SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2003; **33**:173–200 (DOI: 10.1002/spe.502)

A music notation construction engine for optical music recognition

David Bainbridge[1] and Tim Bell[2,*,†]

[1]*Department of Computer Science, University of Waikato, Private Bag 3105, Hamilton, New Zealand*
[2]*Department of Computer Science, University of Canterbury, Private Bag 4800, Christchurch, New Zealand*

SUMMARY

Optical music recognition (OMR) systems are used to convert music scanned from paper into a format suitable for playing or editing on a computer. These systems generally have two phases: recognizing the graphical symbols (such as note-heads and lines) and determining the musical meaning and relationships of the symbols (such as the pitch and rhythm of the notes). In this paper we explore the second phase and give a two-step approach that admits an economical representation of the parsing rules for the system. The approach is flexible and allows the system to be extended to new notations with little effort—the current system can parse common music notation, Sacred Harp notation and plainsong. It is based on a string grammar and a customizable graph that specifies relationships between musical objects. We observe that this graph can be related to *printing* as well as *recognizing* music notation, bringing the opportunity for cross-fertilization between the two areas of research.

KEY WORDS: optical music recognition; music notation construction; definite clause grammars; graph traversal

INTRODUCTION

The development of software that converts digitally scanned pages of music into a computer-pliable symbolic form spans many areas of computer science: from image processing to graph representation; from pattern recognition to knowledge representation; from probabilistic encodings to error detection and correction. Literature on the topic, known as optical music recognition (OMR), is sizeable and steadily growing. There have been at least eight PhD theses since the inception of OMR in the mid-1960s [1–8] and numerous other projects—see [9] or [10] for a review. Commercial products have also been available since 1993.

*Correspondence to: Tim Bell, Department of Computer Science, University of Canterbury, Private Bag 4800, Christchurch, New Zealand.
†E-mail: tim@cosc.canterbury.ac.nz

Received 18 February 2000
Revised 13 June 2002
Accepted 8 October 2002

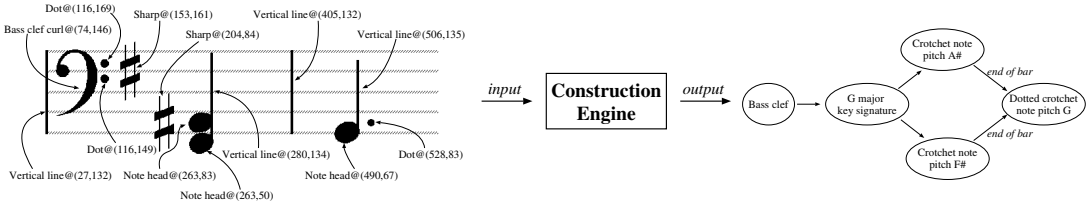


Figure 1. The role of the music notation construction engine.

Software systems that tackle the OMR problem are largely composed of two parts: an image processing 'front end' that detects the musical primitives (the horizontal staff lines, note-heads, note stems and so forth) present in a scanned image; and a music notation 'construction engine' that builds a symbolic representation of the music based on the detected musical primitives[‡].

There is little variation among the image processing strategies employed by OMR projects. All start by locating the staff lines[§] (which helps to establish the size of the music notation) before progressing to the detection of the musical primitives using well-established pattern recognition techniques—a mixture of template matching, bounding box comparisons, projection methods, the Hough transform, nearest-neighbour classification and neural-network classification schemes being the most popular.

By way of contrast, there is much diversity in the design of the construction engine. The focus of this article, therefore, is to describe—in the context of existing work—our experiences of writing software in this area.

Figure 1 illustrates the role of the construction engine. It takes as its input the detected graphical primitives and, through the application of musical knowledge, produces a graph as an output that symbolically represents the music score. Even in this simple example, the construction engine has much to do and the interplay between shapes can be subtle. In the instance of the first note (actually two notes played simultaneously), a vertical line and two note-heads must be grouped logically together and two nodes in the output graph formed; for the second note it is a vertical line, a note-head and a dot that must be grouped together, and this time only one node is added to the output graph. In the meantime, the vertical line between these two notes must be correctly identified as a bar line (by the lack of note-heads in the vicinity) and the appropriate construction formed in the output graph (shown as labelled arcs in the graph). Vertical lines and other primitive symbols can have overloaded meanings and the engine must determine the correct meaning. For example, a dot might be part of a bass clef (as at the left of Figure 1), or it might be used to extend the duration of a note (as on the rightmost note of Figure 1).

Further intricacies follow from the processing of the remaining input primitives. Once the two dots to the left of the score have been grouped with the primitive labelled 'bass clef curl' to form a bass

[‡]This division is also referred to as low-level and high-level recognition in some literature.

[§]The *staff* or *stave* refers to the groups of (usually) five horizontal lines on which notation is placed.

clef, its effect has to be mapped through the graph, establishing the pitches of all the notes. That done, it is the turn of the accidentals to affect pitch. The first sharp forms a G key signature; the second is an 'accidental', affecting only those notes that follow it in the bar in which it occurs. The key signature, therefore, must be processed first, modifying the pitch of any F note before the localized effect of the accidental can be safely added. Note that the distinction between the two sharps in Figure 1 is subtle, yet given the rules of music notation, it is unambiguous. The distance between the two shapes, the particular pitches of the sharps and the proximity of the note-head all contribute to provide a clear indication of their different roles. Finally, in the absence of a time signature, a metre of $\frac{4}{4}$ is implicitly assumed and consequently the double note has a duration of 1 beat and the following note 1.5 beats.

From these examples we can see that the role of each symbol depends on its context and given sufficient contextual information, it is usually unambiguous. It is this observation that led us to use grammar as the basis of the music recognition engine described in this paper.

A large part of the problem is knowing the order in which to process information. Unlike conventional optical character recognition (OCR) systems, OMR must deal with notation that represents two-dimensional 'events'. Generally, the two dimensions are for rhythm and pitch, but music notation is not as simple as this. For example, the position of text in music distinguishes lyrics from dynamic markings and melodic motifs may have a stronger correlation horizontally than vertically. Thus a digital representation of music notation must be suitable for a two-dimensional world in which many different relationships can exist.

EXISTING WORK

Early work, juggling many other constraints, favoured customized code for the construction engine. However, more recent and more successful work has tapped into artificial intelligence (AI) and parsing literature to provide a more flexible solution.

Kato and Inokuchi are the first known researchers to use a sophisticated AI-based paradigm to solve the problem [11]. Using a shared memory model to communicate, different layers of the system produce hypotheses that other layers must verify or refute. In making its decision, a contacted layer may itself spawn hypotheses and so the process continues until all hypotheses are answered. Control is through a top-level 'goal' layer. The software was specifically designed to process piano music and consequently can be seen as a mixture of the general model and customized code.

Coüasnon *et al.* take a different approach, basing their work around a parser [7,12,13]. Their grammar rules specify how the primitives are to be processed, what makes a valid musical event and even how to segment graphical shapes. To counter the extra complexities added in trying to parse two-dimensional data (the graphically detected primitives), the researchers augmented the grammar syntax with position and factorization operations and implemented the parser in λProlog, a higher dialect of Prolog with more expressive power, but even then they reported difficulty in the task [13].

Fahmy *et al.* have concentrated their efforts on a graph grammar approach [14–16], a technique used successfully in other structured document image analysis problems to counter the complexities of two-dimensional data [17]. An extension of string grammars, graph grammars replace the one-dimensional string data structure (used, for example, to parse computer languages) with a graph-based data structure. The new data structure is better suited to handle the two-dimensional nature of the musical primitives; however, as in Coüasnon's work, this complicates the development of a parser.

Two other projects have continued the graph grammar theme [18,19]. Their work has been directed towards simplifying the task, with the use of *a priori* knowledge, to ease the task of writing the parser.

Set in the context of this existing work, the remainder of this article describes our experiences in developing this same component of an OMR system. The work brings two key developments. First, the problem is decomposed into two stages that reduce the level of code complexity experienced by other researchers. Second, the decomposition heightens the similarity of the latter stage with certain tasks performed by music editors and, consequently, experience can be drawn from the developers of these programs [20]. This second point becomes clearer once a detailed description of the construction engine has been given; consequently we defer the discussion of this until the end of the article.

The structure of the article is as follows. First we give a general overview of our software architecture. Through necessity this includes a (brief) description of our image processing front end, but the main attention is given to the construction engine. A detailed description follows of the two stages that make up our construction engine: primitive assembly and musical semantics. We conclude with a summary of our findings and experience.

OVERVIEW OF SOFTWARE ARCHITECTURE

Our OMR system, known as CANTOR, was originally developed as part of the principal author's PhD. In the intervening years since then, the software has matured through its use in various projects [21–25]. The overarching aim of the work has been to produce an OMR system that is fully extensible so that quite different music notations can be recognized with minimal effort in configuring the system. The problem has been divided into four stages: staff detection, primitive detection, primitive assembly and musical semantics, each stage being implemented in as general a way as possible. Together the primitive assembly and musical semantic stages form a construction engine.

Figure 2 gives an overview of the system for the three different types of music notation our software is capable of processing: common music notation (CMN), Sacred Harp notation and plainsong notation. Rudimentary examples of music notation are used for most of this paper to aid clarity in the descriptions. Figure 3 gives a more representative CMN example of what CANTOR is intended to work with and Figure 4 shows the result of recognizing it using the system described in this paper.

While the result strongly resembles the original, several mistakes have been made—for example, a natural and three ties are missing; more subtly the pitch of the first C note in bar 4 is off by one and, of the initial note-heads in the third bar of the second line, the lower one is incorrectly associated with the beamed structure rather than being a crotchet (quarter) note in its own right. Of the three ties, the first two are missing due to failed pattern recognition, whereas the final tie was correctly recognized graphically, but omitted due to processing by the musical sematics stage. Because the example image processed was an artificially cropped excerpt the music notation ends somewhat abruptly, leaving a tie that 'goes nowhere'. The musical semantic processing detected this and consequently removed the object. To faithfully reconstruct the score, these errors must be corrected using a music editor.

