**INScore expressions to compose symbolic scores**
Gabriel Lepetit-Aimon, Dominique Fober, Yann Orlarey, Stéphane Letz

▶ To cite this version:
Gabriel Lepetit-Aimon, Dominique Fober, Yann Orlarey, Stéphane Letz. INScore expressions to compose symbolic scores. International Conference on Technologies for Music Notation and Representation, 2016, Cambridge, United Kingdom. pp.137-143. hal-02158908

**HAL Id: hal-02158908**
https://hal.archives-ouvertes.fr/hal-02158908

Submitted on 18 Jun 2019

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
INSCORE EXPRESSIONS TO COMPOSE SYMBOLIC SCORES

G. Lepetit-Aimon  D. Fober  Y. Orlarey  S. Letz
Grame
Centre national de création musicale
Lyon - France
gabriel.lepetit.aimon@grame.fr

ABSTRACT

INScore is an environment for the design of augmented interactive music scores turned to non-conventional use of music notation. The environment allows arbitrary graphic resources to be used and composed for the music representation. It supports symbolic music notation, described using Guido Music Notation or MusicXML formats. The environment has been extended to provided score level composition using a set of operators that consistently take scores as arguments to compute new scores as output. INScore API supports now score expressions both at OSC and at scripting levels. The work is based on a previous research that solved the issues of the notation consistency across scores composition. This paper focuses on the language level and explains the different strategies to evaluate score expressions.

1. INTRODUCTION

Contemporary music creation poses numerous challenges to the music notation. Spatialized music, new instruments, gesture based interactions, real-time and interactive scores, are among the new domains that are now commonly explored by artists. Classical music notation doesn’t cover the needs of these new musical forms and numerous research and approaches have recently emerged, testifying to the maturity of the music notation domain, in the light of computer tools for music notation and representation. Issues like writing spatialized music [1], addressing new instruments [2] or new interfaces [3] (to cite just a few), are now subject of active research and proposals.

Interactive music and real-time scores are also representative of an expanding domain in the music creation field. The advent of the digital score and the maturation of the computer tools for music notation and representation constitute the basement for the development of this musical form, which is often grounded on non-traditional music representation [4] [5] but may also use the common music notation [6, 7].

In order to address the notation challenges mentioned above, INScore [8, 9] has been designed as an environment opened to non-conventional music representation (although it supports symbolic notation), and turned to real-time and interactive use [10, 11]. It is clearly focused on music representation only and in this way, differs from tools integrated into programming environments like Bach [12] or MaxScore [13].

INScore has been extended with score expressions that provide symbolic scores composition features (e.g., putting scores in sequence or in parallel). Building new scores from existing scores at symbolic level is not new. Haskell is providing such features [14]. Freeman and Lee proposed score composition operations in a real-time and interactive notation context [15]. Regarding the score operations used by INScore, they are imported from a previous work [16] that was focusing on the music notation consistency through arbitrary scores composition.

The novelty of the proposed approach relies on the dynamic aspects of the scores composition operations, as well as on the persistence of the score expressions. A score may be composed as an arbitrary graph of score expressions and equipped with a fine control over the changes propagation.

The paper introduces first the score composition expressions. Next, the different evaluation strategies are explained and illustrated with examples. The articulation with the INScore environment is presented in detail and followed by concrete use cases. An extension of the primary scores composition design to score expressions composition is next introduced. A generalisation of this approach to the whole set of INScore graphic objects is finally considered in the concluding section.

2. LANGUAGE SPECIFICATION

The main idea behind the project is to design a relevant language that provides easy to use tools to compose and to manipulate symbolic scores. Indeed, as all the operators have already been defined in a previous work [16], the point is to imagine a handy way to use them from INScore but above all, to benefit of the dynamic aspects of the INScore environment.

