Bell, a textual language for the bach library

Jean-Louis Giavitto, Andrea Agostini

bell, a textual language for the bach library

Andrea Agostini
Conservatory of Turin
andrea.agostini@conservatoriotorino.eu

Jean-Louis Giavitto
CNRS, STMS – IRCAM, Sorbonne University
jean-louis.giavitto@ircam.fr

ABSTRACT

In this paper we introduce bell*, a new, small programming language included in the* bach *package for Max. The main design goals of* bell *are ease of integration with Max and* bach *and maximum compatibility with pre-existing syntaxes and conventions* bach *users are already acquainted to. The language is mainly exposed in Max through a new object named* bach.eval*, but other, older objects have been updated so as to take advantage of it. In this article, we shall discuss the main choices underlying the development of* bell*, and give a brief outline of its syntax and the way it integrates within Max.*

1. INTRODUCTION

The *bach* package [1] for Max [2] is a library of more than 200 patches and externals aimed, among the other things, at easing work in the fields of computer-aided and algorithmic composition. It contains some modules for displaying and editing graphically musical scores, but the vast majority of its components are tools for manipulating a tree data structure called *llll* (an acronym for Lisp-like linked list) which somehow constitutes the backbone of *bach*, as virtually all the *bach* objects exchange with each other *lllls* containing pieces of information of any complexity, from a single number to a whole score combining instrumental notation and directives for controlling electroacoustic processes. [1]

It has been clear since the beginning of *bach* that non-trivial tasks in the aforementioned fields require the implementation of potentially complex algorithms and processes, something that the graphical, data-flow programming paradigm of Max, beyond the long-standing theoretical debate on whether it is a programming language or not, is notoriously not well-suited to. [2, 3]

In fact, other systems for computer-aided composition and related activities are either extensions of established textual programming languages including imperative features, or they do actually implement new visual languages, but nonetheless incorporate imperative traits in their underlying textual representation. Examples of the first kind include *OpusModus* [3] and *Common Music* [4] (based upon Common Lisp) or *Music21* [5] and *Abjad* [6] (based upon Python). In the second class we find, for instance, the Lisp-based *PatchWork* family, currently composed of *PWGL* [7] and *OpenMusic* [8]. This class combines the strengths of graphical and textual programming through the use of graphical widgets representing custom-made portions of textual code, something that provided the original impetus behind the idea of augmenting *bach* with such a possibility.

2. LANGUAGE BINDINGS IN MAX

Max offers indeed several options for embedding traditional code in a graphical patch, as it contains out-of-the-box bindings to Java, JavaScript and Lua. Moreover, with somewhat greater effort, it is possible to write one's own external objects in C (and any other language that can be compiled to object code linkable to C, the most obvious example being C++). When investigating the possibile ways to give *bach* users a way to express processes through textual coding, we obviously first considered these possible language bindings. Unfortunately, it quickly became clear that all of them had important drawbacks in the context of what we were looking for.

2.1 C and C++

Objects in C and C++ can be very efficient, handle the native data types of Max and (through the use of a public API) *bach*, and have fine-grained, low-level access to virtually every aspect of the Max environment. On the other hand, code written by the end user must be compiled and then integrated in Max as a plugin, which requires to master the compilation chain and increases considerably the time needed for a write-test-debug cycle.

In addition, C and C++ have a well-deserved reputation for being difficult languages, which very few musicians today have the interest and inclination to learn (whereas extending Max in C was originally meant to be a normal part of the Max user's workflow [2]) and the Max and *bach* C APIs require taking care of a great amount of low-level details only remotely related to the specific problem that the object itself is intended to solve.

For these reasons, C and C++ do not appear to us as valid choices for a language for composers willing to implement musical operations through textual coding.

[1] www.bachproject.net
[2] http://cycling74.com
[3] https://opusmodus.com
[4] http://commonmusic.sourceforge.net
[5] http://web.mit.edu/music21/
[6] http://abjad.mbrsi.org
[7] http://www2.siba.fi/PWGL/
[8] http://repmus.ircam.fr/openmusic/home

2.2 Javascript, Java and Lua

The JavaScript language binding (exposed by the *js* and *jsui* objects) has the major problem of not being well integrated within the threading model of Max, as code is only allowed to run in the main thread. Moreover, the JavaScript implementation, although higher-level than C's, is anyway very revealing of the low-level workings of the Max environment, requiring users to implement methods responding to the different messages sent to the object, deal explicitly with inlets and outlets, taking care of not breaking the Max's conventions about evaluation order and so on.

