OMDN at the C@merata 2015 Task: A Description of the CPNView Approach to Answering Natural Language Questions about Music Scores

Donncha S. Ó Maidín
University of Limerick
Department of Computer Science and Information Systems,
Limerick, Ireland
Donncha.OMaidin@ul.ie

ABSTRACT
The task is concerned with using a natural language query and a file containing an encoding of a music score to produce a list of locations in the score that match the query. This task is achieved in four stages: (1) parsing the input query string, (2) formation of matching templates, (3) performing the search on the score and (4) reporting results in the specified format. Parsing is performed using string processing. Searches on the score are performed using CPNView[2,3,4]. CPNView implements a container-iterator representation of the score that is generated from MusicXML representations.

1. INTRODUCTION
The focus of this work is on searches for sequences of notes and rests. Such queries account for the majority of the questions posed in the C@merata task, but exclude other queries, such as those dealing with harmony. The description of the task is given in Task Description V2[1], where the task is specified as a combination of explicit rules and examples. An example of an explicit rule is quoted in 1.1.1 in reference to the 'followed by' text. However these explicit rules are insufficient for interpreting the contents of queries. It is necessary to make assumptions about the structure and interpretation of queries, since the more formally stated rules are incomplete. One example is where note names are always given in Task Description V2 in upper case. No instance is given where a name occurs in lower case, although there is not any indication that this must necessarily be the case.

1.1 Query Structure for Note and Rest Sequences
The tackling of a sub-set of the queries involving the matching of strings of notes and rests, can be broken down into elements representing the notes or the rests separated by operators that link elements together. Some examples are

Ab2 followed by Eb3
Bb5, G5, F#5, E5, Eb5
F# E G F# A

1.1.1 Operators
The three different operators are contained in the above strings; (1) ‘ followed by ‘, (2) ’ (comma + space), and (3) ‘ ’ (space). In order to correctly parse the string, it is essential to take into account the context-sensitive nature of some of these operators. For example the space appears in many different contexts and should be regarded as a candidate operator only when it occurs between two elements where the element on the left side has been fully and exhaustively identified.

An example is given in Task Description of the ‘ , ’ (comma + space) operator, together with some explanatory text:

C, D, E “all assumed natural, adjacent, any octave pitch, commas optional in lists” (on page 4).

The text given for the ‘ followed by ’ operator is more specific:

“There must be no gap between one element and the next, and neither are they allowed to overlap.” And “Concerning two melodic elements one followed by another, they must both be on the same stave.”

Many issues are raised here such as (1) whether the rules for the ‘ followed by ’ operator also apply to the ‘ , ’ (comma + space) operator and the ‘ ’ (space) operator; (2) whether the word ‘adjacent’ means ‘notationally adjacent’ or ‘sonically adjacent’. Staccato notes are notationally but not sonically adjacent. In contrast notes or rests that are separated by bar lines are sonically, but not notationally adjacent. In such cases it might be reasonable to interpret both cases such as consecutive staccato notes or notes separated by barlines as satisfying the required adjacency criterion.

1.1.2 Pitch Names
An unstated rule that might be inferred is that pitch classes names for all notes are specified using capital letters – for example A, or A# or A#2 (in that sequence) or A2 sharp. This style is used in all of the examples given in the Task Description. However in the query set, one query appears as

'sixteenth note f'

If the lower case ‘f’ instead of ‘F’ is intentional, catering for such in general will make the parsing and the verification task more complex. Cases such as the following might arise

a b followed by a c

There are three valid interpretations of this query, depending of whether one, or both, or none of the ‘a’s are interpreted as pitches. Catering for such introduces a requirement to evaluate multiple valid answer strings in order to cater for the ambiguities.

Copyright is held by the author/owner(s).
1.1.3 Element Core

The core forms the unqualified part of the query element that identifies some basic feature of a note or rest, such as pitch, pitch class and duration, either singly or in combination. The core can be parsed unambiguously in a straightforward manner, provided that the initial analysis is performed correctly.

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>pitch class (note)</td>
<td>A, Bb, C sharp</td>
</tr>
<tr>
<td>Pitch (note)</td>
<td>A3, Bb4, C4 sharp</td>
</tr>
<tr>
<td>pitch class and duration (note)</td>
<td>A crotchet, eight note A, A eight note</td>
</tr>
<tr>
<td>pitch and duration (note)</td>
<td>A#4 crotchet, crotchet A4 sharp</td>
</tr>
<tr>
<td>Duration (note)</td>
<td>sixteenth note, quaver</td>
</tr>
<tr>
<td>Rest</td>
<td>sixteenth note rest, quaver rest</td>
</tr>
</tbody>
</table>

1.1.4 Element Pre-Qualifiers

Element pre-qualifiers precede the element core. They serve a number of functions. These include the specification of repetition factors for the element core (...):

- melody lasting 8 ...
- melody of seven ...

