Takt: A read-eval-play-loop interpreter for a structural/procedural score language

Satoshi Nishimura
University of Aizu
nisim@u-aizu.ac.jp

ABSTRACT

A new language for describing musical scores as well as its interpreter is developed. The language allows a concise description of note and chord sequences, and at the same time, it provides rich programming functionalities with C-like syntax, which are useful for algorithmic composition. Representing structures in music such as repetitive occurrences of a common phrase or its variation is supported with macros and phrase transformation modules. The interpreter enables us to execute its program code interactively with a read-eval-play loop. The interpreter can also be used for the real-time processing of MIDI events coming from input devices. The language is extensible in that C functions can be called from its program code.

1. INTRODUCTION

Composers often use instruments during composition for examining their ideas including phrases, algorithmic rules, or a combination of the both. A problem in such a process is that some compositions are hard or rather impossible to be played manually. Music educators may also face this problem for demonstrating algorithmic pieces of music to students.

One of the efficient solutions to the above problem is to provide a shell-like interactive environment based on a music language. With such a tool, users can repeatedly try a different idea and examine its resulting sound instantly. In the field of general-purpose languages, this type of a command interpreter is called a read-eval-print loop (REPL); however, it should better be called a read-eval-*play* loop in our context.

To make such a REPL environment comfortable, the language should be designed so that frequently-used command patterns are simple and easily-typeable. For example, it is desirable that a sequence of notes can be played just by typing their pitch names like '`c d e`' in the command line. It is also demanded that algorithmic descriptions can be easily mixed with such direct descriptions; in particular, the use of delimiter symbols to switch between direct and algorithmic descriptions is discouraged.

So far, numerous score description languages have been proposed [1, 2, 3, 4, 5, 6, 7]; however, to the best knowledge of the author, no languages are satisfactory for the aforementioned purpose. Some of the languages (e.g., [4, 5]) are constructed by extending existing general-purpose languages while others are not. The formers may have an advantage that existing resources developed in the base languages can be utilized. However, it is difficult to allow that a bare sequence of pitch names constitutes a complete program for playing notes. For example, if the base language is LISP, we need at least parentheses like '`(c d e)`' (here '`c`' is a function).

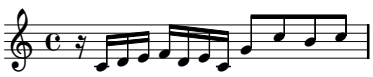
There are many score languages designed with the latter approach (i.e., designed from scratch). Some of them are shown in Figure 1. Nevertheless, such languages to date do not provide sufficient features for algorithmic composition per se. Although embedding another general-purpose language into such a language might be a solution (for example, Lilypond [6] allows embedding Scheme in its code), such an approach will require additional delimiters for switching between two languages. As a result, a combination of note-by-note and algorithmic descriptions tends to be complicated.

This paper proposes a new language named Takt which provides a simple top-level description of note and chord sequences yet integrates rich programming functionalities for algorithmic composition per se. The language is originated from the author's previous work [8]; however, its syntax is thoroughly re-designed. By a newly-developed interpreter, called a Takt interpreter, the language is translated into a stream of MIDI events in real time. The language resembles other C-like languages and thus enables smooth migration for their users. The language is extensible in that C functions can be called from its source code, by which utilizing existing libraries written in C/C++ is possible.

This paper is organized as follows. In Section 2, the proposed language is explained. Section 3 describes the developed interpreter together with its performance evaluation. Section 4 discusses other music programming tools related to this work. Section 5 concludes this paper.

2. THE LANGUAGE

This section first introduces a core language for describing note/chord sequences and later explains how the actual language extends the skeletal language.



SCORE [1]
```
P2 RHY/-16/16 X 7/8 X 4;
P3 NOTES/R/C4/D/E/F/D/E/C/G/C5/B4/C5;
```

DARMS (The Note-Processor Dialect) [3]
```
RS -1 0 1 2 0 1 -1 3E 6 5 6
```

MML [3]
```
L16RCDEFDECL8G>C<B>C
```

ABC [7]
```
L:1/8
R z/C/D/E/ F/D/E/C/ GcBc
```

Lilypond [6]
```
r16 c' d' e' f' d' e' c' g'8 c'' b' c''
```

Takt
```
{r c d e f d e c}\\ {g ^c b ^c}\
```
or
```
l16 r c d e f d e c g~ ^c~ b~ ^c~
```

Figure 1. Comparison of Score Description Languages.

