The design of ReadScoreLib – Anthony Wilkes, Organum Ltd June 2021

This design overview is provided for anyone interested in RSL and how it works.

RSL has been under development since 2006. RSL aspires to the greatest possible accuracy in reading engraved scores. This is an ongoing process.

Because of the many similarities, RSL is designed on the model of the language compiler. Although the input is graphical rather than textual, the two tasks are essentially similar. Both take a more or less well defined non-natural input language and convert it to another equivalent one. Similar ideas of syntax and semantics apply, and the means and data structures involved are also similar.

The main difference between the two is that in the case of OMR the material processed is imperfect. Optical process are intrinsically lossy, and the language of musical notation itself is less well defined and more open ended than a programming language.

This lossy character pervades the whole OMR process. Everything discovered from the score is subject to a degree of error. Fortunately music notation also contains some redundancy, which often makes it possible to corroborate facts by statistical and other means.

RSL Phases

Like a compiler the RSL process is divided into phases. RSL uses the compiler names for these: Syntax analysis, Semantic analysis/translation, Optimization and Code generation.

The compiler concepts of separate compilation and linking also apply to the efficient processing and combining of pages, and we have also adopted the term *build* for the complete end-to-end process.

Because optical image capture is lossy, the RSL process begins by trying to recover the original music engraving that lies behind the input image.

Music notation consists of black foreground graphics on a white background. As laid out by the engraver the staff lines are horizontal and the measure lines vertical. Whatever the state of the image, this is the form in which the music started, and RSL must first recover it from the input and take the material back to ‘standard form’.

Two sorts of transformation are applied:

1) Distortions of position, angle and shapes. The music can be skewed, but it can also contain distortions arising from page curvature and close-up photography.

2) Distortions of focus, light level, color and greyscale. This kind of distortion can vary over the image due to light fluctuation and surface quality.

Once the engraved image is recovered, RSL’s first real step is a syntax analysis phase through which the input image(s) is transformed into a tree similar to the parse tree generated by a classical compiler. This tree mirrors the hierarchical structure of music as score, systems, staves, measures and the syntactical structure of musical objects.

Once the tree is complete, a linear structure is built alongside the parse three giving a view of the music as a series of ‘events’ moving left to right through the score. As more becomes known about the score events in the music will receive timestamps defining the musical beat on which each occur.

The next step, like the semantic analyser of a compiler extracts information expressing the ‘meaning’ of the syntactic structures. This includes pitch, key, time and part information. This is a converging process that may require many passes over the music.

When this process is complete the internal representation contains everything necessary to realise the score as music. However it will almost certainly still contain errors resulting from recognition failures, and from missing or inconsistent information. This can include actual mistakes or inconsistences in the score itself.

After this semantic phase, RSL applies a further ‘optimisation’ phase analogous to the optimization phase of a compiler. Optimisation looks for likely mistakes and inconsistencies, and attempts to reconstruct missing material.

Finally, what in compiler terminology would be described as code generators produce output for different ‘architectures’ such as MIDI, MEI or MusicXML.

Syntax analysis

The process of recognising the graphical music in an OMR system is a relatively small part of the whole task, but it is probably the most error prone and the most time consuming.   One approach is to apply relatively unstructured pattern matching.   For simple music this might begin by looking for specific patterns such as note heads over the score, perhaps guided along the horizontal lines of the staff, and then work around them looking for clues indicating connecting material such as stems, tails and beams. This is of course is a very simplified picture, and the complexity of handling real world scores using a pattern matching approach has led to increasing interest in the use of neural nets.

RSL, rather than viewing the process as an exercise in pattern matching, takes the syntax directed approach well established in textual language parsers. In the same way that the parser for a suitable textual language can be constructed from the language's BNF grammar, the RSL parser applies a syntax tree approximating Common Western Notation (CWN) engraved music to an image, and generates a parse tree.

The syntactical structure of CWN (though not the semantic structure) is broadly hierarchical. Score, systems, staves and measures form a hierarchy as do beam groups, stems and heads.