Returning to Figure 2, the staff detection stage locates and removes staff lines to make it easier for subsequent stages to locate and process the musical features. The algorithm deals with staves consisting of one or more staff lines, different staff sizes within the same image and staff systems that are placed side by side, as commonly occurs in a score with a coda section. To cope with the variety of different shapes in music notation, a specially designed programming language, called PRIMELA,

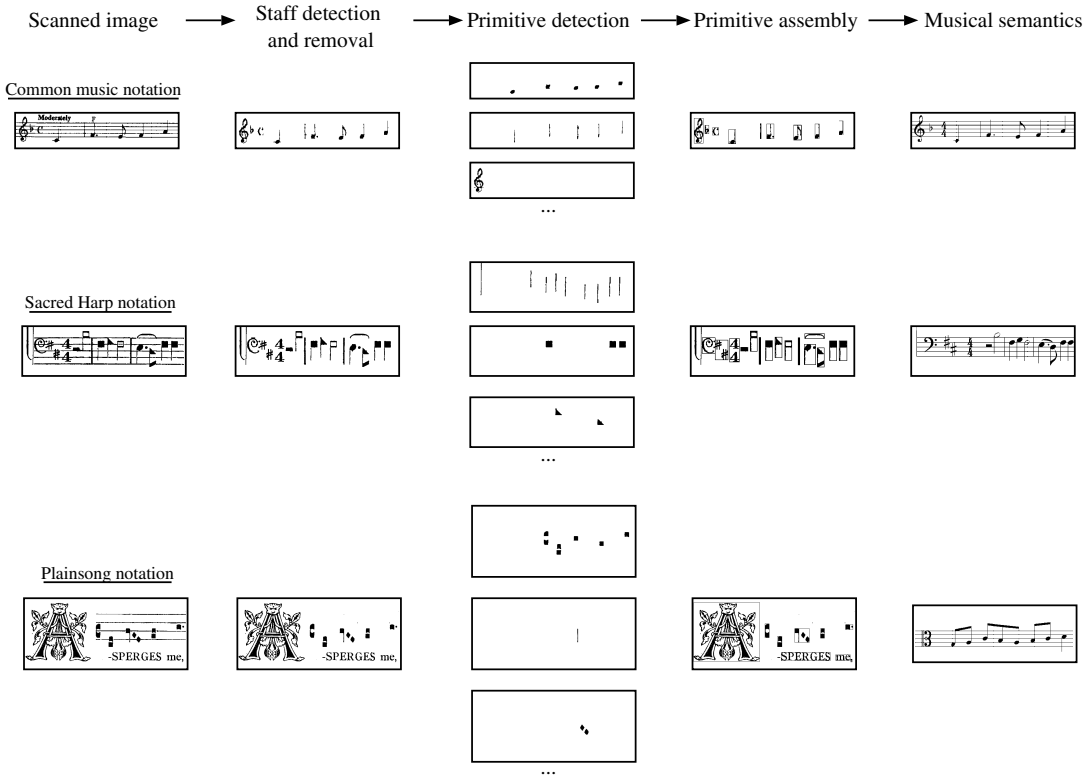


Figure 2. Overview of the principle stages in CANTOR.

was developed to detect musical primitives [26]. In Figure 2 three sample primitives are shown for each type of notation—for CMN they are the filled-in note-head, vertical line and treble clef. Using this language, new graphical shapes can be added with ease to the list of musical primitives processed. For the configuration used to process Figure 3, the average number of lines per primitive description was 56.

The assembly of primitives into musical features is accomplished using a knowledge base of music notation taxonomy expressed as grammar rules. The grammar rules are straightforward to write—Figure 5 shows a simplified example for a rudimentary type of chord[¶] in CMN. Figure 6 shows examples of notes that conform to this rudimentary description.

[¶]In music, a chord should actually have three or more notes, but for our purposes the term is used for any number of notes on one stem.

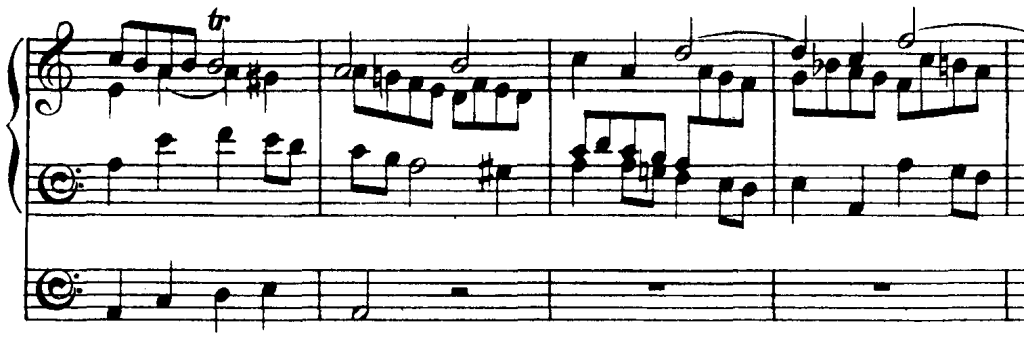


Figure 3. A more representative CMN excerpt for CANTOR ('Passacaglia' by J. S. Bach).

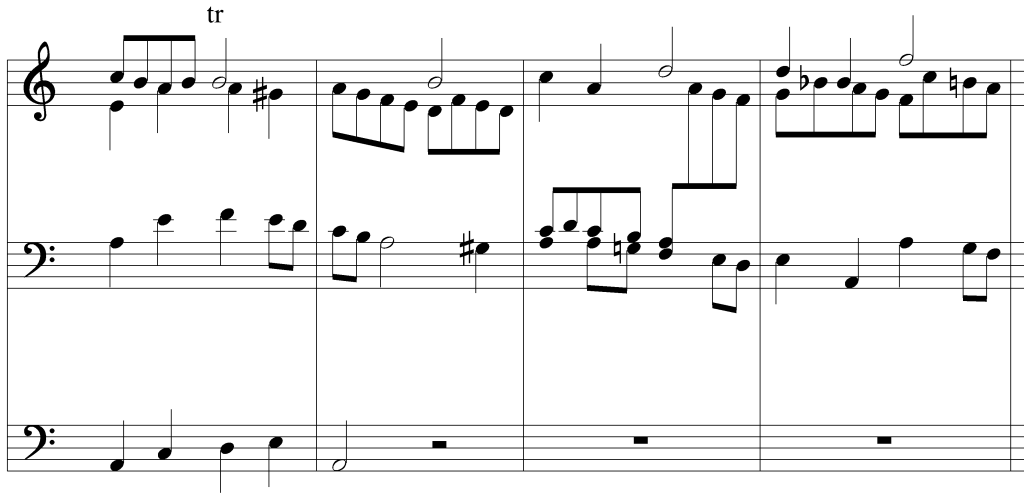


Figure 4. Reconstructed excerpt of 'Passacaglia' produced by CANTOR.

Closely resembling a Prolog style of syntax, tokens to the language appear in square brackets `[ . . . ]` and represent the musical primitives detected in the image; embedded constraints appear in curly brackets { . . . } and in the figure have been replaced with comments describing the type of constraint applied (`%  . . .`); the remaining constructs are non-terminals in the grammar, to be expanded further during parsing. The figure is discussed in more detail in the section on primitive assembly.

The parser is written in Prolog, the intentional similarity in grammar style easing development. The parser is small and simple to write, constituting 220 lines of code. The end result of parsing the musical primitives is a collection of annotated derivation trees, an example of which is shown in Figure 7. Oval nodes represent non-terminals and rectangular nodes represent musical primitives. This forms the input to the final stage in CANTOR: musical semantics.


```
simple_note_up
          ==> [(vertical_line,_,Sxl,Syt,Sxr,Syb,_)],
              opt_tails_up(Sxl,Syt,Syb,NoTails),
              note_heads_up(Sxl,Syt,Syb,NoTails).

note_heads_up(Sxl,Syt,Syb,NoHalves)
          ==> note_head_within_up(Sxl,Syt,Syb,NoHalves),
              note_heads_up(Sxl,Syt,Syb,NoHalves).
note_heads_up(Sxl,Syt,Syb,NoHalves)
          ==> note_head_within_up(Sxl,Syt,Syb,NoHalves).

note_head_within_up(Sxl,Syt,Syb,NoHalves)
          ==> note_head(Syt,Syb,NoHalves,Nxl,Nyt,Nxr,Nyb),
              { % note head must be close to stem (to the left) }.

note_head(Syt,Syb,NoHalves)
          ==> [(full_note_head,_,Nxl,Nyt,Nxr,Nyb,_)],
              { % retrieve appropriate StaffGap for note head },
                opt_dur_dots(Nxr,StaffGap,Nyt,Nyb,NoDots),
              { % store note head duration etc }.

opt_dur_dots(Nxr,StaffGap,Nyt,Nyb,NoDots)
          ==> opt_dur_dots(Nxr,StaffGap,Nyt,Nyb,0,NoDots).
opt_dur_dots(Nxr,StaffGap,Nyt,Nyb,NoDotsIn,NoDotsOut)
          ==> [(dot,_,Dxl,Dyt,Dxr,Dyb,_)],
              { % check dot close enough to note head },
              { IncNoDots is NoDotsIn + 1 },
                opt_dur_dots(Dxr,StaffGap,Nyt,Nyb,IncNoDots,NoDotsOut).
opt_dur_dots(_,_,_,_,NoDots,NoDots)
          ==> []. % epsilon

% opt_tails_up is similar to opt_dur_dots
```

Figure 5. Edited BCG production rules that describe simple stem-up notes consisting of note-heads, a stem, dots and tails.

Figure 6. Examples of simple stem-up notes.