2.1 The operators

All the operators have a common interface: regardless their actual definition, they always take two scores as input to produce a score as output. The scores are expressed using the Guido Music Notation format [GMN] [17]. A few low-level score manipulation operations are defined (which ap-
The composer applies effects... and mixes them until this creative process where, after choosing sound material, transforming on various materials and combining form and aggregate scores fragments. This process (apply operators allows to compose a music by successively transforming parts of a score, time stretching, or to the score structure (extracting voices). A few additional operations are provided: transposition and application of a score’s rhythm or pitch to another score. The small set of operators is not a real limitation, as the uniformity between their inputs and output make them easy to combine into pipeline designs, creating more high-level operations. The selected basic operators are not intended to cover the composition process (a real programming language like Open Music [18] would be required) but to provide tools for dynamic symbolic score computation, especially in the context of music performance.

Table 1. INScore operators

<table>
<thead>
<tr>
<th>operation</th>
<th>arguments</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>s1 s2</td>
<td>puts the scores s1 and s2 in sequence</td>
</tr>
<tr>
<td>par</td>
<td>s1 s2</td>
<td>puts the scores s1 and s2 in parallel</td>
</tr>
<tr>
<td>rpar</td>
<td>s1 s2</td>
<td>puts the scores s1 and s2 in parallel, right aligned</td>
</tr>
<tr>
<td>top</td>
<td>s1 s2</td>
<td>takes the n first voices of s1, where n is the number of voices of s2</td>
</tr>
<tr>
<td>bottom</td>
<td>s1 s2</td>
<td>cuts the n first voices of s1, where n is the number of voices of s2</td>
</tr>
<tr>
<td>head</td>
<td>s1 s2</td>
<td>takes the head of s1 on s2 duration</td>
</tr>
<tr>
<td>evhead</td>
<td>s1 s2</td>
<td>takes the n first events of s1, where n is s2 events count</td>
</tr>
<tr>
<td>tail</td>
<td>s1 s2</td>
<td>cuts the head of s1 on s2 duration</td>
</tr>
<tr>
<td>evtail</td>
<td>s1 s2</td>
<td>cuts the n first events of s1, where n is s2 events count</td>
</tr>
<tr>
<td>transpose</td>
<td>s1 s2</td>
<td>transposes s1 so its first note of its first voice match s2 one</td>
</tr>
<tr>
<td>duration</td>
<td>s1 s2</td>
<td>stretches s1 to the duration of s2</td>
</tr>
<tr>
<td>pitch</td>
<td>s1 s2</td>
<td>applies the pitches of s1 to s2 in a loop</td>
</tr>
<tr>
<td>rhythm</td>
<td>s1 s2</td>
<td>applies the rhythm of s1 to s2 in a loop</td>
</tr>
</tbody>
</table>

Adapting this approach to the symbolic music notation would not only make the language easy to learn for composer but could offer great tools for composition: carving and assembling score samples using structural operators, placing the musical structure in the center of the creative process. In some ways, the art wouldn’t emerge from the quality of the raw score fragments but from the process that transforms, shapes, and links them together.

It’s with this perspective and emphasis of the structure that the score expressions syntax has been defined. In particular, these expressions should make use of various heterogeneous materials including score expressions or existing score objects.

2.3 Score expressions syntax

Score expressions can be defined using two syntaxes:

\[
\text{score expression:} \quad \text{expr} (\text{score}\text{ operator}\text{ score})
\]

1. The classic syntax reflects the way Guido operators actually work: two scores are combined into one, according to the operator.

2. The alternate syntax defines an expression using a single score, which can be useful to duplicates objects e.g. to provide different views (see section 6.2).

Note that the leading expr token is present to disambiguate parenthesis that are already used in INScore scripts with messages lists.

Both of the syntaxes make use of score arguments. Score expressions are quite permissive regarding to their type:

\[
\text{score:} \quad \text{score string} \quad \text{score file} \quad \text{score object} \quad \text{score expression}
\]
1. **score string**: are GMN or MusicXML strings.

2. **score file**: refers to a score file that should contain GMN or MusicXML data. File path complies to INScore file handling and could indicate an absolute, a relative path or a URL.

