INScore
Evaluable Expression Reference
v.0.1

D. Fober
GRAME
Centre national de création musicale
<fober@grame.fr>

October 28, 2015

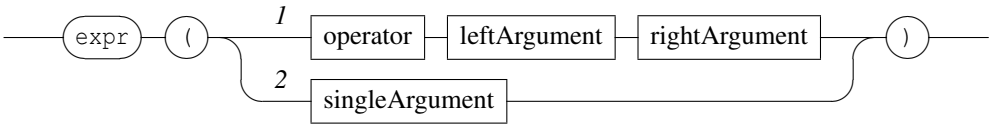
Contents

Chapter 1

General format

Evaluable expression are useful to dynamically construct objects by combining diverse resources (like exiting objects) together according to predefine operators. Unlike variable or messages parameters, an Evaluable Expression contained in a message is not evaluated during parsing, but only when the message is processed. Indeed the parser will only construct a tree version of the expression, which can then be evaluated or re-evaluated as one pleases.
The evaluable expression syntax is quite simple:

EvaluableExpression



- **1**: Define an expression as an operation combining two arguments. Where `operator` is a string referring to a predefine INScore operation, `leftArgument` and `righArgument` the arguments passed to the operator as defined in the next section. Even if expressions are not evaluated on parsing, passing an `operator` that doesn't match any INScore's operator will generate a parse error (thus none of the messages will be processed).
- **2**: Define on expression as a single argument. This syntax is useful when defining an object as a dynamic copy of an other existing object. `singleArgument` is an argument as defined in the next section (with the only difference that it can't be an evaluable expression).

Each of these tokens can, of course, be separated by spaces, tabulations or carriage returns (allowing multiline expression definition).

EXAMPLE

Creating a guido object by sequencing two guido string

```
/ITL/scene/score set gmn expr( seq "[c d e]" "[f g h]");
```

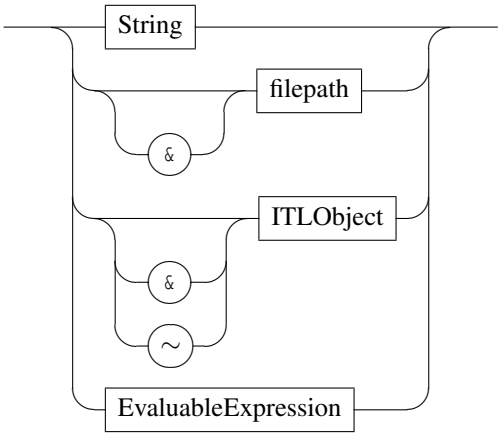
is equivalent to

```
/ITL/scene/score set gmn "[c d e f g h]";
```

1.1 Arguments

Evaluable expression can make use of diverse resources, thus arguments can be from many types. In any way, on evaluation the argument will be reduce to a string so it can be processed by the operators.

Argument



- `String`: on evaluation the string is passed without modifications to the operator, simple quoted or double quoted are accepted.
- `filepath`: on evaluation INScore try to read the file at `filepath` and pass its contents to the operator. `filepath` accept absolute path and relative path (from scene ROOTPATH), quotes are optional if the path doesn't contain any space.
- `ITLObject`: on evaluation INScore try to find `ITLObject` among existing objects then try to evaluatuate a string or an expression from it (for now only symbolic notation objects are supported). `ITLObject` can be the name of an object from the sibling of the current object, a absolute path or a relative path (from the current object path).
- `EvaluableExpression`: an expression can also be used as arguments allowing the creation of expression's trees. In that case one can omit the 'expr' token: parenthesis are sufficient.

EXAMPLE

Creating a guido object by putting the contents of a guido file above his transposition according to the first tone of an existing score.

```
/ITL/scene2/score set gmn "[b]";
/ITL/scene/score set gmn expr(par file.gmn (transpose file.gmn ../scene2/score) );
```

Evaluation modificator

Both `filepath` and `ITLObject` may be preceded by the `&` modificator, stating to INScore that this arguments should be evaluated dynamically (see Evaluating expression section for details).

`ITLObject` also accept the `~` modificator: on evaluation the arguments shall be replaced by a copy of the expression tree of the targeted ITLObject (see Evaluating ITLObject argument for details).

1.2 Evaluating an expression

Evaluable expressions can be manually re-evaluated, for example it can be useful to update a score constructed on existing objects when these experience changes. As one could want to update only certain objects from the expression and as re-evaluating unchanged arguments is a waste of computing ressources, INScore put a difference between statically and dynamically evaluated arguments. Manual re-evaluation command are explained in 'expr' commands section.

