scholar has observed a particular feature that appears to occur frequently in the music of Respighi. Is this pattern a characteristic feature of Respighi's music? We cannot know this by looking only at the music of Respighi. We must choose a comparison repertory and show that the same feature is not common in the comparison group.

Ideally, the researcher should choose a comparison repertory that is as similar as possible to the target repertory. For example, if we found that Respighi's music contrasted with Rameau's music, we would not be able to dismiss the possibility that the observed differences arise because of differences in nationality, or differences in stylistic period, or differences in instrumentation, etc. Choosing a similar composer would be a better test of the proposed feature.

Of course it is almost never possible to select a perfectly matched comparison group.

When selecting a comparison repertory, the researcher should make a list of possible confounds. For example, the feature might be

In more careful studies, the researcher might attempt to match the repertories for a series of attributes. For example, one might match the number of works in both repertories that are in triple meters. Similarly, one might match the modality so both the target and comparison repertories have the same proportion of major/minor keys. One might similarly match instrumentation, date or period of composition, duration of work, tempi, and so on.

Humdrum users can use the **find** and **grep** commands to identify works that match particular controlled characteristics (see Chapter 33). For example, the following command might locate all files that contain scores in triple meter. The results are placed in a file called `control`:

```
find /scores -type f -exec grep -l '!!!AMT.*triple' "{}" ";" \
   > control
```

In some cases, the number of possible control works is excessively large. In this case one can make a random selection from the control list using the **scramble** command described later.

Note that in many instances it is difficult to establish a good comparison repertory. For example, suppose we find significant differences between Beethoven melodies and Kanartic folk melodies. How should we interpret these differences? The differences might be symptomatic of the difference between folk music and classical music. Or the differences might reflect differences between German and Indian music. Or the differences might reflect differences between early 19th century music and mid-twenthieth century music. Or the differences might be attributable to differences of instrumentation.

No matter what comparison repertory we choose, someone might be able to claim that any observed differences arise due to some other factor. For example, we might compare early and late works by Bach as a way of tracing his musical development. However, someone might claim that any differences found are not related to Bach's development as a composer, but are due to different tastes in Weimar versus Leipzig. Bach was simply showing his ability to adapt to local tastes.

Randomizing

Sometimes scholars formulate hypotheses that are intended to pertain to the whole of music — that is, the feature is thought to be a musical universal. In these cases it may be impossible to identify a comparison repertory.

Many theorists have noticed, for example, that melodies tend to be composed predominantly of small intervals. That is, there appears to be a preference for small melodic intervals in most forms of music. Unfortunately, the predominance of small intervals might simply be an artifact of a small range. Consider the case of a melody that is restricted to the range of an octave. Between any two pitches, there is only a single instance of a 12-semitone interval, however, there are 12 possible 1-semitone intervals. Even in a random ordering of notes, one would expect to see a preponderance of small intervals.

So is there any way of testing whether composers really do favor small melodic intervals? Or is this simply an artifact of range restrictions? Many such hypotheses can be tested using a randomizing procedure. We can illustrate this procedure by working through the problem of small melodic intervals.

We begin by measuring the average melodic interval size in semitones for a sample of actual melodies. We can use the **semits** command to translate data to semitone representations and then use the **xdelta** command to calculate numerical differences. The **-a** option for **xdelta** causes only absolute (unsigned) values to be calculated. The **rid** command can be used to eliminate everything but data records and the **grep** command can be used to eliminate barlines and rests. We can then calculate the average interval size by piping the output to the **stats** command. For typical folk melodies, the average interval size is roughly two semitones.

```
semits melody | xdelta -a | rid -GLId | grep -v '[=r]' | stats
```

Next, we need to determine the average melodic interval size that would result for a random re-ordering of the pitches within each melody. We can do this using the Humdrum **scramble** command.

Using the *scramble* Command

The **scramble** command is useful for randomizing the arrangement of Humdrum data. Suppose we had the following Humdrum input:

```
**numbers
1
2
3
4
5
*-
```

We can scramble the order of data records using the following command:

```
scramble -r numbers
```

The **-r** option indicates that it is the order of records which should be randomized. A possible output might look like this:

```
**numbers
3
2
5
1
4
*-
```

Notice that only data records are scrambled: comments and interpretations stay put. Each time **scramble** is invoked, it produces a different random ordering.

Returning to our melodic interval problem, we can now generate an inventory of melodic intervals for our original repertory, where the order of the notes has been randomly ordered:

```
scramble -r melody | semits | xdelta -a | rid -GLId \
   | grep -v '[=r]' | stats
```

For a typical folksong repertory, the average melodic interval size for a randomly re-ordered melody is roughly 3 semitones in size. Using common statistical tests, it is possible to prove that this difference is unlikely to occur by chance and that it likely is a symptom of real efforts to organize melodies using relatively small melodic intervals.

A similar approach can be used to address innumerable questions. For example, in Haydn's music, it seems that Haydn tends to avoid following the dominant by a subdominant chord (i.e., *V-IV*). On the other hand, Haydn's use of the *IV* chord is comparatively infrequent, so the apparent absence of this progression may simply be an artifact of the relative scarcity of subdominant chords. We can address this question by comparing Haydn's actual harmonic progressions with randomly generated progressions. First we count the total number of *V-IV* progressions:

```
extract -i '**harm' haydn  | context -n 2 -o ^= \
   | grep -c '^V IV$'
```

Next we randomly re-order his harmonies and count the number of *V-IV* progressions:

```
scramble -r haydn | extract -i '**harm' | context -n 2 -o ^= \
   | grep -c '^V IV$'
```

In some cases, problems can be addressed by randomizing one part of voice with respect to another. For example, there is strong evidence that Bach uses more augmented eleventh harmonic intervals than would occur by chance. That is, the tritone is "sought-out" rather than "avoided" in his writing. Suppose we are looking at a two-part invention. We begin by counting the number of augmented elevenths in his actual writing:

```
ditto -s = bach | hint | grep -c 'A11'
```

We can create a random comparison by extracting one of the parts, scrambling the order of notes, and then re-assembling the scrambled part with the original. The resulting harmonic intervals arise from a random juxtaposition of parts.

```
extract -f 1 bach > temp1
extract -f 2 bach > temp2
scramble -r temp1 > temp1.scr
assemble temp1.scr temp2 | ditto -s = | hint | grep -c 'A11'
```

Note that the **scramble** command also provides a **-t** option so that the order of tokens within a data record can be randomly re-arranged.

Retrograde Controls Using the *tac* Command

Suppose a theorist found an unusually large number of occurrences of the B-A-C-H pitch pattern in some repertory. Are these patterns intentional on the part of the composer? Or should we expect a fair number of such patterns to occur simply by chance?

One way of determining the chance frequency of B-A-C-H might be to randomize the order of pitches using the **scramble** command, and then use **patt** to count the number of occurrences in the reordered melodies. Unfortunately, we already know that musical lines tend to be constructed using small intervals, and the pitches B-A-C-H are very close together. Since random reordering of the pitches will reduce the proportion of small intervals, we would naturally expect fewer instances of B-A-C-H in the random musical lines. We need some way to maintain the identical interval distribution in our control repertory.

One way to do this is by *reversing* the sequencial order of the notes. If we could rearrange the notes so that the first note was last and the last note was first, then our control repertory would preserve both the frequency of occurrence of all the pitches, and also preserve the pitch-proximity distribution.

The UNIX **tac** command can be used to reverse the order of records. Suppose we had an input consisting of the number 1 through 10 on successive lines. The **tac** command would transform this input so that the output consists of the reverse ordering of numbers from 10 to 1.

If we apply **tac** to a Humdrum file, then the result will no longer conform to the Humdrum syntax — the spine-path terminators will appear at the beginning of the file and the exclusive interpretations will appear at the end of the file. If we use **tac** we could simply restore the correct syntax by hand-editing the file and moving the exclusive interpretations and the spine-path terminators to their proper locations. We now have a "retrograde" passage.

Such a retrograde passage will provide a useful control repertory to test our B-A-C-H hypothesis. If a composer is intentionally composing several instances of B-A-C-H into his/her music, then we would expect the number of occurrences to be somewhat more frequent than instances of B-A-C-H found in retrograde versions of the works.