The same, relatively low-level approach applies to the Java and Lua bindings, respectively exposed by the *mxj*/*mxj~* and *jit.gl.lua* objects.

2.3 Lisp

The fact that the main data structure of *bach* is so close to that of Lisp suggested us Lisp itself as the way to go. Moreover, countless composers have been writing Lisp code over the years for implementing their own compositional processes, sometimes leading to cornerstones such as Camilo Rueda's, Jacques Duthen's and Tristan Murail's *Esquisse* library [9] or Marco Stroppa's OM-Chroma project [10]. Since the only (to our best knowledge) existing Lisp binding for Max, Brad Garton's *maxlispj* [11], seems to be an experimental project rather than a practical one because of its limitations, instability and apparent lack of maintenance, we went as far as trying to embed the ECL Embeddable Common Lisp compiler into a Max object. It quickly became clear, though, that a true, modern Lisp implementation has an enormous set of constraints about threading, garbage collection (which is a curse in a real-time system such as Max, as it can delay computation arbitrarily) and more, and many other data types besides the list. All these features and requirements, if not correctly dealt with, would lead to a horribly crippled Lisp dialect, consisting only of the most fastidious aspects of the language (such as the infamous amount of parentheses and the rather impractical, by today's standards, prefix notation) but deprived of all the features that make Lisp the amazingly powerful and elegant language it still is. So we eventually discarded the Lisp option (an interesting approach to working in Lisp on *llll*s is described in sec. 8).

2.4 Other languages

We also evaluated the possibility of embedding other languages (Haskell and Python briefly enticed us), but none of those we considered seemed apt to the task and, at the same time, reasonably easy to integrate within Max and *bach*, so we found ourself back at the starting point.

3. THE EXPR FAMILY

Max includes another programming language, or a stub thereof, in the *expr*, *vexpr* and *if* objects. They provide a way to implement mathematical expressions and conditionals through a form of textual coding so simple to be easily overlooked, but expressive enough for greatly simplifying small tasks that might require complicated solutions if tackled otherwise. The syntax of the *expr* family of objects is based on infix notation, with support for the most common arithmetical operators and mathematical functions. Data received from the object's inlets are referred to via the *$i1*... *$i9* and *$f1*... *$f9* keywords (respectively for integers and floating-point numbers). Function arguments are surrounded by parentheses and separated by commas. Parentheses are also used for controlling evaluation precedence. The *expr* and *vexpr* objects (differing only in the ability of the latter of performing element-wise operations upon lists of values) return the result of each computation from their only outlet; the *if* object allows choosing between two different results, optionally routing them to two different outlets, according to the result of a boolean expression.

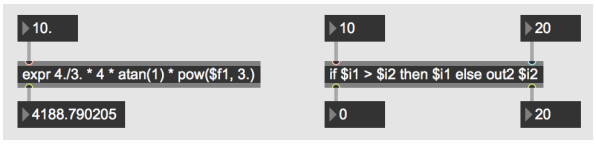

Figure 1. An example of the *expr* and *if* objects.

Since its first version, *bach* has added to the *expr* family a new member called *bach.expr*, essentially based on the behavior and syntax of *vexpr* but capable to operate upon rational numbers and pitches (two base data types added by the *bach* library), as well as perform element-wise operations through a depth-first traversal of the tree structure of llll*s*.

For the simple tasks they are designed for, these objects pose several advantages when compared to the 'real' language bindings discussed above:

- The code can be typed directly in the object box, thus making the role of the object itself in the patch much clearer than if the code was hidden in a separate editor window or read from a source code file.
- Inlets and outlets are managed by the object itself, with no need to specify details concerning their management in the code.
- There is no need to explicitly define specific methods or functions responding to the different messages the object can accept.
- All being considered, the code only expresses the core of what it really has to do, with no need for any surrounding infrastructure (at the other extreme, a Max object written in C easily requires dozens of lines of code only for managing its definition, lifecycle and mechanism of communication with the host environment).