In real music there are of course exceptions and complications: a beam group may span multiple measures, the measure-staff relationship is not strictly hierarchical, and so on. Composers and engravers are also free to break the 'rules' and often do. But the overwhelmingly hierarchical syntax makes the parse tree a highly practical representation for CWN.

In RSL the CWN grammar is represented by a series of classes forming a recursive descent parser. Each grammar class is a parser for its own type. The SCORE parser parses scores by following score syntax in the score image to instantiate objects of the SYSTEM class. The SYSTEM class parses staves and so on. This is all accomplished in a uniform tree walk down to the level of note heads.

This procedure allows parsing to be directed in such a way that only possible structures are sought at a particular point. Even though there must be some backtracking, CWN is well adapted to recursive descent parsing and this allows the basic algorithm to be kept simple, reducing errors, as well as the overall number of pixel visits.

In practice, real world score images do not consist of neat contiguous regions of foreground, each forming exactly one musical symbol. Pixels are often missing, leaving an object broken into several pieces. Objects often overlap and a single island of foreground may comprise several beam groups, accidentals and other symbols overlapping one another. In RSL, reforming the score foreground so as to separate distinct objects and bring together separated fragments is part of the parsing process. This combination and separation is done physically on a working bitmap using constructions and barriers. The geometric rules for doing so form part of the parsers and are applied through transactionalised 'experiments' applied recursively on the physical image. In a poor bitmap many constructions may be applied and rolled back before a satisfactory parse is found As a consequence poor images take longer to parse than perfect ones.

To simplify the parsing process, the main working bitmap is rendered in 24 planes, allowing each pixel to be encoded with identity information. Thus each pixel of an object such as a note head or a fermata dot can act as a pointer to its parse tree node.

Each node in the parse tree also defines a local coordinate system with Cartesian (0, 0) as the origin of its own object. Using the object's unique bit pattern, each node also defines a virtual FOREGROUND(x, y) function yielding true only when the addressed pixel belongs to the object. This applies even though the object may overlap others.

Syntax analysis is applied a page at a time, possibly in parallel and yields a page object analogous to the object file generated by a compiler.

Translation

At this stage the score ceases to be a series of page objects and becomes a linear sequence representing the whole piece

Because individual systems may be sparse and represent only a subset of the scoring, the instruments actually present in each system must now be identified, and linked over the score. As there is often nothing explicit, or at least nothing reliable to identify a staff, the matching process utilises information such as staff gauge, clef, transposition, and general musical activity.

To build the global score representation, a special walk of each page parse tree visits all terminal nodes representing score 'events'. These are linearised in the form of relational tables. Each event takes a globalised x-coordinate and a time stamp. A link into the parse tree allows everything known about the object and its relation to the whole score to be available. If the bitmap is still cached the object's bits with each object in a unique bit pattern are also accessible.

This database is then traversed in a series of passes until every measure is found and solved. The first task is to establish all measure boundaries and the metric time assigned to each. This is a converging process that often takes several passes. A translation unit will contain zero or more time signatures. In much music it is not possible to read all time signatures sufficiently reliably. However it is usually possible at least to identify the points in the score where there must be a time signature. From this, the score can be divided into isotime blocks (these are similar to basic blocks in a compiled programming language). In an isotime block all measures contain the same number of beats. In simple music it would in theory be possible to read off the length of each measure from the notes and rests therein. In practice this is complicated by the following:

1) There may be missing measure lines or measure lines that are artifacts.
2) It may not be obvious which notes belong to which part. All measures must be assumed multi-part until proved otherwise.
3) A measure may contain tuplets unmarked as such. Unmarked triplets are more common than marked ones and may contain rests or missing rests, or lie over more than one staff. In some isotimes triplets predominate overall.
4) Anacruses and compliment anacruses.
5) Notes and rests on the same beat may not be aligned vertically, even on the same staff. Conversely, aligned heads may actually lie on *different* beats. This is often the case where triplet-quavers occur against semiquavers
6) Small sized notes that may or may not occupy logical time (grace, cue or just small)
7) Free bars not marked as such.