Another way of testing the same hypothesis would be to search for the reverse pitch sequence: H-C-A-B.

Autophase Procedure

Frequently researchers are interested in the relationship between concurrent musical parts or voices. Suppose, for example, that we had reason to suspect that a particular polyphonic composer tends to actively avoid octave intervals between the bass and soprano voices. If we find that the proportion of octave intervals is 6 percent, how do we know whether this is a lot or a little?

One approach to answering this question is to use an *autophase procedure* (Huron, 1991a). The essence of this approach is to shift two spines with respect to each other.

Recall that the **reihe** command (Chapter 35) provides a **-s** option that causes a shift in the serial position of data tokens. For example, suppose we had an input consisting of the numbers 1 through 5. The following command:

```
reihe -s +1 file
```

Will cause all data tokens to be moved forward one position, and the last data token to be moved to the beginning:

```
**numbers
5
1
2
3
4
*-
```

Let's apply this technique to our problem of whether a given composer tends to avoid octaves between the soprano and bass voices. First, we extract each of the voices. Let's also eliminate barlines and use **ditto** to replicate the pitch values through null tokens.

```
extract -i '*sopran' composition | grep -v = | ditto > voice1
extract -i '*bass' composition | grep -v = | ditto > voice2
```

Now let's shift one part with respect to the other using **reihe -s**.

```
reihe -s voice1 > voice1.shifted
```

Now we reassemble the parts, determine the harmonic intervals present, and count the number of octave intervals:

```
assemble voice2 voice1.shifted | hint | grep -c 'P8'
```

In effect, we have concocted a control group, by shifting the parts with respect to each other. Of course we have utterly destroyed the *relationship between the two parts.* However, many things remain untouched. The bass voice remains identical, and the soprano voice is identical except that there is an extra melodic interval (between the first and last notes) and one melodic interval missing. In short, we have preserved the within-voice organization while destroying the between-voice organization.

Rather than using a single shifted control, it is typically better to repeat the procedure, methodically shifting the spines through a complete 360 degree rotation. We can then compare measures for the actual work with a distribution of measures for all of the shifted values.

The following script calculates the number of P8 intervals for each of the possible shifts between the first and second spines in the file `composition`. The number of data records is determined and assigned to the `LENGTH` variable. A `while` loop is used to calculate the number of octave intervals for each of the possible shifts between the parts:

```
extract -f 1 composition | grep -v = | ditto > spine1
extract -f 2 composition | grep -v = | ditto > spine2
LENGTH=`rid -GLId spine1 | wc -l | sed 's/ //g'`
X=1
while [ $X -ne $LENGTH ]
do
    reihe -s $X spine1 > temp
    assemble spine2 temp | hint | grep -c 'P8'
done
rm spine[12] temp
```

This *autophase* procedure has been used to address many differ kinds of questions pertaining to how musical parts interrelate.

Reprise

When using computers to measure or observe something it is important not to jump to conclusions from what we find. Just because something is either prevalent or rare does not mean that it is significant: it might be prevalent or rare simply by chance (e.g., von Hippel & Huron, MS). In this chapter we have illustrated a number of methods for testing hypotheses by contrasting a target repertory with various controlled data.

We have discussed four different control methods: comparison repertory, randomizing, retrograde, and the autophase procedure. In addition, one might use musical inversion as a control method. All of these methods allow us to compare how music is actually organized with how it might be organized. In certain cases, such contrasts allow us to infer aspects of musical organization that would otherwise be difficult or impossible to decipher.

Chapter 39

Trouble-Shooting

Computers have an unbounded capacity to generate nonsense. Even when commands appear to execute correctly, there is no guarantee that the results are accurate or meaningful. In this chapter we will identify some of the many things that can go wrong when using the Humdrum Toolkit. We will also present a number of suggestions and tips that will help you avoid potential problems.

Errors can arise from a number of sources including corrupt or error-prone input data, failure to anticipate special circumstances or exceptions, improper processing, software bugs, and incautious interpretations of results.

Encoding Errors

In the first instance, the accuracy of your results will depend on the accuracy of the input data. Humdrum data may originate from a variety of sources. Users may encode their own materials, or use existing data encoded by other individuals or available from institutional sources. Data quality can be highly variable and there may be no easy way to determine the accuracy of a given data set.

It is important to spend time with a data set. Historical musicologists may spend a considerable amount of time becoming familiar with a manuscript, and the same practice is recommended for computational musicologists. Users should look at the data, listen to the data, compare the data to published sources, and generally browse and peruse it. Most data errors are discovered while processing the data — such as finding a suspicious major ninth melodic interval in a simple song. Over time, more and more errors are eventually discovered and corrected. Unfortunately, there is no magic flag that pops up to notify us when all errors have been eliminated from an encoded musical work. Only over time will the user gain confidence (or lose confidence) in a given data set. In working with a file, we are far more apt to discover something is wrong with the data than to learn that the data is a pristine encoding.

Errors can be magnified by the type of processing that is applied. For example, consider the case of an encoded repertory that is known to have a pitch-related error rate of 1 percent. That is, roughly 1 out of every 100 notes has an incorrect pitch representation. If we were to do an inventory of pitches in this repertory, then our results would also exhibit a 1 percent error rate. For many applications, such errors are not a problem.

However, consider what happens when we create an inventory of melodic intervals. One incorrect

pitch will falsify *two* melodic intervals, hence the error rates for intervals is now 2 percent. Similarly, if we are looking at four-note chords, a single wrong pitch will falsify an entire chord. So we will have roughly a 4 percent error rate for chord identification. If we are investigating simple chord progressions, a single wrong note will now disrupt the identification of two successive chords. Thus we have an error rate of 8 percent for chord progressions.

There are two general lessons that can be drawn from these observations. The first lesson is obvious. Always try to use the best quality data that is available. When encoding your own data, aim for total accuracy. The second lesson is more subtle. The more data that participates in identifying some pattern, the greater the likelihood that a single data error will cause a problem. Whenever possible try to restrict pattern searches to small or concise patterns.

Searching Tips

Many of the problems in computer-based musicology are evident when searching for some pattern. In general, there are two types of searching errors: *false hits* and *misses*. A *false hit* occurs when the search returns something that is not intended. A *miss* occurs when the search fails to catch an instance that was intended to be a match. Unfortunately, efforts to reduce the number of false hits often tend to increase the number of misses. Similarly, efforts to avoid misses often tend to increase the number of false hits. Precision and caution are necessary.

Search failures can arise from five sources: (1) corrupt or inaccurate data, (2) failure to search all of the intended data, (3) inaccurate or inappropriate definition of the search template, (4) failure to understand how a given search tool or option operates, (5) failure on the part of the user to form a clear idea of what is being sought. Let's deal with each of these problems in turn.

(1) No search can produce accurate results if the data to be searched is inaccurate. You can increase the accuracy of your search by choosing high quality data and preparing the files in an appropriate manner.

Tips:

- Use the **humdrum** command to ensure that the input data conforms to the Humdrum syntax.
- Use the **humdrum -v** command to determine whether the kind of data (signifiers) you are interested in are truly present in all of the files to be searched.
- Use the **proof** command to ensure that any `**kern` data is properly encoded.
- Visually inspect sample passages from the input data. Do not rely solely on the **ms** command; instead, look at the actual ASCII text data using the **more** command.
- Use the **midi** and **perform** commands to listen to sample passages; ensure that the data makes sense.
- If you are uncertain of the quality of the data, try encoding a randomly selected subset and then use the UNIX **diff** command to identify any differences between the original data and the re-encoded data.

- Always read any release notes or README files that accompany the data.
- Use **grep** to search for *warning* and *Nota Bene* reference records (`!!!RWG:` and `!!!ONB:`). These records may contain important editorial notes or warnings.
- Where appropriate, expand files to through-composed versions (using **thru**) before searching. If more than one editorial version is present in a document, select the most appropriate edition before processing.
- Create an inventory of all the types of data tokens present in an input. Inspect the inventory list to determine whether any unexpected data are present.
- If necessary, eliminate certain types of data that might confound or interfere with your search in some way. Use **rid**, **grep -v**, **extract -i**, **sed** and/or **humsed** to restrict the data.