In summary, this architecture allows better embedding of the textual language into the host environment with respect to the traditional language bindings described above. Objects of the *expr* family are usually sprinkled around a

[9] https://github.com/openmusic-project/Esquisse
[10] http://forumnet.ircam.fr/product/openmusic-libraries-en/
[11] http://sites.music.columbia.edu/brad/maxlispj/

patch and take care of relatively small computational tasks, communicating tightly with other objects in charge of the user interface, DSP, MIDI, event scheduling and more. Although their expressiveness is very limited, not supporting variables, iterations, general conditionals and so on, it can be seen as a starting point for a richer functional language.

These considerations led us in the direction we finally chose to pursue: extending the syntax of the *expr* family, adding to it all it takes for turning it into a real, Turing-complete, practically usable (and hopefully easy) programming language, meant to be a strong asset for allowing users to choose the best programming style for implementing their own ideas without breaking the more general graphical paradigm of Max.

4. DESIGN PRINCIPLES

Once determined that a new language (which, from now on, we shall call *bell*, standing for *bach evaluation language for lllls*, but also paying homage to the Bell Labs where Max Mathews invented the seminal MUSIC software) had to be devised, we identified the following main design principles:

Embedding. *bell* should be exposed to Max in a new, specific *bach* object called *bach.eval*. It should be possible to type programs directly in the object box, as in the objects of the *expr* family, or in a separate text editing window, as in the standard Max language objects, or likewise to load them from a source code file (replacing the existing code or appending it). It should be also possible to pass new programs in the form of Max messages. Moreover, the behavior of other *bach* objects should allow being fine-tuned through snippets of *bell* code.

Retro-compatibility. The language should be fully downward-compatible with *bach.expr*, *vexpr* and *expr*. This means that any expression written in one of these objects should be understood by the new language and produce the same result (there are some minor exceptions to this, but they are absolutely marginal and deserve no further mention within the scope of this article).

Applicative style. As programs are expressions and produce results, the language should be a functional one, or at least fully support functional-style programming. As a consequence, all the constructs of the language should yield a result. On the other hand, for practicality's sake, the language should also have a set of imperative-style constructs, such as variables, loops and a sequence operator. Also, *bell* code should need no notion of time (and therefore have no event scheduling capabilities), nor of its role in the patcher, nor of the surrounding Max environment: it should strictly be an expression evaluator.

The final result of a program, which is an *llll*, should be output from the main (and, in the simplest cases, sole) outlet of *bach.eval*; it should be possible to output data from other outlets through specific syntactic features. In the case of other objects, the return value should be passed to the host object for controlling its behavior.

***llll* algebra.** *llls* should be the only data type of *bell*, at least in its first version: the parameters and return values of all *bell* functions, operators and constructs should be *llls*. A wide set of *llll* primitives is already present in *bach*, and most of them are exposed by specific *bach* objects. These primitives should generally be exposed as *bell* functions as well, retaining as much as possible the conventions and nomenclature adopted in their *bach*-object form. Thus, for example, the *llll_rev()* C function of the *bach* API, whose purpose is reversing an *llll*, is exposed in the *bach.rev* object and should be exposed in the *rev() bell* function as well.

5. THE *BELL* LANGUAGE

The implementation of the *bell* language is strictly based upon the design principles stated above. In this section, we shall list some of its most important features, without aiming for completeness or thoroughness.

5.1 *llll* values

As hinted at before, an *llll* is a tree structure mostly resembling a Lisp list. Its leaves can be integers, floats and symbols (the standard data types of Max), rational numbers, musical pitches (expressed in terms of a degree, an optional alteration and an octave: for more details, see [4]) and functions (only useful in the context of *bell*). Each *llll* can recursively contain other *llls* as well: in the standard textual representation, these are enclosed with pairs of square brackets. [12] A typical *llll* literal value might be:

```
1 2.3 4/5 [Eb6 [foo bar]] 7
```

Besides the obvious presence of the *Eb6* pitch, the most apparent difference with respect to most Lisp dialects is that the root level of an *llll* is not enclosed with brackets. Although this may seem a mere cosmetic detail, it bears some important structural consequences, including the fact that there is no concept of atom: single values such as *1* are always considered one-element *llls* in the context of *bach*. The rationale behind this choice is that it allows regular Max lists and messages to be treated as flat *llls*.

5.2 Expressions and arithmetical operators

A *bell* program is an expression, typically composed by sub-expressions connected by operators. A valid, if simplistic, *bell* program is therefore `1`, which is an expression returning the value *1*. A slightly more interesting program is `1 + 2`, returning *3*. Whitespace is optional in this case, so the form `1+2` is equivalent to the previous one.