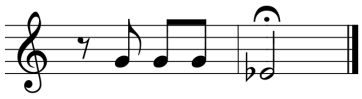
By default arguments are statically evaluated, meaning that, on the first evaluation (when the set message is processed) their evaluated value will be stored. When the expression is re-evaluated these arguments will be replaced by their previously evaluated value, even if the ITLObject or file from which they have been processed have changed. Moreover an Evaluable Expression containing only statically evaluated arguments is also consider statically evaluated, as such, it won't be re-evaluated.

Arguments which are preceded by the `&` modificator are dynamically evaluated: on re-evaluation INScore will ask the ITLObject or the file for its actual content and will update the argument value accordingly. Evaluable Expression containing at least one dynamically evaluated argument is considered dynamically evaluated, the operator is actually re-processed only if its arguments value changed.

EXAMPLE

```
/ITL/scene/score set gmn "[_/8 g g g]";
/ITL/scene/result set gmn expr(seq score &score);
/ITL/scene/score set gmn "[\fermata(e&/2)]";
```

after re-evaluation `/ITL/scene/result` should look like this:



NOTE

Be aware when using statically evaluated arguments that evaluable expression are evaluated during the `set` message processing, meaning that the value taken by such arguments depend on the INScore's state at this exact moment.

1.3 Evaluating an ITLObject argument

When processing an ITLObject argument, the evalutator will (for both statically or dynamically evaluated argument) try to extract a string from the given ITLObject. Passing a incorrect ITLObject identifier will cause the expression evaluation to fail. In general, evaluator can only make use of certain ITLObject (for guido expression evaluator, only symbolic notation object are useful), passing an ITLObject with a uninterpretable type will also cause the evaluation to fail.

Static evaluation

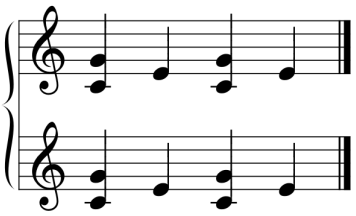
Static ITLObject evaluation is an easy way to construct a score using its actual value, it can be considered as a weak recursion.

EXAMPLE

Creating a simple sample score, then duplicate it using `seq` and `par`:

```
/ITL/scene/score set gmn "[{c,g} e]";
/ITL/scene/score set gmn expr( seq score score);
/ITL/scene/score set gmn expr( par score score);
```

should produce the following output:



Dynamic evaluation

(using &)

Dynamic ITLObject evaluation is useful when manually re-evaluating the expression, the evaluation will be computed again accordingly to the value of the ITLObject at the time of the re-evaluation. This mechanisms also allow recursion. Keep in mind though, that a dynamic argument doesn't automatically update its value when its target's value change, update need to be manually triggered (which prevent any risk of infinite loop).

EXAMPLE

Creating a sample score, then defining a recursive expression

```
/ITL/scene/score set gmn "[c d e f g a b]";
/ITL/scene/score set gmn expr( seq (tail &score "[a]") (head &score "[a]") );
```

Every time `/ITL/scene/score` is re-evaluated, its last note will be moved at the beginning of the score. (see Guido Expressions section for detail on operators).

Copy evaluation

(using ~)

It might be useful, when constructing expression using existing objects defined by expression, to not simply read the evaluated value of the object but to copy its entire expression. Looking back to the static evaluation example: one could want to define `/ITL/scene/score` as the duplication of an existing `/ITL/scene/simpleScore`, and still keep `score` up to date by re-evaluting it when `simpleScore`'s value change... This is possible with copy evaluated arguments.

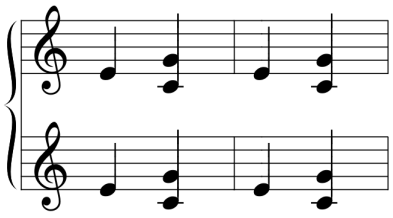
Indeed, on the first evaluation of these arguments (when processing the `set` message), their node in the expression will be replace by the entire expression of their targeted object. Evaluating a copy arguments is in fact expanding the expression using an already created one.