Also the pre-qualifier is used to specify one or more attributes of the element core (...) such as:

- trill on a … [attribute + ‘on a ’ + element core]
- staccato … [ attribute + element core]
- slurred … [adjectival attribute + element core]

Pre-qualifier may also combine the numeric specifications with one or more attributes of the element core (...), for example:

- ten staccato ...

1.1.5 Element Post-Qualifiers

Post-qualifiers are used to limit the search for element cores (...) to specific instruments or clefs, time signatures, articulations, or locations.

- … in the Violins 2
- … in the treble clef
- … in bars 22-32
- … in the right hand
- … in 3/4 time
- … on the word “Arm!”

1.2 The Scores in MusicXML

Part-wise MusicXML is used to encode details such as clefs, time signatures, key signatures, bar lines, notes and rests, and measures are organised by a combination of parts and voices. Notes and rests carry an associated voice number. The voice number is a mechanism of MusicXML that facilitates internal organisation, but does not correspond directly to any entity in the original score. The numbers 1 and 5 are frequently assigned as voice identifiers.

Staves from the original score may be organised within MusicXML in some scores using the 'part' structure. In the case of a multi-stave score, this entails encoding each stave separately. The MusicXML file then consists of sequences of parts, each one of which corresponds to a stave in the original score.

An alternate mechanism for encoding multi-stave scores is used where a group of staves are encoded in a single part. This mechanism also involves using the <backup> and on occasions the <forward> mechanism within each measure to place the contained elements in the correct temporal location.

The encoding of some scores uses a combination of multi-part structure and backup techniques.

Also the combination of voice and backup tags are essential in order to encode simultaneous notes and rests on the same stave where they have contrasting or overlapping time values.

MusicXML files are normally generated from native representations within notation packages. There appears to be a lack of standardization on how such output is created in relation to the use of parts, and voices.

Some of the MusicXML scores used in this challenge contain errors and strange practices. Ornamented notes are encoded as notated but also overlaid with notes of the expanded ornament using the same voice tag identifier. MuseScore 2 deals with such cases by unsatisfactorily overprinting to facilitate this anomaly. The scores of Mozart's Sonata Facile K.545 and Scarlatti's K.30 sonata are instances of such.

Instances are present where notes and rests fail to fill the bar. The second movement of Beethoven's op.18 no.1 is in 9/8 time. However violin 1 has only a semi-breve rest in the first full bar. As 9/8 is the sum of 8/8(semibreve) + 1/8(quaver) or whole-note + eight-note this bar falls short of the correct cumulative duration. There are many other similar instances as well as more complex inconsistencies in the representation of the same score movement.

In bar 12 of the Mozart Horn Duos, a mid-measure bar line indicating repeats, is encoded and followed immediately by a measure tag although the bar is incomplete. This barline/measure entry has the number 13, effectively resulting in measure 12 being counted as two measures. This results in setting subsequent measure numbers one ahead of their correct value. MuseScore 2 ignores this and re-numbers subsequent measures correctly, but at variance with what appears in the XML.

1.3 CPNView

Common Practice Notation View, or CPNView is used to answer a subset of the questions in the C@merata challenge[1]. CPNView formed the main topic of the author’s PhD dissertation[2]. The name CPNView was not used in the dissertation, but appeared in in later publications [3][4][5].

The meaning of symbols in a score depends on their preceding context. Examples are note stress and pitch. The emphasis a performer places on a note is influenced by its position in the bar and by the time signature. Such contexts may be modified by an attached symbol such as marcato placed on an otherwise unstressed note. Contextual mechanism is employed in pitch representation in which key signature, clef and accidental alterations play a part. In CPNView, the user is freed from the need to keep track of such scoping concerns, as contexts are made available automatically by the iterator class.
CPNView models a score as an object-oriented container in a manner similar to that used for other data structures found in computer science textbooks. The CPNView model is designed to provide a value-neutral and objective representation of a score from common-practice notation. The score's internal content is available using iterators. The iterator object keeps track of the context in which an object resides in addition to providing access to the score object itself through its member functions. The iterators and their member functions can be viewed as paralleling the actions of a human reader. Typically, a human might access a score from the start and read through it serially in time sequence. For some purposes the reading may traverse the score along one of the staves. Where a harmonic or polyphonic texture is of interest it will be desirable to access it as a sequence of vertical slices in time order.