As in the *expr* family, it is possible to control evaluation precedence through parentheses: so, `2+3*4` returns *14*, whereas `(2+3)*4` returns *20*.

As stated above, the final result of a *bell* program is output from the main outlet of *bach.eval*.

It should be noticed that, in these examples, a number *n* denotes a singleton *llll*, and that arithmetical operators such as + apply point-wise in *llls*.

[12] Previous versions of *bach* used round parentheses for marking sublists. Starting from forthcoming (at the time of writing) version 0.8.1, the 'official' sublist marker will switch to square brackets, although this will only be mandatory in *bell* code, whereas in all the other situations round parentheses will be accepted too, for backwards compatibility.

5.2.1 Implicit concatenation

Concatenation of elements into *llll*s is done implicitly in *bell*. This means that there is no explicit operator for building a new *llll* out of shorter ones (or, in the simplest case, from single elements, that is, one-element *llll*s): elements or *llll*s just placed one after another, separated by whitespace or parentheses, be they typed in the code as literals or results of evaluation of other constructs, are chained together. We might say that list building is performed by the null operator.

Of course, this is true recursively, so in practice any number of expressions can be placed one after another, so as to build long *llll*s. In the simplest case, the individual expressions are just literals evaluating to themselves, as in `1 2 3`, returning the three-element *llll 1 2 3*. But things like `1 2+3 4` (also written `1 2 + 3 4`), returning *1 5 4*, are also possible, showing that the + operator is applied before concatenation, thus consuming the expression that precedes and the one that follows it (concatenating the operator itself as an element of an *llll* can be achieved through the functional form of the operator, *#+*). The last example also shows that +, as more generally all the unpaired operators, has higher precedence than implicit concatenation. On the other hand, parentheses can control this: so, `(1 2) + (3 4)` first evaluates the two concatenations, respectively returning *1 2* and *3 4*, then sums them, thus returning *4 6* (the addition is extended point-wise on *llll*s).

5.2.2 Wrapping operator

Square brackets denote a unary operator that injects a value into an *llll*—that is, just like in the textual expression of *llll*s, a pair of square brackets marks a sublist. Thus, `[1]` actually returns *[1]* (a singleton *llll* whose only element is in turn a singleton *llll*); `[2*3 4]` returns *[6 4]*; and `[[2*3] [4*5]]` returns *[[6] [20]]*, by virtue of wrapping first, then concatenation, and finally wrapping again. As a matter of fact, the combination of wrapping and concatenation makes it possible to see a *bell* program as the textual representation of an *llll* with some special syntactical conventions for intermingling calculations into it.

5.2.3 Sequences

It is possible to sequence expressions, that is, make it so that they are evaluated one after another, in the order in which they appear in the code. This is expressed through the `;` operator. The result of a sequence is the result of the last (that is, the rightmost) expression composing it. So, the `1 ; 2` expression will return *2*.

As with implicit concatenation, it is practically possible to form sequences composed of any number of expressions. As the `;` operator groups left-to-right, `1+2 ; 3+4 ; 5+6` will first evaluate the sequence `1+2 ; 3+4`, returning *7*, and then the sequence `7 ; 5+6`, returning *11*.

The two last examples have no practical meaning, as sequences are only useful when the expressions that compose them have side effects: this applies in particular to variable assignment (see below).

5.3 Variables

Variables are the main construct with side effects in *bell*. They do not need to be declared before using them, and are of a single type, the *llll*. If a variable is read before being assigned a value, it returns an empty *llll* (called *null* in the *bach* system). Variables are either *global* or *local*. Globals are visible by all the *bell*-compliant objects in the Max session. Locals have a leading $ sign and are accessible only from their scope of definition.

Assignment to a variable is performed through the = operator, not to be confused with the == equality comparison operator. There exist a set of C-style assignment operators, too, such as +=, *= and many more. The return value of all the assignment operator is the assigned value and, since they have right-to-left precedence, it is possible to set up chains of assignments.

5.3.1 Inlets

Data received in *bach.eval*'s inlets can be accessed by *bell* code through a set of keywords, the most general being `$x<n>`, where `<n>` is the rank of an inlet (counted from left to right). So, the expression `$x2` returns the *llll* entered in the second-from-left inlet of the *bach.eval* object. This is consistent with *bach.expr*. Other forms of inlet keywords exist, mimicking the strongly-typed ones of *expr*, *vexpr* and *if*: for example, `$i1` returns the *llll* entered in the leftmost inlet, with all its values cast to integers (and there are more forms for floats, rationals and pitches). This feature is present mostly for downward compatibility, but its usefulness is probably quite marginal.