2.1 The Core Language

The syntax of the core language with a start symbol s is as follows:

Score:	s	$::= \varepsilon \mid s\,p \mid s\,x = e$
Phrase:	p	$::= c \mid \{s\} \mid [s] \mid p\,m \mid p \mid f \mid p \,@\, p$
Expression:	e	$::= n \mid x \mid e + e \mid e * e \mid \ldots$
Pitch:	c	$::=$ `c` $\mid$ `c#` $\mid$ `d` $\mid \ldots \mid$ `^c` $\mid \ldots \mid$ `r`
Modifier:	m	$::=$ `*` $\mid$ `\` $\mid$ `~` $\mid \ldots$
Transformation:	f	$::=$ `Transpose(`e`)` $\mid$ `Inverse(`e`)` $\mid$ `Retrograde()` $\mid \ldots$

where n and x represent a constant and a variable, respectively. Figure 2 shows an example of music written in this language.

The whole score is composed of phrases and assignments. Each phrase corresponds to a fragment of music. Such a phrase consists of a note, a braced block, or a bracketed block, optionally followed by phrase-modifying notations.

Each note is represented by a user-definable pitch name, which is by default an English pitch name. The attributes of the note such as length (aka note value) or velocity (aka dynamics) are obtained from the current *context*, which is a map from a set of variables to their values. Using the assignment $x = e$, the value of x in the context is set to the value of the expression e.

The braces '`{`' and '`}`' create a new copy of the current context and execute the score therein with the new context. Then, the context is resumed to the original one. For example, '`v=80 c {v=90 d e} f`' will play a C note with velocity 80, D and E notes with velocity 90, and then an F note again with velocity 80. The brackets '`[`' and '`]`' also save and restore contexts, and in addition, they perform phrases therein in parallel. They are used for representing chords or polyphony.

When a phrase is accompanied with a modifier such as '`*`', the context for the phrase is temporarily modified. For example, the '`*`' modifier doubles the length attribute of notes, while the '`\`' modifier halves it. Hence, '`c\\`' represents a sixteenth note with the C pitch, since the default length is a quarter note. The '`~`' modifier adds two lengths like a tie.

```
[
  { /* the first voice */
    { r c d e f d e c }\\ {g ^c b ^c}\
    { ^d g a b ^c a b g }\\ {^d ^g ^f ^g}\
                      ⋮
    { ^c bb a g f a g bb }\\
    {a b ^c e d ^c f b}\\
    [e g ^c]**
  }
  { /* the second voice */
    o=3
    r* {r c d e f d e c}\\
    {g _g}\ r {r g a b ^c a b g}\\
                      ⋮
    {e c d e {f d e f}\ g _g}\
    [_c c]**
  }
]
```

Figure 2. The beginning and ending parts of the J. S. Bach two-part invention BWV 772 written in the core language.
Each note or rest is represented by an alphabetical letter. The '`*`' and '`\`' signs specify note length. The '`^`' and '`_`' signs adjust octaves. The '`o=3`' assignment lowers the default octave.

A phrase can be transformed with programmable rules. For example, the '`Transpose(`e`)`' transformation adds the value of e to the pitch (in semitones) of each note. [1] The '$p \mid f$' syntax applies the transformation f to the phrase p. The currently available pre-defined transformations are listed in Figure 3.

The '$p \,@\, p$' syntax provides a special form of transformation that applies the rhythmic structures as well as velocity fluctuations in the second phrase to the first phrase. [2] For example, '`{e f g a}@{c* c\}`' is equivalent to '`{e* f\ g* a\}`'. This feature is useful when one wants to share rhythmic structures or expressive controls across phrases with different pitches.

2.2 The Actual Language

In addition to the functions of the core language, the actual language includes the following features.

2.2.1 Programming Features

Most functionalities found in other general-purpose interpretive languages like Perl [10] are also provided in Takt. They include loops, conditional constructs, arithmetic/logical

[1] Transformation is also found in early notation languages like SCORE [1], although SCORE's transformation was not programmable.