(2) Ensure that you are searching all of the intended data:

- Use **grep** to search for titles, or composers, or opus numbers, etc., in order to ensure that the file or files you are searching are the ones you want. Also check to ensure that you are not searching materials that don't belong in the input.
- Be wary of searching duplicated materials. Create inventories of titles, opus numbers, etc., and use **uniq -d** to identify unwanted duplicate copies of works or files.
- Use the **ls -l** or **wc** commands to determine the size of the search data. Does the amount of input data seem unduly small or unduly large?
- Use the **find** command to search the system for other files that ought to be included in the search task.

(3) One of the most common problems in searching arises from inaccurate or inappropriate search templates.

Tips:

- Be careful when formulating regular expressions. Read aloud the meaning of the regular expression.
- Do not use *extended* regular expressions with the **grep** command. Use **egrep** instead.
- Ensure that you know which characters in your search template are meta-characters.
- Execute your command from a shell script file so that you don't inadvertently make a typing error when entering the command.
- Maintain a command history file so that you have a permanent record of what you did. Depending on the system settings, the UNIX **history** command will display the past 100 (or more) commands you have executed. Place this information in a permanent `record` file as follows:

```
history > record
```

In addition, keep records of the precise regular expressions used for a given project. These records will help you determine later whether you made a mistake. For added security, print-out these files and glue them into a lab book.

- Create a test file containing different patterns, and test the ability of your regular expressions to catch all cases. Included "lures" in your test — i.e., patterns that are close to what you want, but should be rejected.
- Use extra caution when using "not" logic. For example, the **grep** expression "not-A" (i.e. `[^A]`) will still match records containing the letter `A` as long as one non-A letter is present. The commands

```
      grep [^A]
and   grep -v A
```

  are *not* the same.
- Compare outputs from a search that you know ought to increase the number of false hits. Compare outputs from a search that you know ought to miss some sought patterns.
- Translate the data to another representation and repeat the search using a different pattern tailored to the new representation. The results should be identical.
- Maintain a file containing regular expressions you have tested so you can re-use them in later projects.
- Visually inspect the ASCII output to ensure that the results are correct. Remember that visual inspection will only help you identify *false hits.* Visual inspection of the output will not help you identify *misses.*
- Use the **midi** and **perform** commands to proof-listen to your output. Again remember that aural inspection will only help you identify *false hits.*
- Ask whether the output makes sense. Given the amount of music searched, does it make sense to find the number of occurrences found?
- Try making a slight modification to your pattern template — a modification that you know should produce a different result.
- Look for converging evidence. Try two or three contrasting approaches to ensure that the same answer arises for each approach. For example, try searching each part individually using the **extract** command.

(4) Ensure that you understand how a given search tool or option operates.

Tips:

- Remember that *extended* regular expressions require the use of **egrep** rather than **grep**.
- Re-read the documentation to ensure that each software tool does what you think it does.
- Refer to the examples given in the *Humdrum Reference Manual* in order to ensure that you understand what a given option does.
- Compare outputs using different options. Ensure that your selected option(s) is matching the correct pattern.
- Use the **humver** command to determine which version of the Humdrum Toolkit you are using. Ensure that the documentation pertains to the correct version.

- Read the "Release Notes" for the software you use. Known software bugs are often reported in such notes or in the documentation.
- Report discovered bugs to the software's author. Even if the software is not revised, other users should be informed of the problem.

(5) Perhaps the most onerous problems in pattern searching arises when the user fails to have a clear understanding of what is being sought:

Tips:

- Think carefully about the search problem. What precisely are you looking for?
- Inspect the input to familiarize yourself with various contexts and possible variants.
- Check your search by carrying out a manual search of a random subset of the data.

Compared with manual research, computer searches are impressively fast. However, don't let yourself be caught-up by the speed of interaction. Take your time and reflect on the problem being addressed. Formulate a search strategy away from the computer so that you have time to consider possible confounds.

Pipeline Tips

Apart from searching tasks, most Humdrum processing involves two or more software tools linked in a pipeline. Pipelines can obscure all sorts of processing errors.

Tips:

- Slowly assemble your pipeline by adding one software tool at a time. Visually inspect the output following the addition of each process.
- Start with a small volume of input data. Once you have some confidence in your pipeline use a *different* sample of input data. Again add one software tool at a time while inspecting the results at each stage.
- Use the UNIX **tee** command to generate files at intermediate points in the processing. Use the **assemble** command to align inputs and outputs at various stages in the processing.
- Execute your finalized pipeline from a shell script in order to avoid undetected typing errors.

Reprise

In research-oriented activities, it is essential to exercise care when relying on computer-based methods. Computers have an unbounded capacity to generate false results. Unfortunately, computer outputs often seem deceptively authoritative. Take your time and develop a coherent strategy for solving a particular problem. Test your materials and processes, and maintain good records of what you have done. For critical tasks, always use two or more independent methods to ensure that the results agree. In general, cultivate a skeptical attitude; wise users are wary users.

Chapter 40

Conclusion

In Chapter 1, we noted that computers are quite limited in what they are able to do. We noted in particular that computers are not good at *interpreting* data. Because so much music scholarship hinges on interpretations, this would seem to preclude computers from being of much use. However, as we have seen, there are some things that computers do well, and when a skilled user intervenes to make a few crucial interpretations, the resulting computer/human interaction can lead to pretty sophisticated results.

The lucid use of Humdrum depends on understanding how each of the tools works and how it is possible to connect the tools to perform particular tasks. The power and creativity of Humdrum truly lies in the hands of the user.

As we've seen, there are two parts to Humdrum: the *Humdrum syntax* and the *Humdrum toolkit.* The Humdrum syntax simply provides a formal framework for representing any kind of sequential symbolic data. A number of representation schemes are pre-defined in Humdrum, but you are free to construct your own representations as demanded by the tasks. In several of the tutorial examples in the previous chapters we saw various *ad hoc* representations that were concocted as temporary or intermediate representations. You should now feel comfortable with the possibilities of devising your own representations to assist you in whatever task you are interested.

The *Humdrum Toolkit* is a set of inter-related software tools. These tools manipulate text data conforming to the Humdrum syntax. If the data represents music-related information, then we can say that the Humdrum tools manipulate music-related information. Each Humdrum tool carries out a fairly modest process.

By way of review, we can group the various Humdrum (and UNIX) tools according to the type of operation. There are roughly a dozen or so classes of tasks:

1. **Displaying things.** The **ms** command can be used to print or display musical notation.
2. **Auditing.** MIDI sound output can be generated using the **midi** and **perform** commands.
3. **Searching for things.** When searching for things within specified files, appropriate commands include: **grep**, **egrep**, **yank -m**, **patt**, **pattern**, **correl**, **simil** and **humdrum -v**. When searching for files that meet certain conditions, appropriate commands include **grep -l**, **egrep -l** and **find**. Most of these search tools rely extensively on regular expressions to define patterns of characters.

4. **Counting things.** Appropriate commands include: **wc**, **wc -l**, **grep -c**, **egrep -c**, **census** and **census -k**. Eliminating unnecessary or confounding information can be achieved using **rid**, **extract**, **grep -v**, **sed** and **humsed**.
5. **Editing things.** Manual editing may be done using any text editor, such as **emacs** or **vi**. Automated (or "stream") editing may be done using **sed** and **humsed**.
6. **Editorializing.** E.g., add an editorial footnote to a specified note or passage; indicate that a passage differs from the composer's autograph.
7. **Transforming or translating between representations.** Appropriate commands include: **cents**, **deg**, **degree**, **dur**, **freq**, **hint**, **humsed**, **iv**, **kern**, **mint**, **pc**, **pf**, **pitch**, **reihe**, **semits**, **solfa**, **solfg**, **text**, **tonh**, **trans** and **vox**.
8. **Arithmetic transformations of representations.** Manipulating numerical values can be done using **xdelta**, **ydelta**, **recode** and **awk**.
9. **Extracting or selecting information.** Appropriate commands include: **extract**, **yank**, **grep**, **egrep**, **yank -m**, **thru**, **strophe**, **rend** and **csplit**.
10. **Linking or joining information.** Appropriate commands include: **assemble**, **cat**, **cleave**, **timebase**, **context**, **ditto** and **join**.
11. **Generating inventories of things.** Appropriate commands include: **sort** and **uniq**. Once again, unnecessary or confounding information can be eliminated using **rid**, **extract**, **grep -v**, **yank -m**, **sed** or **humsed**.
12. **Classifying things.** Numerical values can be classified using **recode**; non-numerical data can be classified using **humsed**.
13. **Labelling things.** Appropriate commands include: **patt -t**, **recode**, **humsed** and **timebase**.
14. **Comparing whether things are the same or similar.** Appropriate commands include: **diff**, **diff3**, **cmp**, **correl** and **simil**.
15. **Capturing data.** MIDI data can be input via **encode** and **record.**
16. **Trouble-shooting.** Appropriate commands include: **humdrum**, **humdrum -v**, **proof**, **proof -k**, **veritas**, **midi** and **perform**.