The task for the musical semantics stage is to map out the two-dimensional effects of the assembled musical features. For example, a clef affects the register of all notes on its staff, an accidental only within its bar; ties affect the start and end of two notes, and so on. In CANTOR, a basic graph is pre-constructed according to a simple but well-defined criterion—each musical feature's (x, y) position. The result is a lattice-like structure of musical feature nodes that are linked horizontally and vertically (see Figure 16 later for an example). The effect of musical features is implemented by

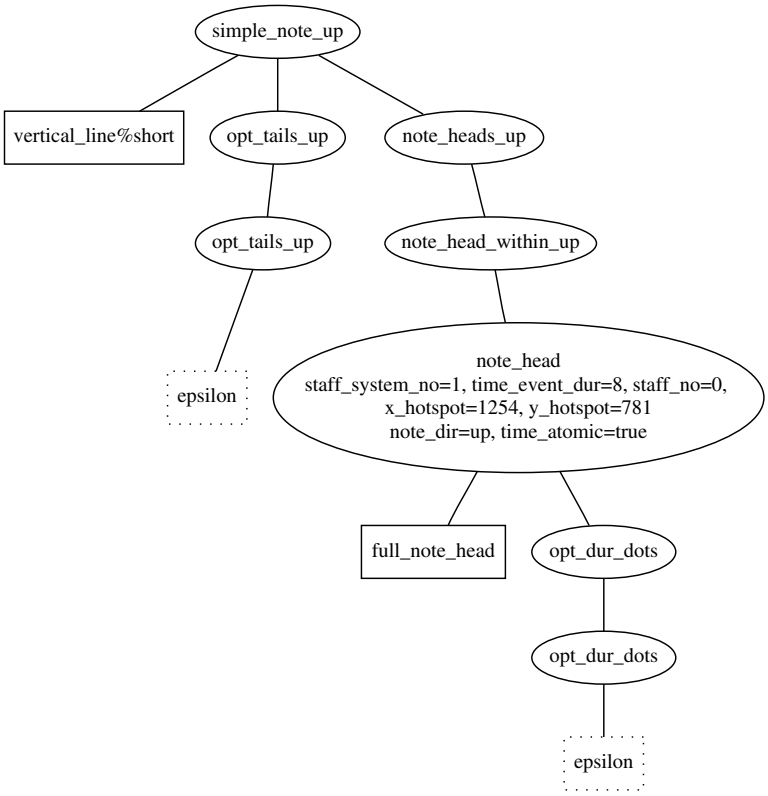


Figure 7. The annotated derivation tree for a crotchet note.

writing graph traversal routines in C++ which are dynamically linked into the main software at run-time. Object-oriented inheritance is used to simplify the task.

In terms of the earlier description of OMR systems, the first two stages of CANTOR form the image processing front end and the last two form the construction engine. Below we discuss in detail the design and implementation of this latter component of the CANTOR system, but first we discuss the issue of robustness.

Robustness of design

To increase the robustness of CANTOR, multiple passes within a stage are used to refine decisions. For instance, initially establishing where the staff lines are is based on an estimated value for staff line thickness calculated by an analysis of the image. It is a simple matter to arrange a second pass that substitutes the actual staff line thickness detected for the estimate, thereby improving the success rate of this stage.

Another technique is to consider multiple solutions, as exemplified by the primitive detection and primitive assembly stages. The key ingredient in terms of software design is to record decisions with a rating—expressed as a value between 0.0 and 1.0—representing how certain the algorithm is about its decision; the higher the value the more certain it is.

When a primitive shape is detected by the system, a certainty rating is assigned to it. Different primitive shapes, through their PRIMELA descriptions, can calculate this in different ways, although most come from the pattern recognition techniques used. In template matching or the Hough transform, for example, two thresholds are specified: *definite* and *possible*. A match above the definite threshold is assigned a certainty value of 1. A match between the possible and definite thresholds is scaled to the range 0.0 to 1.0. For matches above the definite threshold it is common to remove the detected shape from the image, while possible matches are left in the image so that they may match (perhaps with a higher degree of certainty) a subsequent PRIMELA description. In fact, like the decision on how to set the certainty rating, the decision to remove a shape is under the control of individual descriptions.

During the primitive assembly phase, the certainty ratings that were calculated during primitive detection are used to bias the decision to more likely outcomes. In a situation where more than one musical feature can be assembled (parsed) from a comparable set of primitives, the system chooses the one with the highest total score of certainty values. For example, it is not uncommon for minim note-heads—through a variety of causes, both in the original work and due to processing—to be distorted such that some interior pixels are set to black. This in turn can trigger a possible match with the full note-head description, although its certainty rating (unless an extreme example) is lower than the hollow note-head description. Both solutions are allowed to exist within the system. When the primitive assembly phase processes these shapes, the assembled feature that uses the minim note-head is favoured because the total certainty of the assembly shape is higher than the version that used the full note-head.

PRIMITIVE ASSEMBLY

To accomplish primitive assembly, the essential task is that of expressing a valid taxonomy of musical features. By this we mean how the detected primitive shapes, such as filled-in note-heads, vertical lines, and sharps are assembled into musical features such as notes and key signatures. For an extensible system such as CANTOR, some form of knowledge representation is required.

Existing work has favoured grammar-based approaches albeit in substantially different forms: Coüasnon *et al.* [7,12,13] augmented a string grammar with position and factorization operators; Fahmy *et al.* [14–16] replaced the one-dimensional string-based grammar with a graph grammar methodology. Unfortunately, as mentioned above, these extensions also complicate the task of writing a suitable parser.

On closer inspection there are two factors driving these extensions: the need to cope with the added complexity of parsing two-dimensional data and also the fact that their knowledge representation components are used to perform tasks in addition to primitive assembly. By restricting the scope of the knowledge representation task solely to primitive assembly there is perhaps an easier way to accomplish this. Of course, the other parts included in Coüasnon *et al.*'s and Fahmy *et al.*'s work must be implemented elsewhere in the system, but we have found a standard procedural programming language (C++ in our case) well suited to the task.

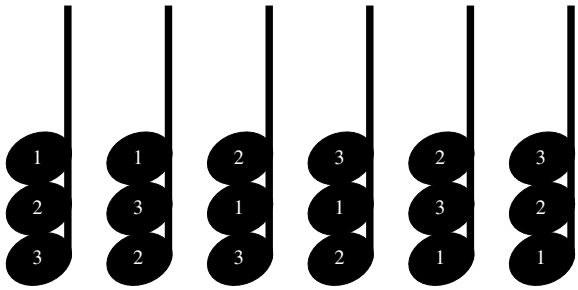


Figure 8. There is more than one way to assemble the note-heads in a chord.

During our research [5] we considered the strengths and weaknesses of three knowledge representation schemes: production rules, semantic networks and frames [27]. In this context, string grammars are merely a syntactic encoding of production rules. All three methods were found to be amenable to the task; however, we noticed that a subtle but significant change to a string-based grammar scheme known as definite clause grammars (DCGs) [28] enabled us to achieve everything we wanted.

Our insight was to change the *list* of tokens traditionally used in a DCG to a *bag* of tokens. Because a bag is also a one-dimensional data structure, the development of a parser is kept simple.

The bag data structure is a cross between a list and a set. For a bag, the order of objects does not matter and the same element can appear more than once. This accommodates common musical symbols, such as a chord with an arbitrary number of note-heads or a key signature with an arbitrary number of sharps or flats.

A bag data structure is straightforward to implement in Prolog as a predicate that extracts elements from a list, with unrestricted backtracking. This modified form of DCG notation was therefore selected as the implementation medium for our grammar-based knowledge representation of primitive assembly.

In terms of run-time complexity, the introduction of a bag data structure is potentially dangerous, since a grammar can now specify combinatorially large search spaces. For example, if a grammar based on a bag of primitives does not specify the order in which note-heads next to a stem are assembled, then for every chord in a piece of music with n note-heads, there are $n!$ combinations of note-heads that satisfy the grammar. This point is illustrated in Figure 8, where six different orderings of three note-heads assemble to form the same crotchet chord. For this example, evaluating all the combinations would be wasted anyway, since the displayed form is identical in all cases.

Fortunately such situations can be avoided by embedding constraints and controlling backtracking in the grammar. For instance, in the example of Figure 8, an improvement (better computationally) is to specify a stack of note-heads that form a chord as a sequence of note-heads constrained to align vertically (with a certain degree of tolerance in the x-axis), where there is no backtracking once a note-head has been found to fall within the geometrically constrained catchment area. Given this configuration, there is only one way to parse a chord with three note-heads like the one shown in Figure 8, but the actual outcome (which of the six possible assemblies) depends on the order in which the primitives are taken from the bag.

Table I. Comparison of the original DCG implementation with the modified BCG implementation.

Parser	Number of lines	Number of characters
Definite clause grammar	120	3719
Bagged clause grammar	216	10 402

Implementation

An existing implementation of a DCG parser (in Prolog) was adapted to use bags rather than lists. Statistics on the size of the original DCG parser and the modified parser (dubbed a bagged clause grammar, or BCG) are shown in Table I.

The extension approximately doubles the length of the implementation. Most of the changes are to allow predicates to carry extra arguments (such as `PageInfo`, used to carry general pre-processed information about the scanned page) and of the 96 extra lines added, 31 lines (2087 characters) support predicates specific to our primitive assembly task.

Extra predicates added are as follows.

- `page_info(LinkName)`. This predicate forms a link between the grammar and pre-processed information about the scanned page. Using this link a production rule can access, for example, scan resolution, staff system information (local or global), staff information (local or global), staff line thickness (local or global) and so on.
- `prim_present(Prim,Xl,Yt,Xr,Yb,CheckType)`. This predicate checks for the existence of the named primitive in the specified rectangle. There are three types of check: at least one `corner` of a primitive of the named type falls inside the specified rectangle; a primitive of the named type fits completely `inside` the specified rectangle; and a primitive of the named type and the specified rectangle `intersect`.
- `prim_not_present(Prim,Xl,Yt,Xr,Yb,CheckType)`. This predicate computes the negation of `prim_present`.
- `prim_present_static(Prim,Xl,Yt,Xr,Yb,CheckType)`. As the assembly process proceeds, the primitives used to assemble a particular instance of a musical feature are removed. Thus the bag of primitives used by the predicate `prim_present` changes dynamically. When checking for primitives, this is not always desirable. An alternative is to use the predicate `prim_present_static` which performs the same function, except a static snapshot of the bag at the very start of the assembly process is used.
- `prim_not_present_static(Prim,Xl,Yt,Xr,Yb,CheckType)`. This predicate computes the negation of `prim_present_static`.
- `store_attribute(AttName,AttValue)`. It is possible to annotate the derivation tree of an assembled musical feature. Using this predicate in a production rule stores the named field and value at the corresponding node in the derivation tree. Permissible types for attributes are integers, floating-point numbers and strings.