3. **score object**: refers to an existing INScore object using a relative or absolute OSC address. The object must be a guido, musicxml or piano-roll object, as well as guido and piano-roll streams.

4. **score expression**: score expressions can be used as arguments of score expressions (in this case the expr token is optional).

Here is an example of a score expression that puts a score in parallel with 2 scores in sequence:

```
expr( par score.gmn (seq "[c]" score) )
```

Note that some operations could take more than 2 scores as arguments. For example, the sequence (seq) or parallel (par) operations could apply to arbitrary lists of arguments and higher-order operations could be defined, similarly to the functional programming fold (or reduce) higher-order function [19]. The current syntax doesn’t support folding but this may be considered in the future. For example, that would allow to write (seq a b c) instead of (seq a (seq b c)).

### 3. Evaluation Specification

The **score expressions** language is first transformed into an internal memory representation. In a second step, this representation is evaluated to produce Guido Music Notation [GMN] [17] strings as output, that are finally passed to the INScore object as specific data.

#### 3.1 Internal representation of score expressions

When encountering an **score expressions**, the INScore parser creates a tree representation of it: arguments are stored as leaves and operators as nodes (Figure 1). This tree form allows to easily store, manipulate, assemble and evaluate **score expressions**.

```
expr( par score.gmn (seq "[c]" score) )
```

The tree representation is strictly matching the expression string. Type specification of arguments is the only difference, whereas types are implicit in **score expressions**, arguments are explicitly stored as GMN code or file or identifier... in the tree form.

Once the internal representation has been constructed by the parser, it is stored with the newly defined object, ready for evaluation.

#### 3.2 Score expressions evaluation process

The evaluation process goes through every nodes of the expression tree using a depth first post-order traversal, reducing all of them into GMN code. A node evaluation is type dependent (Figure 2).

- a GMN file gives its content,
- a GMN string returns the string,
- a MusicXML file returns its content converted to GMN code,
- a MusicXML string returns the string converted to GMN code,
- an object identifier gives its GMN code,
- an operator node returns the application of the operator to the GMN code given as parameters.

```
par
seq
score.gmn
"[c]" score
```

This evaluation scheme avoid recursion issues (e.g., a score that modifies itself using an expression based on its own content) since the caller object is modified only at the end of the evaluation process. All arguments are referentially transparent by default: each argument is evaluated once and its value is then considered constant.

#### 3.3 Dynamic evaluation of score expressions

Referential transparency (i.e., static evaluation) can be a huge limitation. For example, working with guido stream, one could want to maintain the result of a score expression up to date to the stream’s actual state. Thus variable arguments have been introduced using a & prefix: a variable argument is always evaluated regardless of previous values (Figure 3).

```
&score.gmn &score
```

This evaluation scheme avoid recursion issues (e.g., a score that modifies itself using an expression based on its own content) since the caller object is modified only at the end of the evaluation process. All arguments are referentially transparent by default: each argument is evaluated once and its value is then considered constant.

#### 3.4 Example of a score expression

```
expr( par score.gmn (seq "[c]" score) )
```

Note that some operations could take more than 2 scores as arguments. For example, the sequence (seq) or parallel (par) operations could apply to arbitrary lists of arguments and higher-order operations could be defined, similarly to the functional programming fold (or reduce) higher-order function [19]. The current syntax doesn’t support folding but this may be considered in the future. For example, that would allow to write (seq a b c) instead of (seq a (seq b c)).