A score object is created by specifying a file path:

```cpp
Score score(path);
```

This model requires no user knowledge of how the score is represented in a file. CPNView representation is built from a software component that imports from files in a number of different standard encodings.

Access to the internals of the score is facilitated by an iterator object:

```cpp
ScoreIterator cursor(score);
```

The first instance creates an iterator that initially points to the start and can be used to visit all of the objects in a score in time order. Where the score contains multiple staves, this is an appropriate iterator for harmonic analysis. The second form has an additional parameter and is used to iterate a single stave; in the example the stave with identifier 1 is selected.

In either case the iterator can be made to step through all of the objects in the score using the `step` member function. The `step` function returns a value `true` as long as a succeeding object exists. The following code skeleton makes all objects available, in sequence to any code that replaces the ellipsis.

```cpp
while ( cursor.step() ) {...}
```

If it is required to visit only the notes in the score, a parameter may be given to the `step` function as in the following code to count the notes in a score.

```cpp
long count = 0;
while ( cursor.step(NOTE) ) {count++;}
```

A `locate` member may be used to place the score iterator in an arbitrary position. For example the iterator may be positioned at the start of bar 20 by means of

```cpp
cursor.locate(BAR, 200);
```

The `ScoreIterator` object has a comprehensive range of member functions to retrieve all of the information that is contained within the score.

A natural language query that searches for all of the D notes and prints details of each note arrived at is achieved by

```cpp
while ( cursor.step(NOTE) )
if ( cursor.getAlpha() == 'D' )
    cout << cursor << "n";
```

In addition to modeling a score, CPNView has a set of components that facilitate processing musical information. They include container/iterator classes for Lists, Sets and Stores.

A specialised class exist for calculating pitch class sets. The pitch class object is based on a modified version of the classification system of Alan Forte[6]. It has been modified for the classifying tonal, rather than atonal combinations of pitches such as those that occur in scales, modes and in harmony[2]. At present the underlying structure of CPNView is undergoing a major revision.

### 2. APPROACH

#### 2.1 Parsing the Input String

A string, S(n) with n elements separated by n-1 operators is processed as follows. A list consisting of search template records is formed where each string element generates a corresponding search record, in sequence. Any failure to identify an element causes this procedure to be abandoned. In a fully developed version, the parse of S(1) will be expanded to cater for harmonic as well as melodic sequences or alternately, in which case control will be passed to additional algorithms for analysis.

<table>
<thead>
<tr>
<th>Step</th>
<th>Form</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S(1)</td>
<td>Parse, create a search template record, and add the template record to the template list, exit and perform search</td>
</tr>
<tr>
<td>2</td>
<td>S(n)</td>
<td>Structure the string as [ S(1) + operator + S(n-1) ], process S(1) to create a new list member as in step (1) and recursively apply step (2) to S(n-1) until exit from step (1).</td>
</tr>
</tbody>
</table>

#### 2.2 Search Template

The template is a C++ record consisting of native C++ types, and various convenient types from CPNView, as well as a `comparisonType` that is used to select the type of search. The `comparisonType` value is created from an analysis of the core of the search string element. The part of the template used for matching note and rest core elements is

```cpp
comparisonTypes comparisonType;
char alpha;
long oc;
accidType ac;
Duration dur;
```

where `comparisonType` has a value that represents the type of comparison using a value from `PITCHCLASSONLY, PITCHONLY, DURATIONONLY, FULLNOTEONLY, DURATIONWITHPITCHCLASS, RESTONLY, DURATIONRESTONLY, INTERVALONLY OR NOCTYPE`.

alpha is the note name and has values from A to G, oc is the octave number.
ac is an enumerated variable representing the accidental type with one of the values from \{NOACCID, F, S, N, DF, DS\}

dur is a record with structure

\[
\begin{align*}
durType & \text{dur}; \\
\text{long dots;}
\end{align*}
\]

dur has one of the enumerated values \{N0, N1, N2, N4, N8, N16, N32, N64, N128, Nundefined\};
dots represents the number of dots.

Search details from the pre- and post element qualifiers are encoded in additional fields of the search template.

Set nr;
String inThe;
handType hand;
Instrument in;
String word;
String tempo;
long startBarNo;
long endBarNo;
clefType clef;

Set is a CPNView class for representing sets of attributes. CPNView has a large number of enumerated attributes representing attributes of notes and rest, that can be represented in any combination in the Set object, attributes such as STACCATO, TIE_FROM, TIE_TO, TENUTO, PLUS, FERMATA, etc.