In general, solving an isotime block sufficiently to determine the metre is a multi-pass converging process. As more issues are resolved others become resolvable. RSL compiles a histogram of the length of each measure-part using some measure of length which becomes more accurate as the process proceeds. After weighting for likely combinations, the modes and overtones will often give clues to the metre and to any repeating tuplet structure. Surprisingly often, the metre is not the dominant mode.

Once the metre, and those measures that are exceptions to the metre are known, each measure can be individually solved. Solving a measure means assigning a timestamp to every note and rest, and is essentially a matter of solving (sometimes large numbers of) simultaneous equations. However in some music guesswork is required using contextual knowledge.

Optimisation

The purpose of a compiler optimiser is not the same as RSL's optimiser, however its place in the overall picture is similar. In both cases it acts on the otherwise finished output looking for specific types of situation, often heuristic where the result can be improved.

The RSL optimiser uses a collection of techniques and heuristics. Here are some examples

1) Ties and accidentals – when a modified note is carried over the measure line the accidental is normally not repeated. This fact can be useful if one or other (the tie or the accidental) has not been recognised, perhaps due to poor input. Small ties, especially those that overlap a staff line are sometimes difficult to detect, and can by in some cases be safely guessed.
2) Accidentals in multi-part music – where the same degree of scale is differently modified in multiple parts an accidental may have been missed or misrecognised. If a genuine false relation can be ruled out the note can be safely corrected.
3) Ties with missing notes – where there are more ties than heads on one side of the tie, heads (especially open heads which are hard to detect) can be recovered. Conversely, missing tie arcs can be reconstructed.
4) Key signature and clef redundancy – where the clef is uncertain a key signature can be used to guess what it must be.
5) Global key signature analysis – with a global view of all key signatures, and places where there must be key signatures it is possible to fill in any that have been lost or recognized incorrectly. A similar technique is possible for clefs.
6) Part consistency – where a multi-part passage on one staff joins a single part one as often happens in keyboard music, global analysis over the score can prevent untidy part switching.

Code generation

From the end of the translation phase the parse tree and the linear database tables contain enough information to generate output formats such as MIDI and MusicXML. In the case of MusicXML the aim is to represent the original music faithfully, retaining the same cross-staff beaming, tremolos arcing and so on, but with an agnostic layout that does not restrict a rendering program to (for example) any particular number of measures per system.

The MIDI generator goes a little further. Here the aim is to make the music sound as musical as possible. There is no attempt to put in expression, but the length of fermata must be appropriate, the articulation sound good, the grace notes and ornaments elegant and natural, and so on.

New code generators can be added for other formats as desired.

The build process

In RSL pages are treated as separate compilation units, in the same way as files are in many programming languages, and for similar reasons. The later stages of compilation require a global view of the score, but up to the parse tree stage they can be processed separately and independently to yield page objects.

Because it involves the source bitmap, compiling page objects is the most time consuming and the most memory hungry of the build phases. However, as these require a global view of the score, only when all page objects exist can the semantic and optimization phases begin. For many reasons build speed is important for an OMR system, especially in an interactive setting, and it is important that pages can be built in parallel, using one processor core per page.

The future of RSL

Both human and machine reading of music is intrinsically imperfect. Humans learn to sight read well by bringing to bear wider knowledge of music and indeed knowledge in general.

Although a program cannot have knowledge of music in the way humans do, there are still endless possibilities for using what can be learned about a score in general to improve recognition accuracy. Vital in doing this is the ability to organize score knowledge effectively and to process it efficiently so that more can be done without seriously impacting build performance.

The present version of RSL (4.339) builds at the rate of between 1 and 2 seconds per page. The generation of RSL now under development has a threaded architecture which delivers a performance of around 4 pages per second on today's mobile and desktop hardware. We expect this and other improvements to allow us to carry on increasing accuracy indefinitely.