- a GMN file gives its content,
- a GMN string returns the string,
- a MusicXML file returns its content converted to GMN code,
- a MusicXML string returns the string converted to GMN code,
- an object identifier gives its GMN code,
- an operator node returns the application of the operator to the GMN code given as parameters.

```
par
seq
score.gmn
"[c]" score
```

This evaluation scheme avoid recursion issues (e.g., a score that modifies itself using an expression based on its own content) since the caller object is modified only at the end of the evaluation process. All arguments are referentially transparent by default: each argument is evaluated once and its value is then considered constant.

```
&score.gmn &score
```

This evaluation scheme avoid recursion issues (e.g., a score that modifies itself using an expression based on its own content) since the caller object is modified only at the end of the evaluation process. All arguments are referentially transparent by default: each argument is evaluated once and its value is then considered constant.

```
par
seq
score.gmn
"[c]" score
```

This evaluation scheme avoid recursion issues (e.g., a score that modifies itself using an expression based on its own content) since the caller object is modified only at the end of the evaluation process. All arguments are referentially transparent by default: each argument is evaluated once and its value is then considered constant.

```
par
seq
score.gmn
"[c]" score
```

This evaluation scheme avoid recursion issues (e.g., a score that modifies itself using an expression based on its own content) since the caller object is modified only at the end of the evaluation process. All arguments are referentially transparent by default: each argument is evaluated once and its value is then considered constant.
the dynamic tree attribute is propagated up to the tree root. During the evaluation process, only the dynamic parts of a tree is recomputed. For optimisation, INScore checks if a variable argument value has changed and recomputes the corresponding operator only when needed.

Using variable arguments, an expression tree with arbitrary variable parts can be described: that may be viewed as building a symbolic score with arbitrary aggregation of static and variable parts.

4. SCORE EXPRESSIONS API IN INSCORE

In order to fully integrate score operators, the implementation relies on INScore existing features. As a result, score expressions support URLs as file arguments, interaction events and benefit of web features. Interaction events have been extended notably for the purpose of dynamic evaluation (see section 4.3).

4.1 Declaring score expressions

Both gmn and pianoroll objects can be defined with score expressions using an extension of the set message. The evaluation of the expression is actually triggered by the target object when the set message is processed.

```
/ITL/scene/score set gmn expr(score.gmn);  
/ITL/scene/pr set pianoroll expr(&score);
```

The previous example creates two objects: score is a symbolic representation of the GMN file score.gmn, and pr is a piano roll representation of score (here dynamically evaluated due to the & prefix).

4.2 Score expressions specific messages

Objects that are based on score expressions support additional messages:

- reeval: triggers the re-evaluation of the expression tree taking account of the static and dynamic parts.

```
/ITL/scene/score expr reeval;
/ITL/scene/pr expr reeval;
```

Finally, the score expression of an object can be retrieved with the get expr message:

```
/ITL/scene/score get expr;
```

4.3 Events typology extension

INScore interaction features are based on the association between an event and arbitrary set of OSC messages [10]. These messages are sent when the event occurs (e.g., a mouse down). INScore events typology has been extended with a newData event, that is triggered when the value of the target object changes, either due to a set or reeval message, or because data has been written in a stream object.

Using the expr reeval message in conjunction with the newData event, may trigger the automatic reevaluation of an expression when an object changes. With the example below, changing the content of score will fire the newData event and the associated expr reeval message is automatically sent to copy that updates its content accordingly.

```
/ITL/scene/score set gmn "[a]";  
/ITL/scene/copy set gmn expr(&score);  
/ITL/scene/score watch newData  
~/ITL/scene/copy expr reeval;
```

In order to catch infinite loop issues, newData event handling is postponed to the next INScore time slot. As a result, updating the whole scene after changing the value of an object can take several event loop (if one object is referring to another object, itself referring to another one...) and during this process the INScore’s graphic scene could go through transitory states. However, if objects are defined with recursive references and are auto-updated using this mechanism, INScore will still be able to update the score (without freezing).

5. COMPOSING SCORE EXPRESSIONS

While the expressions already presented allow to compose symbolic scores, it is also possible to compose score expressions which are stored in the referred objects using the prefix ~. Indeed, whereas score and &score refer to the object’s value, ~score refers to the score expression used to define score. In practical, before the first evaluation, all arguments prefixed by ~ are replaced by a copy of the expression tree from the corresponding objects. It allows to easily make use of previously defined score expressions to create more complex ones.