It should be remarked that, as in the other *expr* family objects, the number of inlets of the objects is usually inferred automatically from the inlet keywords. Unlike *expr*, *vexpr* and *if*, though, it is also possible to set it explicitly through the *inlets* object attribute: this can be necessary if the program (and, subsequently, the number of addressed inlets) is meant to change after the object instantiation.

5.3.2 Outlets

It is possible to output data from auxiliary ('extra') outlets of the *bach.eval* object by assigning them to the `$o<n>` pseudovariables, where `<n>` is the rank of an outlet (counted from left to right). This allows using *bach.eval* for routing messages to different parts of a Max patch according to complex conditions: in this sense, *bach.eval* can behave in ways related to Max objects such as *if*, *route* and *gate*. For an example, see fig. 2.

The fact that data are assigned to extra outlets does not detract from the fact that the program has anyway one single return value proper. So, even if extra outlets are declared, *bach.eval* has one main outlet, distinct from them and placed at their right, from which the actual return value of the computation is output before all the extra outlet data, which are subsequently output in right-to-left order (regardlessly of the order in which the pseudovariable assignment happens during evaluation). The rationale behind the choice of moving the main outlet to the right is that, if extra outlets are present, the 'true' return value is customarily not used in the patch, thus not deserving the most important, leftmost outlet. Not less importantly, in this way `$o<n>` refers to the actual nth outlet, not the (n+1)th one.

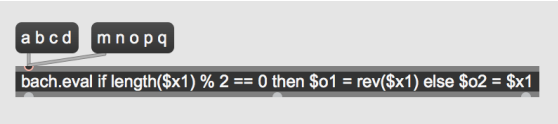

Figure 2. An example of a *bach.eval* with multiple outlets. As the highest-rank outlet pseudovariable declared is `$o2`, two extra outlets are created at the left of the main outlet. At the end of the computation, the evaluation result is output from the rightmost, main outlet (for details on the return value of conditionals, see subsection 5.4). Then, if the condition is true (that is, if the number of elements of the incoming *llll* is even), the reversal of the incoming *llll* is output from the leftmost outlet, otherwise the incoming *llll* is output unmodified from the middle outlet.

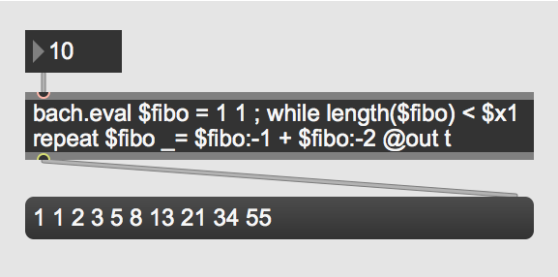

Figure 3. An example of *bach.eval*, computing a Fibonacci sequence up to a given term, something that would typically require a fairly complicated patch. This example contains some language features that have not been described yet: `$fibo:-1` and `$fibo:-2` respectively extract the last and one-but-last element of the *llll* contained in the `$fibo` variable; `_=` is an assigning operator that appends new elements at the end of the *llll* contained into a variable (in this case, it appends the sum of the two last elements to the already computed sequence). `@out t`, on the other hand, is not a language feature, but an attribute of the *bach.eval* object (and virtually all the other *bach* objects) whose precise meaning is discussed in [1]: suffice to say that it is required for outputting the *llll* produced by *bach.eval* in a format that the message box below can understand.

5.4 Conditionals

In *bell* there is no dedicated boolean type. Thus, the following convention is used in constructs requiring a boolean value: values of 0 (that is, integer 0, floating-point 0., rational 0/1 and pitch C0) and *null* are considered false; any other value is considered true.

The main conditional construct in *bell* is `if ... then ... else`. It returns the value of the evaluated branch, chosen according to the provided conditional expression, or *null* if the condition is false and no `else` clause is provided.

5.5 Loops

Two looping constructs are implemented in *bell*: the unbounded `while` iteration, repeating the evaluation of an expression as long as a given condition is true, and the `for` loop, iterating depth-first over one or more *llll*s according to a rather large set of options, including the abilities of limiting the traversal depth and setting an additional condition for prematurely stopping the iteration. Both loop constructs return the value of the loop body at the last iteration performed, or *null*.