Examples

To illustrate the techniques used in primitive assembly using a BCG, let us now look at some examples taken from the grammar for CMN assembly.

Figure 6 shows example notes from a collection of musical features we will call simple stem-up notes. In Figure 5, BCG rules are given for the representation of such notes. The example has been simplified to focus attention on the structure of the rules. The first production rule (starting at line 1) picks a vertical line from the bag and looks for suitable tails and note-heads. The second production rule (starting at line 6) allows a vertical stack of note-heads to be associated with the stem. The `note_head_within_up` production (starting at line 12) checks to see if a selected note-head is close enough to the stem. The note-head to check is provided by the `note_head` production (starting at line 16) which also discovers any durational dots to the right of the note-head using `opt_dur_dots` (starting at line 22) and stores the duration of the note in the derivation tree. The production rule for tails (`opt_tails_up`) is identical in structure to `opt_dur_dots`, except a (possibly non-existent) vertical stack of tails is built, rather than a (possibly non-existent) horizontal line of dots.

In Figure 9 an excerpt of the simplified BCG of Figure 5 (lines 12–30) is shown in full detail.

Constraints that were explained in English in the simplified figure, are defined by Boolean expressions. For example, on line 3 the centre of the note-head picked from the bag is constrained to fall between the top and the bottom of the stem. If this condition is not met, then the note-head is discarded and another is chosen. Note also the use of the cut operator ('!') on line 8, which controls back-tracking. Once a note-head has been found that binds to a stem, there is no need to go back to try and start a vertical stack of note-heads with a different note-head. Thus the undesired combinatorial situation illustrated in Figure 8 is prevented.

Figure 9 also shows the use of supporting predicates specific to OMR. The `mid` and `dim` predicates respectively calculate the mid-point and the distance between two numbers. On line 12, the `page_info` predicate is used to form a link to the page specific data. This link is then used on line 16 to obtain the value of the staff gap resulting from the largest staff that can place musical features on the pixel line in the image specified by `Nym`. Lines 24–30 store information in the derivation tree for future use by the musical semantics stage of CANTOR—how these values are used is explained in the next section. The derivation tree from a crotchet note is shown in Figure 7.

To see an example of the `prim_present` predicate, let us study the BCG production rule for a sharp, which is shown in Figure 10. Since a sharp can form part of a key signature (at the beginning of a line) or appear as an 'accidental' in front of a note, it is necessary to distinguish between the two situations. This is accomplished by lines 18–23, which confirm that the sharp under consideration appears in isolation by checking to see if there is a note-head immediately to the right of the sharp. The static version of the predicate is used because the assembly of notes occurs before accidentals and therefore note-heads are no longer in the dynamically changing bag.

A similar use of `prim_not_present_static` is made in the production rule for a single bar line, shown in Figure 11. For a vertical line to be 'assembled' into a bar line, its length must be at least 80% of the minimum staff height operating in the area of the vertical line (line 6), it must intersect a staff (line 9) and the top of the vertical line must be located close to the top of the staff it intersects with (lines 12–17). Alone, however, this is an insufficient specification for a single bar line, since it is conceivable that a note might be drawn in such a way that its stem is long enough to be a bar line, and the top of the note stem stops near the top of the staff. The discrepancy is resolved by using


```
note_head_within_up(Sxl,Syt,Syb,NoHalves)
          ==> note_head(Syt,Syb,NoHalves,"up",Nxl,Nyt,Nxr,Nyb),
              { mid(Nyt,Nyb,Nym), (Nym>=Syt) and  (Nym=<Syb), % within
                dim(Nxl,Nxr,Nxd), mid(Nxl,Nxr,Nxm),
                Wxr is Sxl, Wxl is Sxl - (Nxd//4),
                % note head must be close to stem (to the left)
                (Nxr>=Wxl) and (Nxm=<Wxr) },
              !. % committed choice


note_head(Syt,Syb,NoHalves,NoteDir)
          ==> [(full_note_head,_,Nxl,Nyt,Nxr,Nyb,_)], page_info(PageInfo),

              % retrieve appropriate StaffGap for note head
              { mid(Nyt,Nyb,Nym), mid(Nxl,Nxr,Nxm),
                max_gap_height(PageInfo,Nym,StaffGap) },

              opt_dur_dots(Nxr,StaffGap,Nyt,Nyb,NoDots),
              { cmn_time_unit_duration(8,4,NoHalves,NoDots,NoteHeadDur) },
              { pi_staff_systems(PageInfo,StaffSystems),
                closest_staff(StaffSystems,Nxm,Nym,StaffSystemNo,StaffNo) },

              % store note head duration etc
              store_attribute("time_atomic","true"),
              store_attribute("time_event_dur",NoteHeadDur),
              store_attribute("note_dir",NoteDir),
              store_attribute("staff_system_no",StaffSystemNo),
              store_attribute("staff_no",StaffNo),
              store_attribute("x_hotspot",Nxm),
              store_attribute("y_hotspot",Nym).


opt_dur_dots(Nxr,StaffGap,Nyt,Nyb,NoDots)
          ==> opt_dur_dots(Nxr,StaffGap,Nyt,Nyb,0,NoDots).

opt_dur_dots(Nxr,StaffGap,Nyt,Nyb,NoDotsIn,NoDotsOut)
          ==> [(dot,_,Dxl,Dyt,Dxr,Dyb,_)],

              % check dot close enough to note head
              { (Dxl>Nxr) and (Dxl<Nxr+StaffGap), % within
                mid(Dyt,Dyb,Dym), HalfStaffGap is StaffGap // 2,
                (Dym>Nyt-HalfStaffGap) and (Dym<Nyb+HalfStaffGap) },

              { IncNoDots is NoDotsIn + 1 },
              opt_dur_dots(Dxr,StaffGap,Nyt,Nyb,IncNoDots,NoDotsOut).

opt_dur_dots(_,_,_,_,NoDots,NoDots)
          ==> []. % epsilon
```

Figure 9. Unedited BCG production rules for simple stem-up notes.


```
 1  sharp
 2     ==> [(sharp,_,Sxl,Syt,Sxr,Syb,_)],  page_info(PageInfo),
 3         { pi_staff_systems(PageInfo,StaffSystems),
 4           intersects_staff(StaffSystems, (Sxl,Syt,Sxr,Syb), StaffSystemNo,
 5                            StaffNo, StaffRect,StaffInfo),
 6           staff_info_staff_height(StaffInfo,StaffHeight),
 7           SharpGap is StaffHeight // 2,
 8           Cxl is Sxr, Cxr is Sxr + SharpGap, Cyt is Syt, Cyb is Syb },
 9         note_head_present(Cxl,Cyt,Cxr,Cyb),
10         { mid(Sxl,Sxr,XHotSpot), mid(Syt,Syb,YHotSpot) },
11         store_attribute("secondary_atomic","true"),
12         store_attribute("acc_type","sharp"),
13         store_attribute("staff_system_no",StaffSystemNo),
14         store_attribute("staff_no",StaffNo),
15         store_attribute("x_hotspot",XHotSpot),
16         store_attribute("y_hotspot",YHotSpot).
17
18  note_head_present(Cxl,Cyt,Cxr,Cyb)
19     ==> prim_present_static(full_note_head,Cxl,Cyt,Cxr,Cyb,intersect).
20  note_head_present(Cxl,Cyt,Cxr,Cyb)
21         prim_present_static(hollow_note_head,Cxl,Cyt,Cxr,Cyb,intersect).
22  note_head_present(Cxl,Cyt,Cxr,Cyb)
23         prim_present_static(semibreve,Cxl,Cyt,Cxr,Cyb,intersect).
```

Figure 10. The BCG production rule for a sharp.

`prim_not_present_static` to rule out potential bar lines that have note-heads too close to the vertical line (line 20–24).

As a final example, an informal explanation of the production rules for beamed notes[‖] is given since the coding is prohibitively long for presentation in this article.

A beamed note can be a complex, nested structure, such as the example shown in Figure 12(a). Writing production rules to represent this assembly is, perhaps, a daunting prospect. However, if tipped on its side, a beamed note—such as the example shown in Figure 12(b)—can been seen as an abstract version of a block-nested programming language: beams correspond to nested blocks and stems correspond to statements. Developing grammar rules for a block-nested programming language is a familiar task—often included in a compiler assignment for an undergraduate course. The task of assembling beamed notes is no different, with the same pattern of production rules forming the main structure to this part of the grammar.

The analogy is not exact, the main difference being the structure of a beamed note permits stems to be placed on both sides of the primary beam. Fortunately, this is a superficial difference that can be encoded by developing two sets of rules in parallel that mirror one another in the structures they describe.

[‖] Beams are the solid lines used to join the stems of adjacent notes.

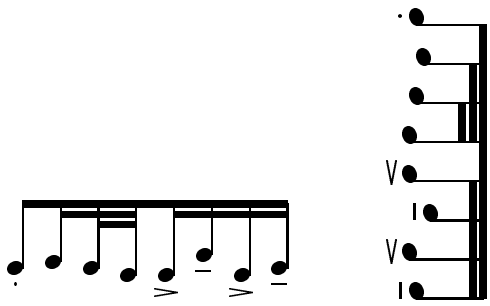
```
single_bar_line
   ==> [(vertical_line,_,Bxl,Byt,Bxr,Byb,_)], page_info(PageInfo),
       { dim(Byt,Byb,Byd), mid(Byt,Byb,Bym),
         min_staff_height(PageInfo,Bym,StaffHeight),
         % bar line 80% of staff height
         Byd > (80 * StaffHeight // 100)},
       { pi_staff_systems(PageInfo,StaffSystems),
         % bar line intersects staff
         intersects_staff(StaffSystems,(Bxl,Byt,Bxr,Byb),StaffSystemNo,StaffNo, _,_),

         % top of vertical line start at same height of staff
         retrieve_staff_from_staff_systems(StaffSystems,StaffSystemNo,StaffNo,Staff),
         staff_ytop(Staff,Syt),
         staff_staff_info(Staff,StaffInfo),
         staff_info_gap_height(StaffInfo,StaffGap),
         BarGapTol is StaffGap // 2,
         (Byt >= (Syt - BarGapTol)) and (Byt =< (Syt + BarGapTol)) }, % within

        % no note heads close by
        { BarNotPresTol is StaffGap,
          NPxl is Bxl - BarNotPresTol, NPxr is Bxr + BarNotPresTol,
          NPyt is Byt - BarNotPresTol, NPyb is Byb + BarNotPresTol },
       prim_not_present_static(full_note_head,NPxl,NPyt,NPxr,NPyb,intersect),
       prim_not_present_static(hollow_note_head,NPxl,NPyt,NPxr,NPyb,intersect),

       store_attribute("measure_atomic","true"),
       store_attribute("staff_system_no",StaffSystemNo).
```

Figure 11. The BCG production rule for a single bar line.