Not all of the existing Humdrum Tools were covered in this book. Nor were all of the available options described for all of the tools discussed. Exploring the *Humdrum Reference Manual* is recommended for readers interested in continuing to develop facility with Humdrum.

Pursuing a Project with Humdrum

If you have made it this far in the book, you will now have a fairly sophisticated knowledge of Humdrum. With this background, we might review some general principles and specific tips for making effective use of Humdrum when pursuing some musicological project. The following seven questions provide useful guidelines:

1. *What do I want my final output to look like?*
   Do I want a count or inventory?

```
115 instances of ...
28 instances of ...
```

Do I want to output a found pattern?

```
pattern found in line ...
pattern not found ...
```

Do I want a comparison?

```
file X is the same as file Y
X is similar to Y
X and Y are different
```

2. *What materials are available for processing?*

   Use **find** and **grep** to locate useful materials.

3. *What materials do I need to create?*

   Use **encode** to create new data. Use **humdrum** and **proof** to check the data. If necessary, define your own Humdrum representation for a given purpose.

4. *How do I transform my data so it is easier to process?*

   Use **recode** and **humsed** to classify data into various classes — such as *up*, *down*, *leap*, *long*, *short*, *difficult*, *easy*, *clarion register*, *dominant*, etc.

   Use translating/transforming commands such as **mint**, **ydelta**, **pcset**, etc to translate the data to a different representation.

5. *What data should I eliminate?*

   Use **rid**, **extract**, **yank**, **sed**, **humsed**, **uniq**, **uniq -d** and **grep -v** to eliminate selective materials.

6. *What data do I need to coordinate?*

   Use **context** to generate contextual information. Use **assemble**, **rend** and **cleave** to link information together.

7. *How do I know my results are worthwhile?*

   Use comparative tests whenever you can. Use **scramble -r**, **scramble -t**, **tac** and **reihe -s** to generate control groups.

Appendix I

Reference Records

Reference records are formal ways of encoding "library-type" information pertaining to a Humdrum document. Reference records provide standardized ways of encoding bibliographic information — suitable for computer-based access.

Humdrum reference records are designated by three exclamation marks at the beginning of a line, followed by a multi-letter code, followed by an optional number, followed by a colon, followed by some text.

Over 80 reference codes are pre-defined in Humdrum. Each of these reference records is described below under seven categories: (1) authorship information, (2) performance information, (3) work identification information, (4) imprint information, (5) copyright information, (6) analytic information, and (7) representation information. A final section discusses how to cite electronic documents.

Authorship Information

!!!COM: Composer's name. In some cases, opinions differ regarding the best spelling of a composer's name. If so, all common spellings should be given — each alternative separated from the previous by a semicolon. E.g.

```
!!!COM: Chopin, Fryderyk; Chopin, Frederick
```

With respect to accents, refer to the discussion concerning the **!!!RLN:** reference record (see below). If a work was composed by more than one composer, then each composer's name should appear on a separate **!!!COM:** record with a number designation prior to the colon. For example,

```
!!!COM1: Composer, A.
!!!COM2: Composer, B.
```

!!!COA: Attributed composer. This may include attributions known to be false. Several attributions may be combined on a single record by separating each name by a semicolon. Note that if a document contains both **!!!COA:** and **!!!COM:** records, then the attributed composer is explicitly assumed to be false.

!!!COS: Suspected composer. This reference code indicates the belief of the editor or producer of the document as to the true identity of the composer(s). If more than one composer is suspected, each name should appear on a separate **!!!COS:** record with a number designation prior to the colon.

!!!COL: Composer's abbreviated, alias, or stage name. e.g. Madonna.

!!!COC: Composer(s) corporate name. Corporate names may include the names of popular groups (especially when the actual composer is not known). Corporate names may also include business names, e.g. Muzak.

!!!CDT: Composer's dates. The birth and death dates should be encoded using the `**Zeit` format described in the *Humdrum Reference Manual*. The `**Zeit` format provides a highly refined representation, including methods for representing uncertainty, approximation, and boundary dates (e.g. prior to ..., after ...).

!!!CNT: Nationality of the composer. This reference information is encoded using the language of the nationality. Thus a German composer is encoded as `Deutscher` rather than German, and a French composer is encoded as `Francais` rather than French. Where the composer changed nationality, successive nationalities should be listed (in chronological order) separated by semicolons.

!!!LYR: Lyricist. The name of the lyricist. If more than one lyricist was involved in the work, then each lyricist's name should appear on a separate **!!!LYR:** record with a number designation prior to the colon. If the composer was also the lyricist, this should be explicitly encoding using the independent **!!!LYR:** record — rather than implicitly assumed.

!!!LIB: Librettist. The name of the librettist. If more than one librettist was involved in the work, then each librettist's name should appear on a separate **!!!LIB:** record with a number designation prior to the colon. If the composer was also the librettist, this should be explicitly encoding using the independent **!!!LIB:** record — rather than implicitly assumed.

!!!LAR: Arranger. The name of the arranger. If more than one arranger was involved in the work, then each arranger's name should appear on a separate **!!!LAR:** record with a number designation prior to the colon.

!!!LOR: Orchestrator. The name of the orchestrator. If more than one orchestrator was involved in the work, then each orchestrator's name should appear on a separate **!!!LOR:** record with a number designation prior to the colon.

!!!TXO: Original language of vocal/choral text. The name of the language should be encoded in that language. For example, `russki` rather than `Russian`.

!!!TXL: Language of the *encoded* vocal/choral text. The name of the language should be encoded in the language used for encoding. For example, `Italiano` rather than `Italian`.

!!!TRN: Translator of text. The name of the translator of any vocal, choral, or dramatic text. If more than one translator was involved in the work, then each translator's name should appear on a separate **!!!TRN:** record with a number designation prior to the colon.

Performance Information

Humdrum representations may encode performance-activity information rather than (or in addition to) score-related information. If the representation encodes a given performance (such as a MIDI performance), then the following reference records may be pertinent.

!!!MPN: Performer's name. If more than one performer was involved in the work, then each performer's name should appear on a separate **!!!MPN:** record with a number designation prior to the colon.

!!!MPS: Suspected performer. If more than one performer is suspected, each name should appear on a separate **!!!MPS:** record with a number designation prior to the colon.

!!!MRD: Date of performance. The performance date should be encoded using the `**date` format described in the *Humdrum Reference Manual.*

!!!MLC: Place of performance. (Local language should be used.)

!!!MCN: Name of the conductor of the performance.

!!!MPD: Date of first performance. The date of first performance should be encoded using the `**date` format described in the *Humdrum Reference Manual.*

Work Identification Information

!!!OTL: Title. The title of the specific section or segment encoded in the current file. Titles must be rendered in the original language, e.g. *Le sacre du printemps.* (Title translations are encoded using other reference records.)

!!!XEN: Translated title (in English). (Note that reference codes are also available for translations to languages other than English, French, German, or Japanese.)

!!!XFR: Translated title (in French). (Note that reference codes are also available for translations to languages other than English, French, German, or Japanese.)

!!!XDE: Translated title (in German). (Note that reference codes are also available for translations to languages other than English, French, German, or Japanese.)

!!!XNI: Translated title (in Japanese). (Note that reference codes are also available for translations to languages other than English, French, German, or Japanese.)