5.6 Functions

As hinted at above, *bell* provides a set of built-in functions performing a wide array of standard operations on lists, such as reversal, rotation, search, transposition, sorting and so on. All functions in *bell* have return values, although some, such as `print()`, are typically called for their side effects. Moreover, users can define their own functions, which can be called with exactly the same conventions as built-in ones.

5.6.1 Function calls

Built-in functions have actual names whereas, strictly speaking, all user-defined functions are anonymous. In both cases, functions, from the point of view of *bell*, are technically elements of *llll*s (usually each being the only element of its containing *llll*). This means that, from a formal point of view, `cos(0)` means 'pass 0 to each element of the preceding *llll*, whose only element is the cos function'. Thus, `(sin cos)(0)` would be a perfectly legitimate construct, returning *0. 1.*. Performing a function call upon an *llll* element that is not a function is possible, but it will return *null* and print a warning in the Max console. [13]

We shall discuss below how user functions can be defined but, for the sake of simplicity of this first implementation, we did not introduce an explicit mechanism to name them: variables are used to refer to them instead. This means that, once a function has been defined and assigned to the variable `$myfunc`, it is possible to call it with, e.g., `$myfunc(0)`. If a single-assignment rule is assumed for these variables, there is no noticeable difference between *bell* and the usual functional coding style.

All function arguments have defaults, so no argument is mandatory. On the other hand, if the default is *null*, calling a function without providing enough arguments (e.g., `sin()`) will return *null*. Multiple arguments are separated by commas (e.g., `pow(2, 3)`). Moreover, all parameters are named, so arguments can be given out-of-order using their names (e.g., `pow(@exponent 3, @base 2)`: this is especially useful in the case of functions calls in which the user wants to change only a few parameters from their defaults.

5.6.2 User functions

User functions are defined by the `->` operator, preceded by the comma-separated parameter names (and, optionally, their respective defaults) and followed by the actual function body, e.g., `$x, $y = 1 -> $x * $y`. This is a function literal, not differently from `1` being an integer literal. A function defined in this way can be called directly, or, as hinted at above, be assigned to a variable and called through it: this is the only general way for a function to be reusable in different parts of a program. Functions not assigned to variables are typically passed to other functions as lambdas.

5.6.3 Scope of local variables

Local variables have lexical scoping and dynamic extent: this means that their lifetime is not restricted to its scope

[13] A syntactical subtlety is at play here: the fact that `cos(0)` is treated as a function application, rather than the concatenation of the *cos* function and the *0* literal surrounded by an unnecessary, but legal, pair of parentheses, stems from the fact that there is no whitespace between the function name and its argument list: `cos (0)` would have been considered the other way. Although we are aware that this choice may appear a not very elegant one, it seemed to us the simplest way to allow parentheses to be used as both precedence and function application operators (thus retaining compatibility with the *expr* family), while preserving the fundamental principle of implicit concatenation.

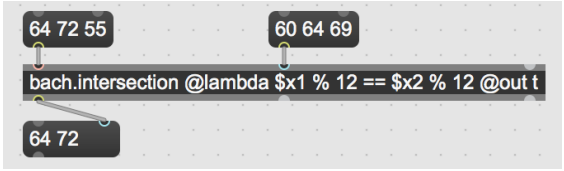

Figure 4. The *bach.intersection* object returns the intersection of two sets according to a custom equality comparison defined by the snippet of *bell* code `$x1 % 12 == $x2 % 12`, called by the intersection algorithm every time it needs to compare a pair of elements from the incoming *llls*. In this context, `$x1` and `$x2` do not refer to the object inlets, but more generically to the data passed to the called *bell* function by the caller.

of definition, and they can be accessed, although indirectly, from another scope. The classical example is the capture of a local variable in a function definition, the function being used elsewhere. Thus, in a scenario like

```
$a = 1 ; F = $x -> $a += $x ; F(10)
```

the value of local variable `$a` will be modified by the function referred to by `F`. The function can be called anywhere else in the program, through the global variable `F`, leading to the assignment of the local variable `$a`. This capture mechanism is especially useful for specifying recursion (see below) and simplifying the passing of extra parameters to anonymous lambda functions.

Function arguments hide local variables with identical identifiers. Thus,

```
$a = 1 ;
($x, $a = 10 -> $a += $x)(20) $a
```

will return *30 1*, as the value of the `$a` variable visible in the outer scope has not been affected.