Figure 4 illustrates how the expression tree is expanded with the example below.
6. EXAMPLES

6.1 Canon structure

A simple but still well-known music structure is of course the canon. Creating such structure from a score is quite easy using score expressions.

With the example below, the first line creates a score object based on a GMN file. It is then transposed to a fifth and a second voice is added, delayed of a half note. Because transposing according to a specific interval is not a basic guido operator (the transposition interval is computed from the 2 scores arguments), one should combine transpose with seq and evtail to prepend the score with a note, transpose the whole score using this note and finally remove it.

```
/ITL/scene/score set gmn
expr(seq "[a]" &sample);
/ITL/scene/score set gmn
expr(seq (seq ~score "[b]")) ~score);
```

The result is a simple canon:

```
Original score:

```

Canon:

```
```

```
```

6.2 Multiform synched scores

Score expressions is a great tools to duplicate and dynamically transform scores, keeping every copies synced to the original.

```
/ITL/scene/stream set gmnstream
'(\meter"4/4")');

/ITL/saxo/score set gmn
expr( evtail
   (transpose gmn
   (seq
   "[e\#1]"
   &/ITL/scene/stream )
   "[c2]"
   "[a]"
   ));

/ITL/audience/score set pianoroll
expr( &/ITL/scene/stream);

/ITL/scene/stream watch newData
(/ITL/*/score expr reeval);
```

The previous example creates 2 copies of the GMN stream object stream, one transposed for the saxophone and one displayed as a piano roll, intended as a visual support for the audience. Both are displayed in different scenes. The
last line ensure the update of the copies when `stream` is modified. The `/ITL/scene/stream` argument is re-evaluated due to the `&` prefix. The result is illustrated in figure 6.

```
/ITL/scene/stream set gmnstream
'[$\text{meter"4/4"}]\$';
/ITL/scene/static set gmn
'[$\text{meter"4/4" g e f a f d c/2}]\$';
```

In a second step, the last two bars of the stream are extracted and stored in a new object named `tail`. Since the 'tail' operation cuts the head of the score using the second argument, the duration of this argument is expressed as the tail of the stream using the desired duration (2 whole notes). Note that `tail` expression is using a reference to the stream in order to be updated each time data is written to the stream.

```
/ITL/scene/tail set gmn
expr(tail &stream $'([a*2])')$);
```

The final result is simply obtained using the 'par' and 'transpose' operations. It makes use of references to `tail` but the `static` object is embedded statically. Note that `tail` is used as an intermediate object intended to optimise the computation and to facilitate reading of the expression. It can be hidden from the overall score without affecting the result.

```
/ITL/scene/score set gmn
expr(par &tail
(transpose static &tail));
```

Activation of the score dynamic computation makes use of the `newData` event watched by the `stream` object, that inform `tail` and `score` that their expressions need to be re-evaluated.

```
/ITL/scene/stream watch newData
 (/ITL/scene/part expr reeval,
/ITL/scene/score expr reeval );
```

7. CONCLUSIONS

Combining a simple set of operators with the powerful features of INScore (like URL support, full OSC compatibility, interaction support...), score expressions fully integrate symbolic score composition into an interactive and augmented music score environment. They suggest a creative process based upon musical structures and scores aggregation by giving the possibility to compose various score materials including score objects. Above all, score expressions provide a handy way to manipulate scores regardless to their origin (files, URL, streams...) or their representation (traditional music notation or piano roll) and to design dynamic scores based on arbitrary score composition.

In future work, we’re considering extending the score expressions to all the INScore objects. Such an approach - composing arbitrary graphic resources using a musical semantic - raises issues that are non-trivial to solve. Indeed, if the operations on the time domain could be applied to any object due to their common time dimension, transformations in the pitch domain or based on structured time (like rhythm) implies to extend the musical semantic of the graphics space.

8. REFERENCES