!!!OTP: Popular Title. This reference record encodes well-known or alias titles such as Pathetique Sonata.

!!!OTA: Alternative title. This reference record encodes earlier or alternate titles.

!!!OPR: Title of larger (or parent) work from which the encoded piece is a part. For example, Gute Nacht (OTL) from *Winterreise* (OPR).

!!!OAC: Act number. For operas and musicals, this reference record encodes the act number as an Arabic (rather than Roman) numeral. The number may be preceded by the word Act as in `Act 3`.

!!!OSC: Scene number. For operas and musicals, this reference record encodes the scene number as an Arabic (rather than Roman) numeral. The number may be preceded by the word Scene as in `Scene 3`.

!!!OMV: Movement number. For multi-movement works such as sonatas and symphonies, this reference record encodes the movement number as an Arabic (rather than Roman) numeral. The number may be preceded by the word Movement or mov. etc., as in `mov. 3`.

!!!OMD: Movement designation or movement name. Typically movements may be named according to the tempo (e.g. "Allegro ma no troppo") or according to a style, genre or form (e.g. "Fugue"), or according to a programmatic title (e.g. "In Full Flower").

!!!OPS: Opus number. The number may be preceded by the word Opus as in `Opus 23`. Once again, Arabic numerals are used.

!!!ONM: Number. The number may be preceded by the abbreviations No. or Nr. as in `No. 4`.

!!!OVM: Volume. The volume number may be preceded by the abbreviation Vol. as in `Vol. 2`. Arabic numbers are used.

!!!ODE: Dedication. Name of person or organization to whom the work is dedicated. If the work was dedicated to more than one person, then each dedicatee's name should appear on a separate **!!!ODE:** record with a number designation prior to the colon.

!!!OCO: Commission. Name of person or organization that commissioned the work. If the work was commissioned by more than one person, then each commissioner's name should appear on a separate **!!!OCO:** record with a number designation prior to the colon.

!!!OCL: Collector. Name of person who collected or transcribed the work. If the work was collected by more than one person, then each collector's name should appear on a separate **!!!OCL:** record with a number designation prior to the colon.

!!!ONB: Free format note related to the title or identity of the encoded work. Nota bene. If more than one such note is encoded, each should appear on a separate **!!!ONB:** record with a number designation prior to the colon.

!!!ODT: Date of composition. The date (or period) of composition should be encoded using the `**date` or `**Zeit` formats described in the *Humdrum Reference Manual*. The `**date` and `**Zeit` formats provides a highly refined representation, including methods for representing uncertainty, approximation, and boundary dates (e.g. prior to ..., after ...).

!!!OCY: Country of composition. Local names should be used, such as 'Espana'.

!!!OPC: City, town or village of composition. Local names should be used, such as 'Den Haag.'

Imprint Information

!!!PUB: Publication status. This reference record identifies whether the document has ever been "published". One of the following English terms may appear: `published` or `unpublished`.

!!!PPR: First publisher. Name of the first publisher of the work.

!!!PDT: Date first published. The date of publication should be encoded using the `**date` format described in the *Humdrum Reference Manual*.

!!!PPP: Place first published. (Local language should be used.)

!!!PC#: Publisher's catalogue number. This should not be confused with better known scholarly catalogues, such as those of Kochel, Hoboken, etc.

!!!SCT: Scholarly catalogue abbreviation and number. E.g. BWV 551

!!!SCA: Scholarly catalogue (unabbreviated) name. E.g.`Koechel 117`.

!!!SMS: Manuscript source name. For unpublished sources, the manuscript source name.

!!!SML: Manuscript location. For unpublished sources, the location of the manuscript source.

!!!SMA: Acknowledgement of manuscript access. This reference information may be used to encode a free format acknowledgement or note of thanks to a given manuscript owner for scholarly or other access.

Copyright Information

!!!YEP: Publisher of electronic edition. This reference identifies the publisher of the electronic document.

!!!YEC: Date and owner of electronic copyright. This reference identifies the year and owner of the copyright for the electronic document.

!!!YER: Date electronic edition released.

!!!YEM: Copyright message. This record conveys any special text related to copyright. It might convey a simple warning (e.g. All rights reserved.), convey registration or licensing information, or indicate that the document is shareware.

!!!YEN: Country of copyright. This reference identifies the country in which the electronic document was created, or where the copyright was established. In effect, it identifies the country under whose laws the copyright declaration is to be interpreted.

!!!YOR: Original document. This reference identifies any original source or sources from which encoded document was prepared. Note that original documents may themselves be copyrighted, and that permission may be required in order to create an electronic derivative document. Original

documents may also have lapsed copyrights.

!!!YOO: Original document owner. If the electronic document was prepared from a copyrighted original document, this reference identifies the copyright owner of the original document. Note that unless the electronic and original documents have the same owner, some licensing agreement or other legal arrangement is necessary in order to create an electronic derivative document.

!!!YOY: Original copyright year. If the electronic document was prepared from a copyrighted original document, this reference identifies the year of copyright for the original document. Note that some licensing agreement or other legal arrangement is necessary in order to create an electronic derivative document.

!!!EED: Electronic Editor. Name of the editor of the electronic document. If more than one editor was involved in the work, then each editor's name should appear on a separate **!!!EED:** record with a number designation prior to the colon.

!!!ENC: Encoder of the electronic document. This reference identifies the name of the person or persons who encoded the electronic document. (Not to be confused with the electronic editor.) If more than one encoder was involved in the work, then each encoder's name should appear on a separate **!!!ENC:** record with a number designation prior to the colon.

!!!EMD: Document modification description. This record type is used to chronicle all modifications made to the original electronic document. EMD records should indicate the date of modification, the name of the person making the modification, and a brief description of the type of modification made. For each successive modification, a separate **!!!EMD:** record should appear with a number designation prior to the colon.

!!!EEV: Electronic edition version. This reference identifies the specific editorial version of the work. e.g. Version 1.3g Only a single **!!!EEV:** record can appear in a given electronic document.

!!!EFL: File number. Some files are part of a series or group of related files. This record indicates that the current document is file *x* in a group of *y* files. The two numbers are separated by a slash as in:

```
!!!EFL: 1/4
```

!!!EST: Encoding status. This record indicates the current status of the document as it is being produced. Free-format text may indicate that the encoding is in-progress, list tasks remaining, or indicate that the encoding is complete. **!!!EST:** records are normally eliminated prior to distribution of the document.

!!!VTS: Checksum validation number. This reference encodes the checksum number for the file — excluding the **!!!VTS:** record itself. When this record is eliminated from the file, any POSIX.2 standard **cksum** command can be used to determine whether the file originates with the publisher, or whether it has been modified in some way. (See the Humdrum **veritas** command described in Section 4.) Note that this validation process is easily circumvented by malicious individuals. For true security, the checksum value should be compared with a printed list of checksums provided by the electronic publisher.

Analytic Information

!!!AFR: Form designation. This is a free-form text record that can be used to identify the form (if appropriate) of the work. E.g. fuga, sonata-allegro, passacaglia, rounded binary, rondo.

!!!AGN: Genre designation. This is a free-form text record that can be used to identify the genre of the work. E.g. opera, string quartet, barbershop quartet.

!!!AST: Style, period, or type of work designation. This is a free-form text record that can be used to characterize the style, period, or type of work. This reference can include any term or terms deemed appropriate by the producer of the document. Designations might include keywords or keyphrases such as: Baroque, bebop, Ecole Notre Dame, minimalist, serial, reggae, slendro, heterophony, etc.

!!!ASW: Associated Work. Some works are associated with other works, such as plays, novels, paintings, films, or other musical works. E.g. Shakespeare's Othello. This reference allows associated works to be explicitly identified by author and title.

!!!AMT: Metric Classification. Meters for a file may be classified as one of the following eight categories: `simple duple`, `simple triple`, `simple quadruple`, `compound duple`, `compound triple`, `compound quadruple`, `irregular`, or `various`.