[2] Similar notion has been proposed as the *slap* structure [9] in the field of musical knowledge representation.

`Transpose`	Chromatic or diatonic transposition
`Invert`	Chromatic or diatonic pitch inversion
`ConvertScale`	Pitch mapping between two scales
`Retrograde`	Retrograde (playing backwards)
`TimeStretch`	Time scaling
`AddNotes`	Adding new note(s) for each note (for octaving or harmonizing)
`Modify`	Event modification using assignments
`SelectIf`	Event selection with conditions
`Clip`	Time-range cutting
`Swing`	Simple time deformation
`Quantize`	Time quantization
`Arp`	Arpeggio effect
`ModifyChords`	Modifying each note of chords using assignments
`Grace`	Adding grace note(s) for each note
`Roll`	Drum-rolling effect
`Trill`	Trill effect
`Tremolo`	Tremolo effect

Figure 3. List of available transformations.

operators, classes, macros and high-order functions. Supported data types include integers, floating-point numbers, rational numbers, arrays, strings, and associative lists (aka hashes). The language syntax is designed so that statement separators or terminators (like the semicolons in the C language), which often bring confusion for beginners, are not required.

In the actual language, statements for programming and the phrases in the core language are comparable syntax elements, and therefore, programming constructs can be arbitrarily mixed with the musical descriptions. For example, the `for` statement of the language can be used for repeating a phrase as follows:

```
for(i,1,4) { c e g }
```

Meanwhile, the same syntax can be used in numeric programming like the following:

```
var sqrsum = 0
for(i,1,10) { sqrsum += i * i }
```

Thus, the language avoids redundant learning efforts.

2.2.2 *Macros*

It is possible to define a phrase as a macro and reuse it later. For example, the following example defines the phrase as a macro named '`x`'.

```
var x = {c d e c e f g*}
```

Once a macro is defined, it can be invoked simply by placing its name. For example, a two-voice canon can be described as

```
[x {r** x}]
```

where '`r**`' represents a whole rest. Moreover,

```
[x {r** x}|Transpose(6)]
```

```
var p1 = {c d e f d e c}\\
[
  { /* the first voice */
    r\\ p1 {g ^c b ^c}\
    ^d\\ p1|Transpose(7) {^d ^g ^f ^g}\
                        ⋮
    ^c\\
    p1|Invert(c,Scale.major(c))|Transpose(10)
    {a b ^c e d ^c f b}\\
    [e g ^c]**
  }
  { /* the second voice */
    o=3
    r* r\\ p1
    {g _g}\ r r\\ p1|Transpose(7)
                        ⋮
    {e c d e {f d e f}\ g _g}\
    [_c c]**
  }
]
```

Figure 4. Another description of the invention written in the actual language.

will construct a polytonal canon by using the transformation. Figure 4 shows how macros and transformations can be used for representing the structure of music.

Macros in Takt are treated as objects just like functional languages considering functions as first-class objects. For example, they can be stored into an array as below.

```
var phrases = %[{c d e c}, {e f g e},
                {d e f d}]
```

The '`%[`' and '`]`' signs construct an array. The macros stored in the array can, for example, be picked randomly and invoked as follows:

```
phrases[irand(0,2)]
```

2.2.3 *Functions*

Takt provides named and unnamed functions that are treated as objects. The body of such functions can be described either in Takt or in an external language like C++.

Functions in Takt can be used for defining not only computing tasks but also parametric musical phrases. For example, the following function defines a simple accompaniment pattern.

```
def waltz(p1:quote, p2:quote) {
  p1 p2 p2
}
```

After this definition, '`waltz(c, [e g])`' will be equivalent to '`c [e g] [e g]`' (the '`quote`' directive makes each argument received as a macro object).

2.2.4 *Repetition*

In addition to loop constructs such as `for` and `while`, a handy way for repetition is provided. When a phrase is

```
def Transpose(semitones:number) {
  return %{
    class = Effector,
    def eventAction(ev:Event) {
      ev.n += semitones
      put(ev)
    }
  }
}
```

Figure 5. Example of user-defined transformation.

followed by '`@`' with an integer i, the phrase is repeated i times (e.g., '`c@4`'). Infinite loops are indicated by '`@@`'; for example, '`{c d}@@`' indefinitely plays alternating C and D notes.