EXAMPLE

Defining `/ITL/scene/score` as a `/ITL/scene/simpleScore` duplicated 4 times.

```
/ITL/scene/simpleScore set gmn "[a]";

/ITL/scene/score set gmn expr( &simpleScore );
/ITL/scene/score set gmn expr( seq ~score ~score);
/ITL/scene/score set gmn expr( par ~score ~score);

/ITL/scene/simpleScore set gmn "[e {c,g} |]";
```

after re-evaluation, `/ITL/scene/score` should look like:



Asking for `/ITL/scene/score` expression should return:

```
/ITL/scene/score expr
expr( par
       ( seq
          &simpleScore
          &simpleScore
       )
       ( seq
          &simpleScore
          &simpleScore
       )
    )
```

1.4 'expr' commands

ITLObject defined using an evaluable expression gain access to these specific commands:

- `get expr`: return the actual expression tree state (including previously evaluated values between brackets for ITLObject and files)
- `expr eval`: force the expression to re-evaluate
- `expr renew`: clean the evaluated values and re-evaluate

Applied to an object which wasn't defined by an evaluable expression, all this commands will cause a bad argument error.

The `renew` command reset the internal state of the evaluated variable, forcing the re-evaluation and update of every arguments in the expression. Be aware that the track of copy evaluated arguments is lost after the first evaluation, thus renewing an expression defined using copy evaluated arguments won't update these arguments to their targeted ITLObject expression. Though, static arguments added by the copy shall be renewed.

EXAMPLE

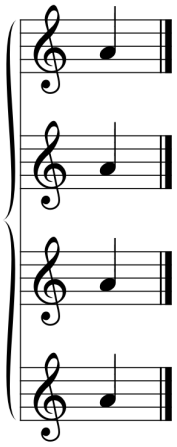
Lets compare the effect of renewing an expression with or without ~:

```
/ITL/scene/score set gmn "[a]";

/ITL/scene/r set gmn expr(&score);
/ITL/scene/r set gmn expr( par r r);

/ITL/scene/score set gmn "[{c,g} |]";
/ITL/scene/r expr renew;
```

`r` evaluated expression is:

```
expr( par r r)
```

so `r` should look like:

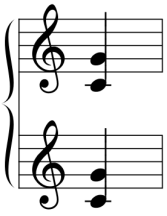


```
/ITL/scene/score set gmn "[a]";

/ITL/scene/r set gmn expr(&score);
/ITL/scene/r set gmn expr( par ~r ~r);

/ITL/scene/score set gmn "[{c,g} |]";
/ITL/scene/r expr renew;
```

`r` evaluated expression is:

```
expr( par &score &score)
```

so `r` should look like:



- Without ~, renewing `r` means updating the two arguments `r` by its new value: `"{[a]} {[a]}"`
- With ~, renewing `r` means updating the value of `score`, thus `/ITL/scene/r expr reeval` would have had the same effect.

1.5 newData event

`newData` is triggered by any object when its value change (generally because of a `set` message). Neither trying to set an object to its actual value without changing its type, nor re-evaluating an object to its actual value will trigger newData.

Of course, the `newData` event can be used together with `reeval` to automatically update an object when the value of an other changes.

EXAMPLE

Creating a copy of `score`, and automatise its update when `score` is changed

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

To avoid infinite loop when using recursion, `newData` event is delayed of one event loop, meaning that, in the previous example, during the event loop that follow `score`'s modification, `score` and `copy` are different (`copy` has not been updated yet...).

NOTE

Because newData event is delayed, if `score` experiences multiple modifications during the same event loop (because multiple `set` messages have been sent together), only his final value will be accessible when newData will be actually triggered, however the event will be sent as many times as `score` have been modified.

NOTE WHEN AUTOMATISING UPDATE

For the reasons raised in the previous note, one should be very careful to delayed update when automatise `reeval` with `newData`. Indeed, in some extreme case, executing a script one line after an other won't have the same result as executing the all script at once!!

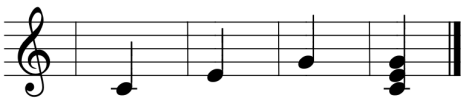
EXAMPLE

Creating a "score buffer", storing every state adopted by `score`

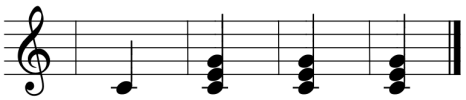
```
/ITL/scene/score set gmn "[c]";

/ITL/scene/buffer set gmn "[]";
/ITL/scene/buffer set gmn expr(seq &buffer (seq "[|]" &score));
/ITL/scene/score watch newData (/ITL/scene/buffer expr reeval);

/ITL/scene/score set gmn "[e]";
/ITL/scene/score set gmn "[g]";
/ITL/scene/score set gmn "[{c,e,g}]";
```

Won't have the same result if run line by line, or the all script as once:

line by line:



all script at once:



To avoid such not deterministic behaviour, one should manually trigger `reeval` after each modification of `score`.

Chapter 2

Guido Expressions

2.1 Operators