InThe is a string that stores the following text when not otherwise classified
Instrument is a CPNView class that stores the instrument name together with transposition details
The string word is used for matching lyrics
The range of the search is stored in startBarNo and endBarNo.
clefType stores clef matching details.

Other fields in the matching template that are not listed are used in harmonic and interval searches. These are omitted here since components for doing these searches were not working reliably in time for the 2015 deadline.

2.3 The Search

The task of matching sequences of notes and rests involves matching members of the template list and exhaustively searching the score for sequences of notes and rests that match the list of templates.

The MusicXML score is input to create a score container object. Score iterators are created for each stave of the score, and these are sequentially searched for a match of the first element in the list.

For each stave

A: set \( n = 1 \)
B: set \( m = 1 \)
set \( mn = n \)
C: visit element \( nn \) on stave and check for match with template list record \( m \)

if element does not match template list record go to D:
if template list at end record a match and go to D:
else increment \( mn \) and go to C:
D: if more elements in stave, increment \( n \) and go to B:
else if all staves processed exit
else move to next stave

3. Results and Discussion

As work on the component for importing MusicXML into a revised CPNView structure was ongoing by the task deadline, and as the development of the search algorithms was at a very early stage, it was possible to complete only the most basic tasks of identifying unqualified single elements. Although the implementation of pre- and post-qualifiers and of inter-element operators had previously produced valid results, these had to be disregarded at the time of submission due to malfunction in the CPNView/MusicXML component. Additionally previously functioning harmonic analysis had to be abandoned for similar reasons. Even with this reduction in the task, there were two instances of malfunction due to unmatching bar numbers – problems that were rectified shortly after the submission deadline.

<table>
<thead>
<tr>
<th>Number of searches performed</th>
<th>Number of matches</th>
<th>Invalid results from misnumbered bars</th>
<th>Disputed results</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>87</td>
<td>Mozart's An Chloe &amp; Horn Duo</td>
<td>Nos. 23 32 42</td>
</tr>
</tbody>
</table>

4. Conclusion

4.1 Observation

Two cases of disputed results would appear to be due to the continued failure to take account of the how the Gold Standard deals with accidental alterations on notes within bars. All notes following an alteration, all similar notes at the same pitch level inherit the same accidental alteration. With the exception of the two invalid sets of results other case achieved scores of 1.0 in beat precision, beat recall, measure precision and measure recall. Other cases arose from either incompatible bar numbering or where the results are disputed.

4.2 Recommendation on Score Standards for Future Trials

It is highly desirable that MusicXML scores are vetted for anomalies before being used. Cases include (1) inconsistent bar lengths, (2) minimizing the amount of additional entries and, as far as is feasible, filtering out material that is superfluous to score representation and (3) adopting a uniform approach to measure numbering. For example the first complete measure could be guaranteed to have the number 1. For offending shorter scores, bar numbers could be normalized by manual editing.

4.3 Recommendations on the Use of Beats per Quarter-Note

The beats per quarter-note construct is unnecessary as there already exists a standard for specifying durations and locations in scores that does not require any such an arbitrary device. The concept of beats per quarter-note was probably inherited from the
MIDI file standard, where it was used as a convenient way for synchronizing time.

German/American time values, such as quarter notes and eight notes, are a considerable advance on the English/French/Italian naming of crotchets and quavers. For example a one-hundred-and-twenty eight note, or rational number 1/128 gives the duration directly, without having to count the number of prefixes using the alternate semi-hemi-demi-semiquaver. Also rational numeric durations give a way of readily checking that a measure has the correct length. For example, for a measure in 3/4 time containing a sequence of a minim followed by two quavers and four semiquavers, the note durations will to add to 3/4 = 1/4 + 2*(1/8) + 4*(1/16). In answering questions, starting and ending positions could be specified in rational measures. For example a span of a minim at the start of a bar might be recorded by means of its start point and end point as 0/4 and 1/4 without any further decisions and calculations needed as is in the case where bpqs are used.

4.4 Recommendations for a Line of Questioning

The rules of either strict or free counterpoint are well documented. These might provide material for constructing future questions. In a future iteration of C@merata, some questions relating to species counterpoint might be considered, starting with first species counterpoint. Success in this endeavor, especially if the five counterpoint species are eventually explored might lead to the development of useful tools for composition students.

5. REFERENCES