(a) (b)

Figure 12. A different way of looking at beamed notes: (a) normal; (b) side-on.

MUSICAL SEMANTICS

The second stage of our music notation construction engine is to determine the musical semantics. The main requirement of this stage is to combine the graphically recognized musical features with the staff systems to produce a musical data structure representing the meaning of the scanned image. This is accomplished by interpreting the spatial relationships found in the image. In applications such as OCR this is a simple (almost trivial) step since the layout is predominantly one dimensional. For music, however, the layout is much more complex. Positional information is extremely important. The same graphical shape can mean different things in different situations. For instance, to determine if a curved line between two notes is a slur or a tie, the pitch of the two notes must be considered. Also, establishing the association of 'free-floating' objects, such as a hairpin crescendo, with the appropriate notes is an important task in understanding music.

Our design is centred around two constructs, which we have called *time-threads* and *structured lists*. Time-threads form a two-dimensional lattice-like structure, linking all musical events together, where a musical event is an indivisible entity within the context of musical semantics, such as a note-head, rest or key signature. Threads run horizontally and vertically. For CMN, this equates to linking musical events sequentially and concurrently. A structured list is built on top of each horizontal thread and provides multiple *levels* for traversal. As a brief explanation, when traversed at one level, all the musical events for a staff are grouped together and can therefore be covered in one pass. This is a convenient level at which to apply the effects of global musical events such as clefs and key signatures to notes. Alternatively, when traversed at a different level, the same events are grouped together as individual bars, which is a convenient level at which to apply more local effects such as accidentals on notes.

Underlying these two data structures is a pre-constructed lattice-like graph built automatically by the software, based on the (x, y) coordinates of musical events. To meet the requirement of extensibility, the traversal and modification of the pre-constructed graph are programmable components of the system, implemented in C++ and dynamically linked into the system at run-time. This way different traversal routines can be written for different types of notation.

In CMN, for example, the effect of a key signature can be applied by supplying a routine that traverses the part of the graph representing the staff that the key signature is from, modifying the pitch of any note-head encountered whose pitch matches that of an accidental in the key signature. A subsequent routine applying the effect of accidentals on note-heads within a bar ensures the correct scoping of key signatures and accidentals. In plainsong notation a vertical column of notes on one staff is played sequentially from bottom to top. This differs from CMN where the same layout of note-heads signifies that the notes are played simultaneously; consequently the respective traversal routines process the note-heads differently.

Since there is, to date, no single recognized standard file format for music representation, we view the graph resulting from the application of the programmable component as the musical interpretation of the scanned image. Additional routines can be provided in the programmable component to traverse this structure, generating files conforming to particular file formats. We have routines for MIDI and CSound (for use in audio applications), and Tilia and NIFF (for use in music editor applications). These formats select a subset of the information that is available—audio applications are more focused on notes as events in time, while music editors require information about layout and auxiliary markings. For example, an audio application does not need to be concerned about cautionary accidentals or stem directions.

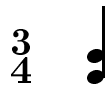


Figure 13. An example musical feature from each atomic category.

The pre-constructed graph

Based on the detected staff systems and assembled musical features, a skeleton graph is constructed at the beginning of the programmable musical semantics stage. A node in the graph represents a *musical event*—an entity that is indivisible with respect to its musical semantics. A musical event can be a primitive shape such as a note-head or a musical feature such as a bass clef. In fact, a musical event can be a partially assembled musical feature, although it is rare to find an example between the two extremes.

Assembled constructs that are musical events are specified during the primitive assembly phase. The annotated attribute field `time_atomic` or `secondary_atomic`, stored at a node of the derivation tree, defines the root of a sub-tree corresponding to the structure that is a musical event. The two categories exist to distinguish between musical events that directly involve duration (notes and rests) and those that do not—effectively all other musical events. Although markings such as *ritardando* and *presto* affect duration, they do so indirectly and are therefore classified as `secondary_atomic` events.

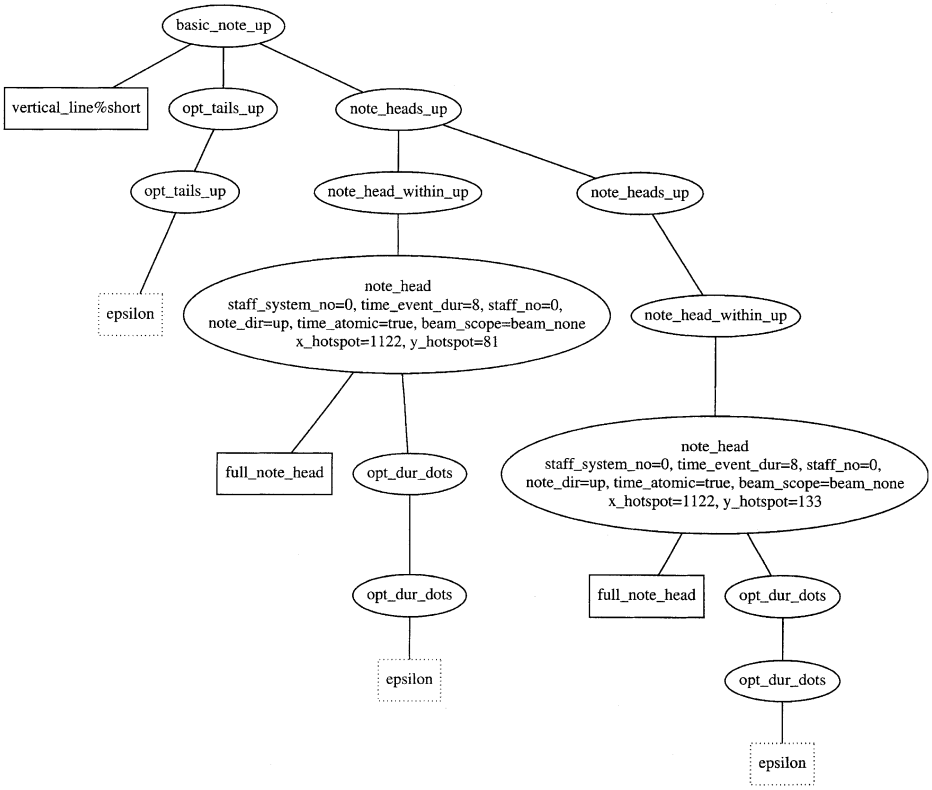
In Figure 13 an example of each atomic category is shown. The derivation tree for the crotchet dyad, shown in Figure 14(a), is annotated with `time_atomic` twice—once for each note-head—seen in the two large elliptical nodes of the tree, whereas the derivation tree for the $\frac{3}{4}$ time signature, shown in Figure 14(a), is annotated with a single `secondary_atomic` attribute at the top node, since this musical feature has no duration of its own.

Time threads

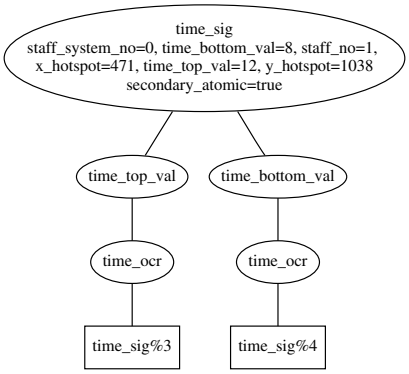
Nodes are constructed into a lattice-like graph structure, horizontally linking events pertaining to one staff together, as well as vertically linking musical events that occur both on the same staff and adjacent staves within the same staff system. These lines are termed *time-threads*.

Figure 15 shows an excerpt of music and Figure 16 shows the corresponding lattice, with musical events linked together both horizontally and vertically. The original scanned image is shown in grey and a dashed box has been drawn around each musical event. Only musical events that are designated `time_atomic` are linked vertically, thus the treble clef, bass clef and two key signatures shown in the example form only horizontal links. Also, to maintain uniformity in the lattice, an extra node is created if there is no `time_atomic` event on a staff at a given point, when other `time_atomic` events exist at the same vertical alignment on other staves within the same staff system. In Figure 16 the node marked 'Ta' beneath the crotchet rest is an example of such a node.

Every node in a derivation tree that specifies a musical event also stores a representative coordinate for its position on the page. It is this coordinate that determines the placement of the musical event



(a)



(b)

Figure 14. Musical events are defined by annotating nodes in the derivation tree: (a) `time_atomic`; (b) `secondary_atomic`.

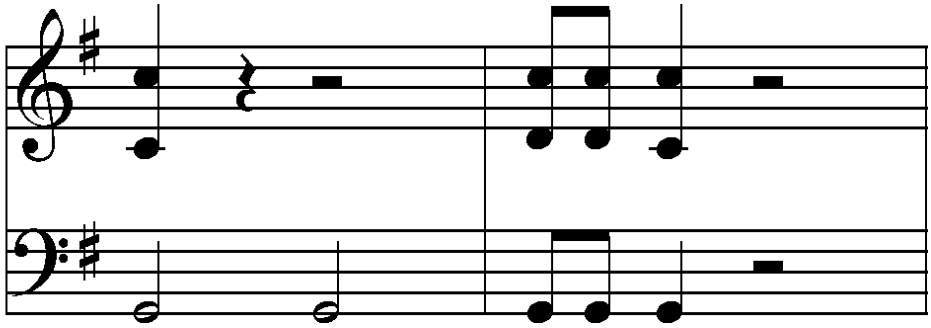


Figure 15. An excerpt of music before the lattice-like graph is constructed.