!!!AIN: Instrumentation. This reference is used to list all of the instruments (including voice) used in the work. Instruments should be encoded using the abbreviations specified by the `*I` tandem interpretation described in Appendix II. Instrument codes must appear in alphabetical order separated by spaces.† E.g.

```
!!!AIN: clars corno fagot flt oboe
```

Representation Information

!!!RLN: ASCII language setting. This reference identifies the language code in which the file was encoded. This is applicable only to computer platforms which provide extended ASCII text capabilities (e.g. Danish or Spanish characters).

!!!RDF: User-defined signifiers. All Humdrum representations provide some signifiers (ASCII characters) that remain undefined. Users are free to use these undefined signifiers as they choose. When undefined signifiers appear in a give document, the **!!!RDF:** code should be used to specify what the signifiers denote. E.g.

```
!!!RDF1: X=hands cross, left over right
!!!RDF2: x=hands cross, right over left
```

!!!RDT: Date encoded. This reference uses the Humdrum `**date` format to identify the date(s) when the document was encoded.

!!!RNB: Representation note. This reference provides a free-format text that conveys some document-specific note related to matters of representation.

!!!RWG: Representation warning. This reference may be used to encode explicit warnings concerning the encoded material.

Electronic Citation

Electronic editions of music might be cited in printed or other documents by including the following information. The author (e.g. **!!!COM:**), the title — either original title (**!!!OTL:**) or translated title (e.g. **!!!XEN:**). The editor (**!!!EED:**), publisher (**!!!YEP:**), date of publication and copyright owner (**!!!YED:**), and electronic version (**!!!EEV:**), In addition, a full citation ought to include the validation checksum (**!!!VTS:**). This number will allow others to verify that a particular electronic document is precisely the one cited. A sample electronic citation might be:

> Franz Liszt, Hungarian Rhapsody No. 8 in F-sharp minor (solo piano).
> Amsterdam: Rijkaard Software Publishers, 1994; H. Vorisek (Ed.),
> Electronic edition version 2.1, checksum 891678772.

In Humdrum files it does not matter where reference records appear. Since it is common for users to inspect the beginning of a file in order to check whether the file is being properly processed, the number of reference records at the beginning of the file should be kept to a minimum. A good habit is to place the composer, title of the work, and copyright records at the beginning of the file, and to relegate all other reference records to the end of the file.

Further Reference Record Codes

The following table provides further pre-defined reference codes not identified in the preceding discussion.

Code	Language
!!!XAL:	translated title in Albanian
!!!XAB:	translated title in Arabic
!!!XAM:	translated title in Armenian
!!!XAZ:	translated title in Azeri
!!!XBE:	translated title in Bengali
!!!XBU:	translated title in Bulgarian
!!!XCB:	translated title in Cambodian
!!!XCA:	translated title in Cantonese
!!!XHR:	translated title in Croatian
!!!XCE:	translated title in Czech
!!!XDA:	translated title in Danish
!!!XNE:	translated title in Dutch
!!!XEN:	translated title in English
!!!XET:	translated title in Estonian
!!!XSU:	translated title in Finnish
!!!XFL:	translated title in Flemish
!!!XFR:	translated title in French

!!!XGA: translated title in Gaelic
!!!XDE: translated title in German
!!!XGR: translated title in Greek
!!!XHB: translated title in Hebrew
!!!XHI: translated title in Hindi
!!!XHU: translated title in Hungarian
!!!XIC: translated title in Icelandic
!!!XIT: translated title in Italian
!!!XNI: translated title in Japanese
!!!XJV: translated title in Javanese
!!!XKO: translated title in Korean
!!!XLI: translated title in Lithuanian
!!!XLA: translated title in Latin
!!!XLV: translated title in Latvian
!!!XMG: translated title in Malayalam
!!!XMA: translated title in Mandarin
!!!XMO: translated title in Mongolian
!!!XNO: translated title in Norwegian
!!!XPL: translated title in Polish
!!!XPR: translated title in Portugese
!!!XRO: translated title in Romanian
!!!XRU: translated title in Russian
!!!XSR: translated title in Serbian
!!!XSK: translated title in Slovak
!!!XSN: translated title in Slovenian
!!!XES: translated title in Spanish
!!!XSW: translated title in Swahili
!!!XSV: translated title in Swedish
!!!XTA: translated title in Tamil
!!!XTH: translated title in Thai
!!!XTI: translated title in Tibetan
!!!XTU: translated title in Turkish
!!!XUK: translated title in Ukranian
!!!XUR: translated title in Urdu
!!!XVN: translated title in Vietnamese
!!!XWE: translated title in Welsh
!!!XHO: translated title in Xhosa
!!!XZU: translated title in Zulu

Appendix II

Instrumentation Codes

Introduction

Humdrum provides standard voice and instrument names in order to facilitate various tasks. Voice and instrument names are used both in tandem interpretations, and in `AIN` reference records identifying the instrumentation for a work (see Appendix I).

Instrument tandem interpretations are used to identify the instrumentation pertaining to a specified spine. The word "instrument" is used in a broad sense and embraces vocal qualities and vocal types as well as mechanical sound makers.

Three distinctions are currently made in Humdrum: *instrument name, instrument class* and *instrument group.* The following table identifies six pre-defined instrument classes:

`*ICvox`	voice
`*ICstr`	string instrument
`*ICww`	woodwind instrument
`*ICbras`	brass instrument
`*ICklav`	keyboard instrument
`*ICidio`	percussion instrument (idiophone)

The following table identifies five pre-defined instrument groups:

`*IGacmp`	accompaniment instrument
`*IGsolo`	solo instrument
`*IGcont`	basso-continuo instrument
`*IGripn`	ripieno instrument
`*IGconc`	concertino instrument

The following set of tables list currently defined instrument names. When the name is used to form an instrument tandem interpretation the instrument keyword is preceded by an asterisk and the upper-case letter '`I`'. For example, the instrument tandem interpretation for the guitar is `*Iguitr`. If the guitar appears as an instrument in a `AIN` Reference Record only the designation '`guitr`' is used.

Voice Range

`soprn`	soprano
`mezzo`	mezzo soprano
`calto`	contralto
`tenor`	tenor
`barit`	baritone
`bass`	bass

Voice Quality

`vox`	generic (undesignated) voice
`feme`	female voice
`male`	male voice
`nfant`	child's voice
`recit`	recitativo
`lyrsp`	lyric soprano
`drmsp`	dramatic soprano
`colsp`	coloratura soprano
`alto`	alto
`ctenor`	counter-tenor
`heltn`	Heldentenor, tenore robusto
`lyrtn`	lyric tenor
`bspro`	basso profondo
`bscan`	basso cantante
`false`	falsetto
`castr`	castrato

String Instruments

`archl`	archlute; *archiluth* (Fr.); *liuto attiorbato/arcileuto/arciliuto* (It.)
`arpa`	harp; *arpa* (It.), *arpa* (Span.)
`banjo`	banjo
`biwa`	biwa
`bguit`	electric bass guitar
`cbass`	contrabass
`cello`	violoncello
`cemba`	harpsichord; *clavecin* (Fr.); *Cembalo* (Ger.); *cembalo* (It.)
`cetra`	cittern; *cistre/sistre* (Fr.); *Cither/Zitter* (Ger.); *cetra/cetera* (It.)
`clavi`	clavichord; *clavicordium* (Lat.); *clavicorde* (Fr.)
`dulc`	dulcimer or cimbalom; *Cimbal* or Hackbrett (Ger.)
`eguit`	electric guitar
`forte`	fortepiano
`guitr`	guitar; *guitarra* (Span.); *guitare* (Fr.); *Gitarre* (Ger.); *chitarra* (It.)
`hurdy`	hurdy-gurdy; variously named in other languages
`liuto`	lute; *lauto, liuto leuto* (It.); *luth* (Fr.); *Laute* (Ger.)