5.6.4 Recursion

As hinted at before, dynamic scoping makes it straightforward to implement recursive functions:

```
$fact = ($x -> if $x <= 1 then 1
               else $x * $fact($x-1))
```

will work because the `$fact` variable, containing the recursive function, is visible in the function itself.

6. IMPLEMENTATION DETAILS

6.1 Code compilation and execution

As the host object receives the source code of a *bell* program, it translates all the defined functions (including the main function, that is, the 'main' body of the program) to separate abstract syntax trees, each associated to a symbol table with its arguments.

In the case of *bach.eval*, code is executed when the object receives data in its leftmost inlet; other objects may execute the code according to their own principles of operation (see, e.g., fig. 4). In any case, code execution consists of the traversal of the abstract syntax tree of the main function, from which other functions are called when necessary.

The interpreter and runtime engine of *bell* are written in C++ (with the help of a Flex/Bison parser) and linked to the C-based *bach* and Max APIs. Although not possible yet, it should not be difficult providing developers with a further *bell* API allowing them to add new built-in functions to the language. We are not sure, on the other hand, of the practical interest of such a possibility.

6.2 Efficiency

Besides the *llll* management (which was already implemented in the *bach* API), the function call procedure is easily the most complex task performed at runtime: dynamic scoping and management of arguments, their defaults and the two ways of passing them (by position and by name) delegate to run time many operations that could otherwise be performed during compilation. On the other hand, we believe that, although computational speed is surely not to be overlooked, a high-level system such as the one we propose should generally favor ease and flexibility of use rather than sheer efficiency. For this reason, we think that the convenience of a richer function calling mechanism outweighs the performance penalty imposed by having to give up some possible compile-time optimizations.

An important underlying feature of *bell* is that, although variables are mutable, *llll*s are inherently immutable. This means that users never need to explicitly clone *llll*s, as all the cloning operations are performed transparently. Although this may slow down the computation unnecessarily, we think it is another case in which ease of use outweighs the additional work required to the machine.

Even taking into account these trade-offs, substantial work is still needed to improve the efficiency of *bell*. As of now, the speed of execution of *bell* programs ranges from comparable to that of corresponding *bach*-based Max patches to significantly slower. At least two important optimizations are possible: avoiding the generation of intermediate *llll*s during evaluation (this can be achieved through the *deforestation* algorithm [5]), and implementing scalar data types, transparent to the user but used internally in place of singleton *llll*s. Both optimizations pose no major technical challenges, but they would require an important development effort, whereas the goal so far has been to produce a reasonably well-tested, working system, something that has seemingly been achieved. [14]

7. FUTURE WORK

Although the framework in which this will happen is not clear yet, there are plans for extending *bell* in a few essential directions.

Firstly, it will be useful to implement a number of optimizations, such as tail-call optimization, dead code elimination or constant folding, plus, of course, the ones described in the above section. These may increase dramatically the memory and speed efficiency of the language runtime.

One of the original plans about *bell* was to allow it to directly access for both query and modification the internal, non-*llll* data structures of *bach.roll* and *bach.score*, the two main objects of the *bach* library devoted to the representation of musical scores. Such a feature would require the implementation of at least a rudimentary object system for *bell*: although some pieces of infrastructure for this are already in place, it would be a massive undertaking, also

[14] At the time of writing, a version of *bell* with all the features described in this article and more has been thoroughly tested internally and distributed to a group of beta testers of the *bach* library. We plan to release *bell* in a few weeks within the official version 0.8.1 of *bach*. It is worth noting that *bell* will be distributed under an open source license, most likely LGPL 3.0.

requiring the involvement of Daniele Ghisi (who develops the *bach* system alongside author Andrea Agostini).

A smaller, but not less important, feature would be the addition of new built-in functions, and possibly the implementation of a sort of 'standard library', consisting of basic functions more apt to be represented in *bell* itself than in its C++ engine. This is probably something that will happen naturally, both before and after the first release of the language.

8. RELATED WORK

The idea of augmenting Max with textual coding capabilities in *ad-hoc* domain-specific languages is not new.

The venerable FTM library for Max [15] contains an object, *ftm.mess*, capable of constructing Max messages containing mathematical expressions, variables and more, and calling methods on other FTM objects. Although a powerful replacement for Max's message box, it is not (and is not meant to be) a 'real' programming language, as it does not allow conditionals, iterations and user-defined functions. On the other hand, *ftm.mess* was surely an important inspiration in the early conception of *bell*.