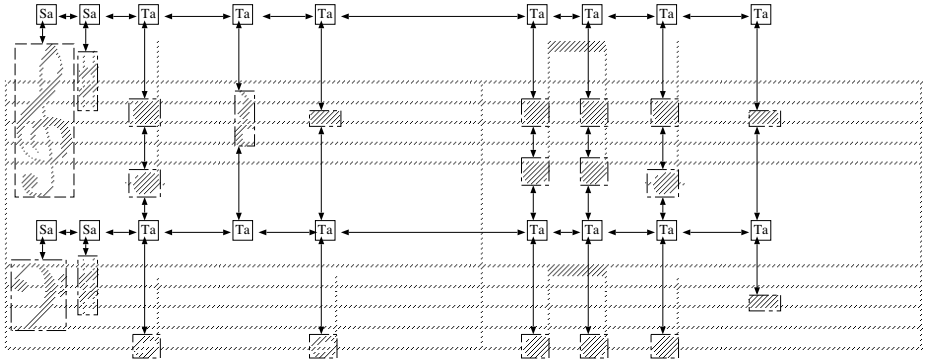


Figure 16. The excerpt of music after the musical events are constructed into a lattice-like graph, where links are referred to as time-threads. Ta = Time atomic, and Sa = Secondary atomic.

within the lattice. Borrowed from cursor terminology, the *hotspot* for an event is stored using the attribute names `x_hotspot` and `y_hotspot`. Examples of this can be seen in Figure 14. To form horizontal threads, all events belonging to a staff (lookup `staff_system_no` and `staff_no`) are sorted based on their `x_hotspot` value. Any events that share the same value are further sorted by comparing `y_hotspot` values. To form vertical threads, only `time_atomic` events are considered, with events being sorted based on their `y_hotspot` value. Tolerances—controlled by a parameter at run-time—exist to include events in the same thread if their `x_hotspot` values differ slightly.

Structured lists

Horizontal time-threads are further linked so they can be traversed by *page*, *staff system*, *staff*, *bar* or *voice*. The construct used is referred to as a *structured list*.

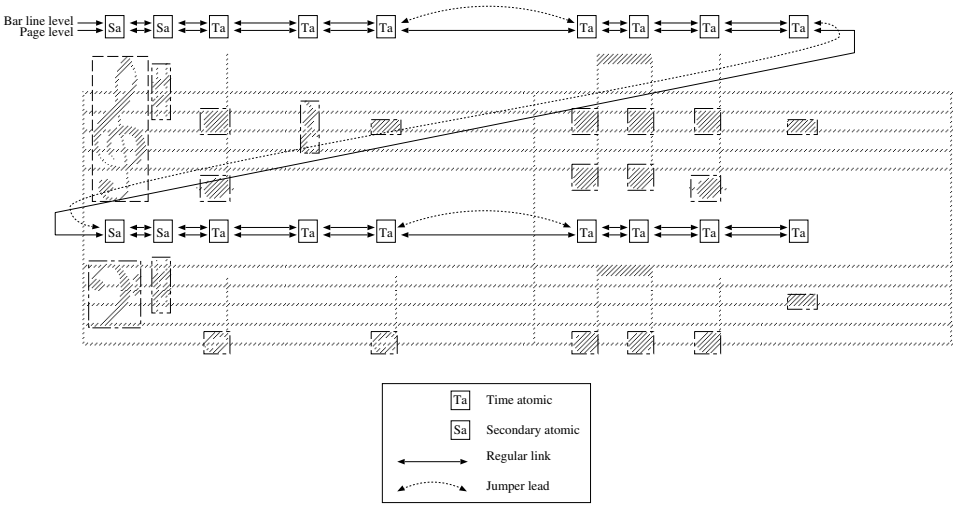


Figure 17. The example lattice with structured lists for bar and page built on top.

At the lowest level—known as the page level—all events are linked together. The start of the second staff follows on from the end of the first staff, likewise with the first and second staff systems and so on. At other levels in the structured list, musical events are grouped by some logical notion, such as staff or staff system. *Jumper leads* exist between adjacent groups of the same type to travel from one group to another. In the case of the staff level, a jumper lead connects the first staff to the second, the second to the third, and so on. The concept *voice* refers to a level in the structured list whereby the sequence of musical events corresponding to a single part can be followed. Hence the first staff in the first staff system is linked to the first staff in the second staff system and so on.

An example illustrating the page and bar levels of a structured list is shown in Figure 17. The page level never uses jumper leads. For the bar level, each time a bar line comes between two musical events in the lattice, a jumper lead is formed to bridge the gap. A wrap-around effect occurs at the end of staves, so a bar line at the end of a staff causes a jumper lead to connect the last event on one staff to the first event on the next staff. The lattice forming the time-threads have been omitted from the figure to simplify the diagram.

To build the graph it is assumed that information relating to the staff systems, staves and bar lines is available. The first two items result from the staff detection stage of CANTOR. For bar line information, we utilize the annotation mechanism used to communicate between the primitive assembly stage and the musical semantics stage. All musical features include the attribute `mf_group` in the root node of their derivation tree, specifying the musical feature group they belong to: clef, key signature, ornament and so on. During the construction of the graph, musical features marked as `bar` for this attribute are sought and used to form jumper leads for the bar level of the structured list, based on their values of `staff_system_no`, `staff_no`, `x_hotspot` and `y_hotspot`.

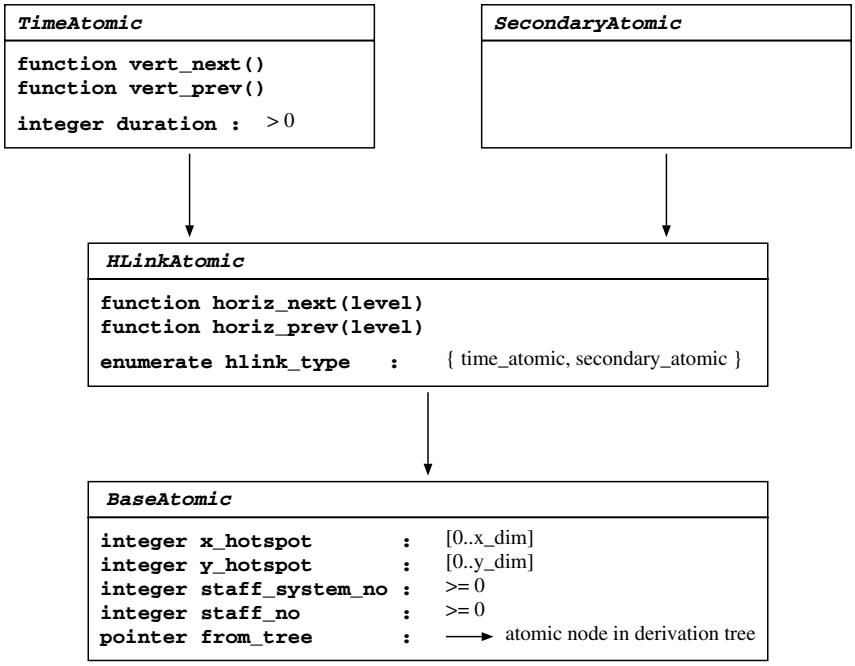


Figure 18. The system provides a minimalistic set of objects for graph construction.

Applying musical semantics

Once the graph has been built, the run-time specified code is called to complete the musical semantics, traversing and modifying the graph. To allow for extensibility and customization, the object-oriented language C++ was selected as a vehicle for implementation. Minimalistic objects for graph construction are provided by the system. Using the inheritance mechanism provided by the language, the programmer developing the run-time routines can add extra attributes to form their own musical event objects containing whatever information they see fit. Inheritance is also used within the minimal set of objects provided by the system to simplify run-time routines.

In Figure 18 the minimalistic configuration of musical events is shown. The most basic object is `BaseAtomic`. This object stores all the attributes a musical event *must* have. The field `from_tree` provides a link back to the derivation tree, so attribute names specific to a particular event can be accessed, such as `time_event_dur` in the case of a note-head.

Additional objects provided by the system form an inheritance hierarchy. In Figure 18 the object `HLinkAtomic` inherits from `BaseAtomic` and can therefore access all the fields in `BaseAtomic`,

such as `x_hotspot` and `from_tree`**. The hierarchy abstracts away detail for the convenience of the programmer developing run-time code. For example, when traversing a horizontal thread, the abstraction *`HLinkAtomic`* means that the traversal code need only process objects of this type, irrespective of the objects being *`TimeAtomic`* or *`SecondaryAtomic`*. The functions `horiz_next(level)` and `horiz_prev(level)` move forwards and backwards, respectively, where `level` is an enumerated type specifying the level to use in the structured list. When an object requiring modification is found, its type can be checked using `hlink_type` and then type cast to *`TimeAtomic`* or *`SecondaryAtomic`* appropriately.

The final layer of musical event objects in the inheritance hierarchy is defined by the implementor of the run-time routines. Figure 19 details one possible setup for CMN. Events vary considerably in their needs. In this example a new object type has been created for each classification of musical feature. In Figure 19(a) *`TimeAtomic`* is subdivided into note and rest, and a representative collection of new objects for *`SecondaryAtomic`* is shown in Figure 19(a). The roles of *`CmnTimeAtomic`* and *`CmnSecondaryAtomic`* are similar to *`HLinkAtomic`* in the minimalistic system hierarchy, serving as common objects that can be used when differentiation between the more specific object types is not required. In C++ these objects are implemented using virtual inheritance.

The information stored in each new object type depends on the musical features the system is configured to classify, as well as the intended application of the data once the page has been processed semantically. In this example, a substantial set of music notation is catered for and the information stored is intended for complete graphical reconstruction. This is why information concerning the scope and direction of graphical shapes such as a beam, a slur and a note stem is stored in *`CmnNoteAtomic`*. A system targeted at only audio playback need not be concerned with such details.

Field names attempt to capture the meaning of the data stored, but for completeness we now elaborate upon the field names used in this example. This is best done by studying the steps that transform the pre-constructed graph into its final form (Table II).

First, the basic pitch†† of each note-head (*`CmnNoteAtomic`*) is calculated based on the staff position it occupies, and this is stored in `pitch`.