`kit`	kit; variously named in other languages
`kokyu`	kokyu (Japanese spike fiddle)
`komun`	kɸmun'go (Korean long zither)
`koto`	koto (Japanese long zither)
`mando`	mandolin; *mandolino* (It.); *mandoline* (Fr.); *Mandoline* (Ger.)
`piano`	pianoforte
`pipa`	Chinese lute
`psalt`	psaltery (box zither)
`qin`	qin, ch'in (Chinese zither)
`quitr`	gittern (short-necked lute); *quitarre* (Fr.); *Quinterne* (Ger.)
`rebec`	rebec; *rebeca* (Lat.); *rebec* (Fr.); *Rebec* (Ger.)
`sarod`	sarod
`shami`	shamisen (Japanese fretless lute)
`sitar`	sitar
`tambu`	tambura
`tanbr`	tanbur
`tiorb`	theorbo; *tiorba* (It.); *tèorbe* (Fr.); *Theorb* (Ger.)
`ud`	ud
`ukule`	ukulele
`vina`	vina
`viola`	viola; *alto* (Fr.); *Bratsche* (Ger.)
`violb`	bass viola da gamba; *viole* (Fr.); *Gambe* (Ger.)
`viold`	viola d'amore; *viole d'amour* (Fr.); *Liebesgeige* (Ger.)
`violn`	violin; *violon* (Fr.); *Violine* or *Geige* (Ger.); *violino* (It.)
`violp`	piccolo violin; *violino piccolo* (It.)
`violp`	piccolo violin; *violino piccolo* (It.)
`viols`	treble viola da gamba; *viole* (Fr.); *Gambe* (Ger.)
`violt`	tenor viola da gamba; *viole (Fr.); Gambe (Ger.)*
`zithr`	zither; *Zither* (Ger.); *cithare* (Fr.); *cetra da tavola* (It.)

Wind Instruments

`accor`	accordion; *accordéon* (Fr.); *Akkordeon* (Ger.)
`armon`	harmonica; *armonica* (It.)
`bagpS`	bagpipe (Scottish)
`bagpI`	bagpipe (Irish)
`baset`	bassett horn
`calam`	chalumeau; *calamus* (Lat.); *kalamos* (Gk.)
`calpe`	calliope
`cangl`	english horn; *cor anglais* (Fr.)
`chlms`	soprano shawm, chalmeye, shalme, etc.; *chalemie* (Fr.); *ciaramella* (It.)
`chlma`	alto shawm, chalmeye, shalme, etc.
`chlmt`	tenor shawm, chalmeye, shalme, etc.
`clars`	soprano clarinet (in either B-flat or A); *clarinetto* (It.)
`clarp`	piccolo clarinet
`clara`	alto clarinet (in E-flat)
`clarb`	bass clarinet (in B-flat)
`cor`	horn; *cor* (Fr.); *corno* (It.); *Horn* (Ger.)

`cornm`	cornemuse; French bagpipe
`corno`	cornett (woodwind instr.); *cornetto* (It.); *cornaboux* (Fr.); *Zink* (Ger.)
`cornt`	cornet (brass instr.); *cornetta* (It.); *cornet à pistons* (Fr.); *Cornett* (Ger.)
`ctina`	concertina; *concertina* (Fr.); *Konzertina* (Ger.)
`fagot`	bassoon; *fagotto* (It.)
`fag_c`	contrabassoon; *contrafagotto* (It.)
`fife`	fife
`flt`	flute; *flauto* (It.); *Flöte* (Ger.); *flûte* (Fr.)
`flt_a`	alto flute
`flt_b`	bass flute
`fltds`	soprano recorder; *flûte à bec, flûte douce* (Fr.); *Blockflöte* (Ger.); *flauto dolce* (It.)
`fltdn`	sopranino recorder
`fltda`	alto recorder
`fltdt`	tenor recorder
`fltdb`	bass recorder
`flugh`	flugelhorn
`hichi`	hichiriki (Japanese double reed used in gagaku)
`krums`	soprano crumhorn; *Krummhorn/Krumbhorn* (Ger.); *tournebout* (Fr.)
`kruma`	alto crumhorn
`krumt`	tenor crumhorn
`krumb`	bass crumhorn
`nokan`	nokan (Japanese flute for the no theatre)
`oboe`	oboe; *hautbois* (Fr.); *Hoboe, Oboe* (Ger.): *oboe* (It.)
`oboeD`	oboe d'amore
`ocari`	ocarina
`organ`	pipe organ; *organum* (Lat.); *organo* (It.); *orgue* (Fr.); *Orgel* (Ger.)
`panpi`	panpipe
`picco`	piccolo
`porta`	portative organ
`rackt`	racket; *Rackett* (Ger.); *cervelas* (Fr.)
`reedo`	reed organ
`sarus`	sarrusophone
`saxN`	sopranino saxophone (in E-flat)
`saxS`	soprano saxophone (in B-flat)
`saxA`	alto saxophone (in E-flat)
`saxT`	tenor saxophone (in B-flat)
`saxR`	baritone saxophone (in E-flat)
`saxB`	bass saxophone (in B-flat)
`saxC`	contrabass saxophone (in E-flat)
`shaku`	shakuhachi
`sheng`	mouth organ (Chinese)
`sho`	mouth organ (Japanese)
`sxhS`	soprano saxhorn (in B-flat)
`sxhA`	alto saxhorn (in E-flat)
`sxhT`	tenor saxhorn (in B-flat)
`sxhR`	baritone saxhorn (in E-flat)
`sxhB`	bass saxhorn (in B-flat)
`sxhC`	contrabass saxhorn (in E-flat)

`tromt`	tenor trombone; *trombone* (It.); *trombone* (Fr.); *Posaune* (Ger.)
`tromb`	bass trombone
`tromp`	trumpet; *tromba* (It.); *trompette* (Fr.); *Trompete* (Ger.)
`tuba`	tuba
`zurna`	zurna

Percussion Instruments

`bdrum`	bass drum (kit)
`campn`	bell; *campana* (It.); *cloche* (Fr.); *campana* (Span.)
`caril`	carillon
`casts`	castanets; *castañ'etas* (Span.); *castagnette* (It.)
`chime`	chimes
`clest`	celesta; *céleste* (Fr.)
`crshc`	crash cymbal (kit)
`fingc`	finger cymbal
`glock`	glockenspiel
`gong`	gong
`marac`	maracas
`marim`	marimba
`piatt`	cymbals; *piatti* (It.); *cymbales* (Fr.); *Becken* (Ger.); *kymbos* (Gk.)
`ridec`	ride cymbal (kit)
`sdrum`	snare drum (kit)
`spshc`	splash cymbal (kit)
`steel`	steel-drum, tinpanny
`tabla`	tabla
`tambn`	tambourine, timbrel; *tamburino* (It.); *Tamburin* (Ger.)
`timpa`	timpani; *timpani* (It.); *timbales* (Fr.); *Pauken* (Ger.)
`tom`	tom-tom drum
`trngl`	triangle; *triangle* (Fr.); *Triangel* (Ger.); *triangolo* (It.)
`vibra`	vibraphone
`xylo`	xylophone; *xylophone* (Fr.); *silofono* (It.)

Keyboard Instruments

`accor`	accordion; *accordéon* (Fr.); *Akkordeon* (Ger.)
`caril`	carillon
`cemba`	harpsichord; *clavecin* (Fr.); *Cembalo* (Ger.); *cembalo* (It.)
`clavi`	clavichord; *clavicordium* (Lat.); *clavicorde* (Fr.)
`clest`	celesta; *céleste* (Fr.)
`forte`	fortepiano
`hammd`	Hammond electronic organ
`organ`	pipe organ; *orgue* (Fr.); *Orgel* (Ger.); *organo* (It.); *organo* (Span.); *organum* (Lat.)
`piano`	pianoforte
`porta`	portative organ
`reedo`	reed organ
`rhode`	Fender-Rhodes electric piano
`synth`	keyboard synthesizer

Index of Problems

Index of Names, Works and Genres

General Index

415

Bibliography

Aho, A., Kernighan, B. & Weinberger, P. *The AWK Programming Langauge.* Reading, Massachusetts: Addison-Wesley Publishing Co. 1988.

Alphonce, B. Music analysis by computer. *Computer Music Journal,* Vol. 4, No. 2 (1980) pp. 26-35.

Alphonce, B. Computer applications: Analysis and modeling. *Music Theory Spectrum,* Vol. 11, No. 1 (1989) pp. 49-59.

Baroni, M. & Callegari, L. (eds.) *Musical Grammars and Computer Analysis.* Firenze: Leo S. Olschki Editore, 1984.