The *o.dot* project [16], a comprehensive system for manipulating OSC messages in Max developed at CNMAT, contains the *o.expr.codebox* module that implements a quite rich 'programming language in a box', not so differently from *bach.eval*.

Antescofo [6] is a large research project whose center is a programming language for description and scheduling of musical events, available, among the other things, as an object bringing a full implementation of the language into Max. The main, central notion in *antescofo* is time, which is represented through very rich semantics and data structures: on the one hand, this sets it apart from *bell*, which has no notion of time whatsoever; on the other hand, as both languages focus on the symbolic representation of music and musical scores, it might be interesting exploring possible ways to put them in relation with each other.

GenExpr, a proprietary language that can be used for programming pixel shaders, DSP algorithm and control-rate computation, is respectively exposed by the *jit.gl.gen*, *gen~* and *gen* objects, included in the official Max distribution. GenExpr programs really lie at the intersection of graphical and textual coding, as they can be written as traditional code or automatically generated as translations of visual graphs drawn in a dedicated environment, similar but distinct from the Max patch. GenExpr has number crunching as its main goal: its only data type is the floating-point number, and its only data structure is the array of floats.

Although based upon an almost opposite approach, Julien Vincenot's remarkable work of building a communication layer between *bach* and the SBCL Common Lisp compiler should be mentioned here: Lisp code for manipulating *llll*s, 'transliterated', so to speak, to Lisp lists, is generated in Max with the help of *bach* objects and run in SBCL in a separate process, after which the results are sent back to Max via a text file. [7]

Whereas each of these tools, including *bell*, has its own focus and goals, we find it interesting to remark how the problem of extending Max through textual coding has been tackled in different ways in relation to different projects.

[15] `http://ftm.ircam.fr/index.php/Download`

[16] `https://github.com/CNMAT/CNMAT-odot`

9. CONCLUSIONS

We have presented a new programming language, named *bell*, meant to be used in the larger context of the *bach* library for Max and conceived as a substantial extension of some features already present in Max itself. We have discussed the motivations behind the very idea of implementing it, as well as its overarching design principles and some basic aspects of its syntax and inner workings.

We wish to add that, for a project like this one to have some relevance, it is important to consider how it can be disseminated. We hope that the *bach* user base will respond favorably to it, and that teachers mentioning *bach* in their computer music classes will find it interesting to give an introduction to *bell* as well. It is worth adding that the *bach* package includes a complete, pedagogical (albeit not formal) documentation of the language, accompained by a substantial number of examples.

Acknowledgments

The *bell* programming language has been conceived and developed by Andrea Agostini with support from an *mrc* residency grant at IRCAM, under the supervision of Jean-Louis Giavitto who also gave essential contributions to the definition of the language syntax and provided a sound theoretical framework for it. We wish to thank Greg Beller, Arshia Cont, Daniele Ghisi and Emmanuel Jourdan for their invaluable help and support, without which this project could never have seen the light.

10. REFERENCES

[1] A. Agostini and D. Ghisi, "A Max Library for Musical Notation and Computer-Aided Composition," *Computer Music Journal*, vol. 39, no. 2, pp. 11–27, 2015.

[2] P. Desain *et al.*, "Putting Max in Perspective," *Computer Music Journal*, vol. 17, no. 2, pp. 3–11, 1992.

[3] M. Puckette, "Max at Seventeen," *Computer Music Journal*, vol. 26, no. 4, pp. 31–43, 2002.

[4] A. Agostini and D. Ghisi, "Pitches in bach," in *Proceedings of the International Conference on Technologies for Music Notation and Representation – TENOR'18*, Montreal, 2018, pp. 128–137.

[5] P. Wadler, "Deforestation: Transforming programs to eliminate trees," in *European Symposium on Programming*. Springer, 1988, pp. 344–358.

[6] A. Cont, "ANTESCOFO: Anticipatory Synchronization and Control of Interactive Parameters in Computer Music." in *International Computer Music Conference (ICMC)*, 2008, pp. 33–40.

[7] J. Vincenot, "LISP in Max: Exploratory Computer-Aided Composition in Real-Time," in *Proceedings of the 2018 International Computer Music Conference*, Shangai, 2018, pp. 87–92.