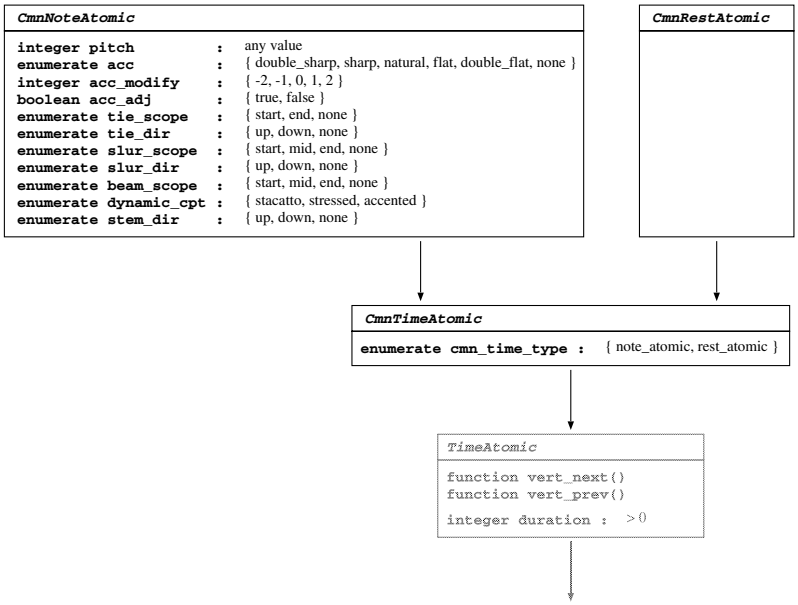
Next, the basic duration‡‡ of each note or rest (*`CmnTimeAtomic`*) is extracted from its derivation tree (`time_event_dur`) and stored in `duration`. This value was calculated during the primitive assembly stage and takes into account the number of tails or beams attached to the note-head's stem, the type of the note-head and the number of durational dots present, but it excludes the influence of any time signature. For example, a crotchet note in $\frac{3}{4}$ time represents one beat, but in $\frac{6}{8}$ time the same musical feature lasts for two-thirds of a beat; no time signature information has been taken into account at this point in the system, however, and both scenarios store the same duration in the derivation tree.

The next two steps (3 and 4) record the direction of note stems and the scope of beams, with respect to individual note-heads. This information was stored during the primitive assembly stage, and therefore

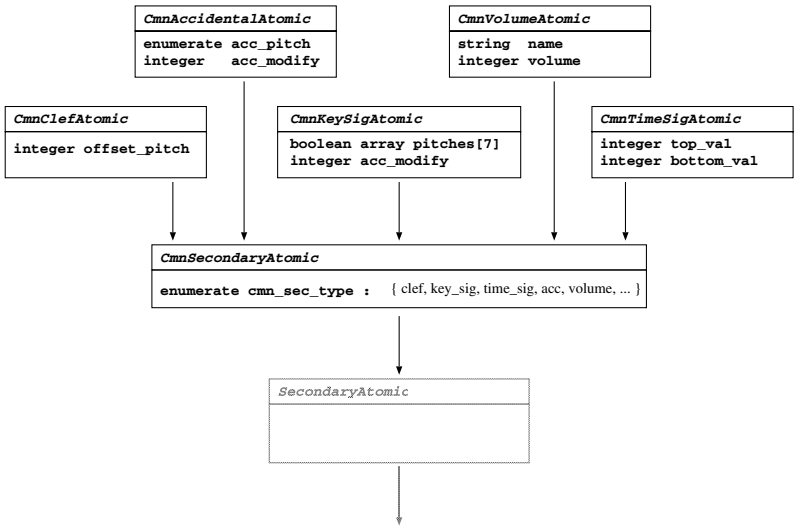
**Here we are assuming a simple form of inheritance, where an object field cannot be shielded from inherited objects.

††Pitch based solely on the position of the event with respect to the staff. For instance, '1' represents the bottom line of a staff, '2' represents the first staff gap above this and so on.

‡‡Duration based solely on the graphical primitives that constitute the musical feature. For instance, '4' represents a crotchet (quarter note), '8' represents a quaver (eighth note) and so on.



(a)



(b)

Figure 19. The developer defines the final layer of musical events in the object hierarchy, through inheritance (a) `TimeAtomic`; (b) `SecondaryAtomic`.

Table II. The order of graph transformations.

	Order of graph transformations
1	Calculate basic pitches
2	Calculate basic durations
3	Store stem directions
4	Store beam scopes
5	Apply clefs
6	Apply key signatures
7	Apply accidentals
8	Apply time signatures
9	Apply slurs and ties
10	Apply dynamics
11	Synchronize bars

is a straightforward extraction process. This done, we are now in a position to apply the effects of secondary events on the primary (time) events.

The clef group (`CmnClefAtomic`) is first. For every clef in the lattice, `offset_pitch` is retrieved from its derivation tree and stored. Then the lattice is traversed at the *staff* level, modifying the pitch of all note events encountered by `offset_pitch`. This process proceeds until the end of the line is reached, or another clef is encountered, in which case the new value of `offset_pitch` supersedes the value being used. As an example, let us assume integers represent pitch, with '0' for middle C, '1' for D, '2' for E, '8' for C one octave above middle C and so on. The offset value required for a treble clef (which indicates that the bottom line is the note E), therefore, is '+1' and is encoded in the production rule for the treble clef in the BCG. A note-head positioned over the bottom line of a staff influenced by a treble clef, therefore, has the basic pitch of '1' modified by the 'apply clef' routine, resulting in the value '2,' which is indeed the first E above middle C.

A similar pattern of retrieval and application occurs for key signatures, accidentals and time signatures, where once again precedence is given to new occurrences of the influencing object type. Key signatures traverse the lattice at the *staff* level, accidentals at the *bar* level and time signatures at the *voice* level.

In applying the effect of a key signature, the routine accesses the Boolean array `pitches` which represents the 'lettered' names of the notes { C, D, E, F, G, A, B } and indicates which accidentals are present in that particular key. The field `acc_modify` stores the modifying effect the key signature has on a note-head whose pitch (modulo 8) matches an entry in the array.

By comparison, the use of data fields in the application of accidentals and time signatures is simpler. For an accidental (`CmnAccidentalAtomic`), `acc_pitch` defines the pitch of the note-heads affected and `acc_modify` defines the change in pitch required. For a time signature (`CmnTimeSigAtomic`) the value of the bottom number, `bottom_val`, is distributed to all notes and rests.

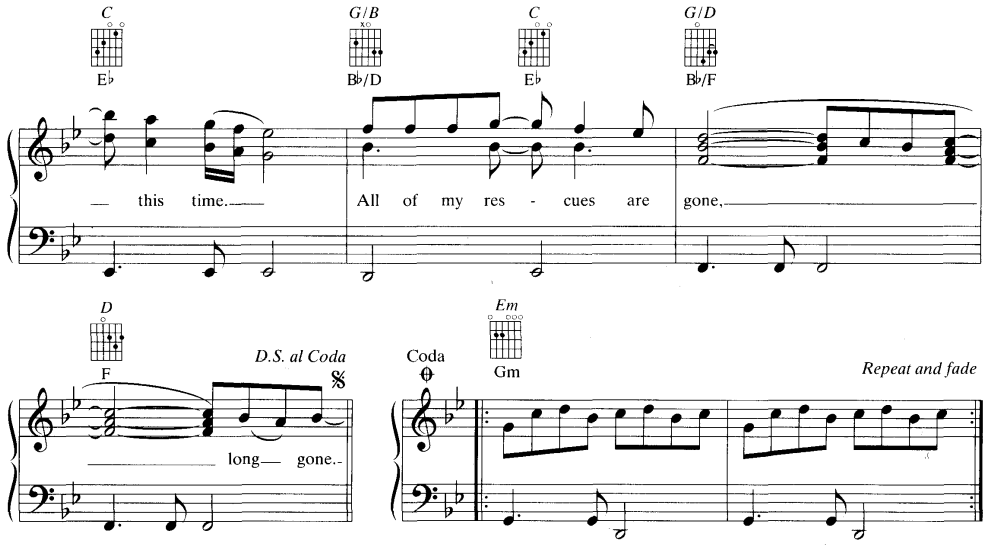


Figure 20. Slurs and ties are similar in appearance, and can both be broken over lines.

An alteration of a note's pitch due to an accidental or a key signature is represented in `CmnNoteAtomic` by an enumerated type field (`acc`) *and* a modifying integer field (`acc_modify`). This redundancy is included to simplify later routines, where sometimes it is more convenient to use the modifying pitch and at other times the type of accidental. In addition to these fields, `acc_adj` indicates whether or not the accidental causing the alteration in pitch should be explicitly shown in the reconstructed score.

Applying the effect of slurs and ties to notes is complex since these objects can be broken over lines. Problems are further compounded by the fact that the two musical features are at times identical in graphical appearance and can only be distinguished after consideration of the context. Both these problems appear in Figure 20. Three ties and one slur are broken between the end of the first line and the start of the second, yet the slur starts and stops at the same height, giving it the same graphical appearance as a tie. Slurs and ties can be successfully processed together by first determining the scope of each object using the *voice* level, then identifying which note-heads fall within this scope. The two types of musical feature are distinguished by studying the number and pitch of the identified notes and thus the appropriate data can be stored in the relevant `CmnNoteAtomic` objects. In the case of the object being a tie, one more step is required. If the starting note of the tie is affected by an accidental (`acc`) then this value must be propagated through to the second note which could be in another bar.

Like slurs and ties, dynamics (indicating loudness, speed and so on) are free floating objects and are consequently processed in a similar manner. First the scope of a particular volume marking is determined using the *voice* level, then affected objects are modified accordingly.

Synchronization of bars is the final step to processing the graph. Unlike the previous steps, which modify the contents of nodes in the lattice, this step modifies the lattice structure itself in an attempt to be tolerant of timing errors. For each concurrent bar within a staff system, the maximum bar duration is found and all other bars are padded out to this length by inserting synchronization rests (`CmnRestAtomic`) into the lattice structure at the end of each bar. Any mistakes involving timing within a bar are thus compensated for before the start of the next bar. Synchronization is a simplistic form of error correction. A better strategy would be to use the time signature information per bar to correct any anomalies.