Bauer-Mengelberg, S. The Ford-Columbia input language. In B. Brook (ed.), *Musicology and the Computer,* New York: City University of New York, 1970; pp. 53-56.

Brinkman, A. Representing musical scores for computer analysis. *Journal of Music Theory,* Vol. 30, No. 2 (1986) pp. 225-275.

Brinkman, A. *Pascal Programming for Music Research.* Chicago: University of Chicago Press, 1990.

Bronson, B. Toward the comparative analysis of British-American folk tunes. *Journal of American Folklore,* Vol. 72, No. 284 (1959) pp. 165-191.

Cole, H. *Sounds and Signs; Aspects of Musical Notation.* London: Oxford University Press, 1974.

Cope, D. *New Music Notation.* Dubuque, Iowa: Kendall/Hunt Publishing Co., 1976.

Erickson, R. The DARMS project: A status report. *Computers and the Humanities,* Vol. 9 ((1975) pp. 91-298.

Erickson, R. *DARMS; A Reference Manual.* Binghamton, NY: typescript, 1976.

Dannenberg, R. Music representation issues, techniques, and systems. *Computer Music Journal,* Vol. 17, No. 3 (1993) pp. 20-30.

Forte, A. *The Structure of Atonal Music.* New Haven: Yale University Press, 1973.

Hall P. & Dowling, G. Approximate string matching. *ACM Computing Surveys,* Vol. 12 (1980) pp. 381-402.

Hall, T. DARMS: The A-R Dialect. In E. Selfridge-Field (ed.), *Beyond MIDI: The Handbook of Musical Codes.* Cambridge, MA: MIT Press, 1997; pp. 193-214.

Halperin, D. Afterword: Guidelines for new codes. In E. Selfridge-Field (ed.), *Beyond MIDI: The Handbook of Musical Codes.* Cambridge, MA: MIT Press, 1997; pp. 573-580.

Haus, G. (ed.) *Music Processing.* Madison, WI: A-R Editions, Inc., and Oxford: Clarendon Press, 1993.

Hewlett, W. B. A base-40 number-line representation of musical pitch notation. *Musikometrika,* Vol. 4 (1992) pp. 1-14.

Hewlett, W. B. The representation of musical information in machine-readable format. *Directory of Computer Assisted Research in Musicology 1987.* Menlo Park: Center for Computer Assisted Research in the Humanities, 1987; pp. 1-22.

Hewlett, W. B. & Selfridge-Field, E. (eds.) *Directory of Computer Assisted Research in Musicology 1988.* Menlo Park: Center for Computer Assisted Research in the Humanities, 1988.

Hewlett, W. B. & Selfridge-Field, E. (eds.) *Computing in Musicology; A Directory of Research;* Menlo Park: Center for Computer Assisted Research in the Humanities, 1989.

Hewlett, W. B. & Selfridge-Field, E. (eds.) *Computing in Musicology; A Directory of Research;* Volume 6. Menlo Park: Center for Computer Assisted Research in the Humanities, 1990.

Hewlett, W. B. & Selfridge-Field, E. (eds.) *Computing in Musicology; A Directory of Research;* Volume 7. Menlo Park: Center for Computer Assisted Research in the Humanities, 1991.

Hewlett, W. B. & Selfridge-Field, E. (eds.) *Computing in Musicology; An International Directory of Applications;* Volume 8. Menlo Park: Center for Computer Assisted Research in the Humanities, 1992.

Howard, J. Plaine and Easie Code: A Code for Music Bibliography. In E. Selfridge-Field (ed.), *Beyond MIDI: The Handbook of Musical Codes.* Cambridge, MA: MIT Press, 1997; pp. 343-361.

Huron, D. Error categories, detection, and reduction in a musical database. *Computers and the Humanities,* Vol. 22, No. 2 (1988) pp. 253-264.

Huron, D. Characterizing musical textures. *Proceedings of the 1989 International Computer Music Conference,* San Francisco: Computer Music Association, 1989; pp. 131-134.

Huron, D. The avoidance of part-crossing in polyphonic music: Perceptual evidence and musical practice. *Music Perception,* Vol. 9, No. 1 (1991) pp. 93-104.

Huron, D. Tonal consonance versus tonal fusion in polyphonic sonorities. *Music Perception,* Vol. 9, No. 2 (1991) pp. 135-154.

Huron, D. Design principles in computer-based music representation. In A. Marsden & A. Pople (eds.), *Computer Representations and Models in Music.* London: Academic Press, 1992; pp. 5-59.

Huron, D. & Royal, M. What is melodic accent? Converging evidence from musical practice. *Music Perception,* Vol. 13, No. 4 (1996) pp. 498-516.

Johnson-Laird, P. Rhythm and meter: A theory at the computational level. *Psychomusicology,* Vol. 10 (1991) pp. 88-106.

Kameoka, A. & Kuriyagawa, M. Consonance theory, part I: Consonance of dyads. *Journal of the Acoustical Society of America,* Vol. 45, No. 6 (1969a) pp. 1451-1459.

Kameoka, A. & Kuriyagawa, M. Consonance theory, part II: Consonance of complex tones and its calculation method. *Journal of the Acoustical Society of America,* Vol. 45, No. 6 (1969b) pp. 1460-1469.

Kippen, J. & Bernard, B. Modelling music with grammars: Formal language representation in the Bol processor. In A. Marsden & A. Pople (eds.), *Computer Representations and Models in Music.* London: Academic Press, 1992; pp. 207-238.

Kornstädt, A. SCORE-to-Humdrum: A graphical environment for musicological analysis. *Computing in Musicology,* Vol. 10 (1995-96) pp. 105-122.

Marsden, A. & Pople, A. (eds.) *Computer Representations and Models in Music.* London: Academic Press, 1992.

Mathews, M. *The Technology of Computer Music.* Cambridge MA: Massachusetts Institute of Technology Press, 1969.

Moles, A. *Information Theory and Esthetic Perception.* Urbana: University of Illinois Press, 1968.

Moore, F. R. *Elements of Computer Music.* Englewood Cliffs, NJ: Prentice Hall, 1990.

Mortice Kern Systems Inc. *The MKS Toolkit.* Mortice Kern Systems Inc., 35 King Street North, Waterloo, Ontario, Canada N2J 2W9.

Nettheim, N. On the accuracy of musical data, with examples from Gregorian Chant and German folksong. *Computers and the Humanities,* Vol. 27, (1993) pp. 111-120.

Newcomb, S. Standard music description language complies with hypermedia standard. *IEEE Computer,* Vol. 24, No. 7 (1991) pp. 76-79.

Newcomb, S., Kipp, N. & Newcomb, V. The HyTime hypermedia time-based document structuring language. *Communications of the ACM,* Vol. 34, No. 11 (1991) pp. 67-83.

Opolko, F., & Wapnick, J. *McGill University Master Samples.* Montreal: McGill University, 1987, 1989.

Orpen, K. & Huron, D. Measurement of similarity in music: A quantitative approach for non-parametric representations. *Computers in Music Research,* Vol. 4 (1992) pp. 1-44.

O Maidiń, D. Representation of music scores for analysis. In A. Marsden & A. Pople (eds.), *Computer Representations and Models in Music.* London: Academic Press, 1992; pp. 67-93.

Overill, R. E. On the combinatorial complexity of fuzzy pattern matching in music analysis. *Computers and the Humanities,* Vol. 27 (1993) pp. 105-110.

Pope, S. T. Music notations and the representation of musical structure and knowledge. *Perspectives of New Music,* Vol. 24 (1986) pp. 156-189.

Prather, R. & Elliot, R. SML: A structured musical language. *Computers and the Humanities,* Vol. 22 (1988) pp. 137-151.

Rahn, J. *Basic Atonal Theory.* New York: Longman Inc., 1980.

Rastall, R. *The Notation of Western Music.* London: J. M. Dent & Sons Ltd., 1983.

Roads, C. An overview of music representations. In M. Baroni & L. Callegari (eds.) *Musical Grammars and Computer Analysis.* Firenze: Leo S. Olschki Editore, 1984, pp. 7-37.

Sandell, G. *Concurrent timbres in orchestration: A perceptual study of factors determing "blend."* PhD dissertation, Northwestern University, Evanston, Illinois, 1991.