SUMMARY AND CONCLUSIONS

CANTOR is a flexible system, easily adapted to quite diverse styles of music notation with relatively little extra coding required. It is also effective at recognition, with error rates of around 2–4% [5] on a range of scanned music. We believe a large part of this success is due to the underlying design.

In this article we have described the design of our music notation construction engine, the component of an OMR system that deals with high-level music recognition and the one that has seen the largest variety in paradigms used to solve it. By decomposing the task into two stages (primitive assembly and musical semantics) we were able to avoid much of the code complexity described by other researchers. Primitive assembly is based on a modified DCG parser, where a bag of tokens is used instead of a list, and constraints and restricted backtracking are used to limit the combinatorially large search space that using such a data type entails.

Musical semantics is accomplished by developing a set of graph traversal routines and applying them to a pre-constructed lattice-like graph automatically built by the system based on the (x, y) coordinates of the assembled musical features. C++ object inheritance and the specially designed constructs of time-threads and structured lists are used to simplify the task. To maintain the extensibility aspect of the CANTOR project, traversal routines are implemented as dynamic components to the system, with a particular set of routines written to process the chosen type of music notation linked in at run-time.

Comparing this work with other projects certain themes recur—most notably the string-based grammar work of Coüasnon *et al.* and the graph construction work of Fahmy *et al.* However, there are differences as well. We only use the string grammar to assemble primitives, consequently the development of the parser is simple (220 lines of code) and is written in Prolog, not λProlog. Of course, the other parts included in Coüasnon's parser must be implemented elsewhere in the system, but we found a traditional procedural programming language (C++ in our case) well-suited to the task.

The construction of a graph representing the recognized scanned music is fundamental to an OMR system. Blostein *et al.*'s and the other works that followed set this in a formal context by adopting a graph grammar approach, or to be precise, an attributed programmed graph grammar approach (a particular class of graph grammar). While our methodology is less formal, many of its features can be seen in the former technique. The term 'an attributed programmed graph approach' describes our method well and also emphasizes the areas of similarity with other work. Our experience is that the technique adequately covers the required task without being over-complicated by other factors.

Finally, we return to the earlier point that the separation of the construction engine into a primitive assembly stage and a musical semantics stage heightens the similarity between the second stage and certain tasks performed by music editors. In a way this is to be expected since the graph output by the

construction engine is a semantically rich representation of the scanned page, rich enough for music editor files to be generated; consequently, there must at least be a certain degree of equivalence between the two forms. The similarity, however, goes further than this.

Many of the tasks performed during the musical semantics stages have analogous operations in music editing software—examples abound. On detection of a key signature, a routine traverses the graph, modifying the pitch attribute of subsequent note nodes. This is analogous to the music editor operation 'add new key signature' which, on completion of the task, can be directed by the operator to modify the pitch of all subsequent notes to be consistent with the new key signature. The same is true for new clefs and new time signatures. On detection of a bar with an inconsistent duration (inconsistent with either the metre or other bars played simultaneously) a node attributed with the appropriate values is added to the lattice to form a synchronizing rest at the end of the bar. This modifies the graph in the same way an 'add rest' operation does in a music editor.

The subject of on-going work [29], it is also possible to initiate a musical analysis traversal of the lattice to detect more subtle errors in the reconstructed music. Pitches and durations are studied and changes made (node attributes altered) in accordance with the rules of music theory. Such corrections require the same ability a music editor has to change the duration and pitch of notes. As a final example, a music editor can output the music notation in a variety of musical file formats, as can the musical semantic stage.

Given this similarity and that even the best OMR systems still make mistakes, it is natural that these two tasks should become integrated. Indeed this trend is already evident in some commercial editor systems that now offer modest OMR capabilities, and as OMR research advances this will increasingly become the case. In fact there is an added advantage to structuring an OMR system this way compared with correcting the output of the construction engine with a music editor as a post-processing step: based on a statistical profile of errors built by processing a sizable cross-section of music, a music editor's user interface could be adapted to better facilitate the types of operations that are necessary to correct a score that has been input through an OMR process. Current editors are not necessarily well tuned to these types of corrections. For example, when there is a series of triplets in a piece of music, a common habit in CMN is to drop the ornamental '3' after the first few occurrences. Unless the OMR system employs some form of global analysis of the detected timing information, these omissions result in mistakes that carry through to the reconstructed score. In some music editing packages the conversion of three quavers beamed together into a triplet can be convoluted, requiring several operations. This is a situation that would be easy to improve and the idea can even be taken one step further where the music editor detects and corrects likely OMR errors, perhaps provided in a fashion similar to a 'spell check' facility in a word processor.

Source code for the CANTOR construction engine can be downloaded from: http://www.cs.waikato.ac.nz/~davidb/omr/download/ where it is available in two archive formats: cantor_ce.zip and cantor_ce.tar.gz. The software has been written and tested under Unix. See the README file in the archived package for further details.

REFERENCES

1. Pruslin D. Automatic recognition of sheet music. *ScD Dissertation*, Massachusetts Institute of Technology, Cambridge, MA, June 1966.

2. Prerau DS. Computer pattern recognition of standard engraved music notation. *PhD Thesis*, Massachusetts Institute of Technology, Cambridge, MA, September 1970.
3. Carter NP. Automatic recognition of printed music in the context of electronic publishing. *PhD Thesis*, University of Surrey, February 1989.
4. Ng KC. Automated computer recognition of music scores. *PhD Thesis*, University of Leeds, Leeds, UK, 1995.
5. Bainbridge D. Extensible optical music recognition. *PhD Thesis*, Department of Computer Science, University of Canterbury, Christchurch, NZ, 1997.
6. Fujinaga I. Adaptive optical music recognition. *PhD Thesis*, Department of Theory, Faculty of Music, McGill University, Montreal, Canada, 1997.
7. Coüasnon B. Segmentation and recognition of documents guided by *a priori* knowledge: Application to musical scores. *PhD Thesis*, IRISA, France, 1997.
8. Stückelberg MV. Preview of an architecture for musical score recognition. *Technical Report UNIGE-AI-97-01*, University of Geneva, 1997.
9. Bainbridge D, Carter N. Automatic reading of music notation. *Handbook on Optical Character Recognition and Document Image Analysis*, Bunke S, Wang PSP (eds.). World Scientific: Singapore, 1997; 583–603.
10. Selfridge-Field E. Optical recognition of music notation: A survey of current work. *Computing in Musicology: An International Directory of Applications* 1994; **9**:109–145.
11. Kato H, Inokuchi S. A recognition system for printed piano music using musical knowledge and constraints. *Proceedings of the International Association for Pattern Recognition Workshop on Syntactic and Structural Pattern Recognition*, Murray Hill, NJ, June 1990; 231–248.
12. Coüasnon B, Camillerapp J. Using grammars to segment and recognize music scores. *International Association for Pattern Recognition Workshop on Document Analysis Systems*, Kaiserslautern, Germany, October 1994; 15–27.
13. Coüasnon B, Brisset P, Stephan I. Using logic programming languages for optical music recognition. *The Third International Conference on the Practical Application of Prolog*, Paris, France, April 1995; 115–134.
14. Fahmy H, Blostein D. A graph grammar for high-level recognition of music notation. *Proceedings of First International Conference on Document Analysis and Recognition*, vol. 1, Saint Malo, France, 1991; 70–78.
15. Fahmy H, Blostein D. Graph grammar processing of uncertain data. *Advances in Structural and Syntactic Pattern Recognition (Proceedings of International Workshop on Structural and Syntactic Pattern Recognition)* (*Series in Machine Perception and Artificial Intelligence*, vol. 5), Bunke H (ed.). World Scientific: Bern, 1992; 373–382.
16. Fahmy H, Blostein D. A graph-rewriting paradigm for discrete relaxation: Application to sheet-music recognition. *International Journal of Pattern Recognition and Artificial Intelligence* 1998; **12**(6):763–799.
17. Baird H, Bunke HS, Yamamoto K (eds.). *Structured Document Image Analysis*. Springer: Berlin, 1992.
18. Baumann S. A simplified attributed graph grammar for high-level music recognition. *Proceedings Third International Conference on Document Analysis and Recognition*, vol. 2. IEEE, 1995; 1080–1083.
19. Reed KT, Parker JR. Automatic computer recognition of printed music. *Proceedings 13th International Conference on Pattern Recognition*, vol. 3. IEEE, 1996; 803–807.
20. Haken L, Blostein D. The Tilia music representation: Extensibility, abstraction, and notation contexts for the Lime music editor. *Computer Music Journal* 1993; **17**(3):43–58.
21. Bainbridge D, Inglis S. Musical image compression. *Proceedings IEEE Data Compression Conference*, Snowbird, UT, Storer J, Cohn M (eds.). IEEE Computer Society Press, 1998; 208–218.
22. Chiang N-T. Optical Music Recognition: processing the Sacred Harp. *MSc. Thesis*, Department of Computer Science, University of Waikato, NZ, 1998.
23. Bainbridge D, Nevill-Manning C, Witten I, Smith L, McNab R. Towards a digital library of popular music. *The 4th ACM Conference on Digital Libraries*, Berkeley, 1999; 161–169.
24. Bainbridge D, Wijaya K. Bulk processing of optically scanned music. *The 7th International IEE Conference on Image Processing and its Applications*, Manchester, U.K., 1999; 474–478.
25. Wijaya K, Bainbridge D. Staff line restoration. *The 7th International IEE Conference on Image Processing and its Applications*, Manchester, U.K., 1999; 760–764.
26. Bainbridge D, Bell T. An extensible optical music recognition system. *Proceedings of the Nineteenth Australasian Computer Science Conference*, Melbourne, 1996; 308–317.
27. Barr A, Feigenbaum EA. *The Handbook of Artificial Intelligence*. Addison-Wesley: Reading, MA, 1981.
28. Bratko I. *Prolog: Programming for Artificial Intelligence*. Addison-Wesley, 1990.
29. McPherson JR, Bainbridge D. Coordinating knowledge within an optical music recognition system. *The Fourth New Zealand Computer Science Research Students' Conference*, Christchurch, NZ, 2001; 50–58.