2.2.5 Event Buffers

An event buffer is a data structure storing a list of time-stamped events (e.g., note-on and note-off events) together with playing duration. Event buffers are convenient for manipulating phrases in a non-streaming way. An event buffer containing the events generated by a phrase p is constructed by an expression formed '`${`p`}`'. Transformations can be applied to event buffers by the '`|`' operator; for example,

```
var buf = ${c} | Transpose(2)
```

first creates an event buffer containing the events of the C note and then transforms it into a new event buffer containing events of the D pitch, which is assigned to the variable `buf`. The contents of an event buffer can be played just like a macro (e.g., after the above assignment, '`buf`' will play the D note) or can be written to a standard MIDI file by calling a library function.

2.2.6 User-Defined Transformation

The process of phrase transformation is user-programmable. It is given by defining a function called for each input event optionally with initialization and finalization functions. Figure 5 shows an example of such definition for transposition.

Transformations are categorized into streaming and non-streaming types. In streaming transformation, input events must be processed in chronological order, while the non-streaming (aka buffered) type allows random access of input events by using the event buffers discussed in Section 2.2.5. Streaming transformation is more favorable because it can be applied to infinite phrases or event streams from input devices. Nevertheless, some kinds of transformation such as retrograde (reverse play) can only be implemented with the non-streaming type.

Streaming transformations are further classified into two categories: non-rewinding or rewinding. Non-rewinding transformations (also called causal transformations) never decrease the time-stamp values of events. Transformations like pitch conversion belong to this category. On the other hand, rewinding transformations possibly decrease the time-stamp values of events to some extent. They are useful, for example, for implementing time-quantizing transformation. Rewinding transformations can still be applied to infinite phrases; however, when they are used to a real-time event stream from an input device, the reduction of the time stamps becomes invalid.

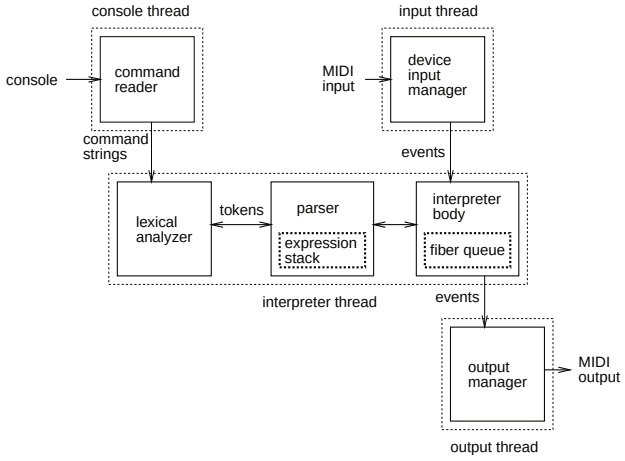


Figure 6. Structure of the interpreter.

3. THE INTERPRETER

The developed interpreter operates either in REPL mode or source-file mode. MIDI events generated by the interpreter are transmitted to MIDI output devices or stored to MIDI files. It is written in the C++ language and its current version consists of approximately 20,000 lines. Currently, the interpreter runs under Windows, Mac OS X, and Linux platforms; furthermore, porting to other platforms should be straightforward as long as the `pthread` thread library working with a timer with sufficient resolution is available.

3.1 Organization

Figure 6 illustrates the structure of the interpreter. The command reader receives program code from the console and passes them to the lexical analyzer, in which they are converted to tokens. The parser analyzes the syntax of the token stream and evaluates expressions and statements in it. When the parser encounters an invocation of a macro, it expands the macro by pushing its definition (a list of tokens) back to the lexical analyzer. The interpreter body, maintaining contexts and function-calling stack frames, generates events in response to requests from the parser. The output manager stores those events in a priority queue and sends them to the MIDI output devices according to their time-stamp values. The device input manager buffers events from the MIDI input device.

To handle macros as first-class objects, the Takt interpreter does not use an internal intermediate language which many language interpreters employ for improving their speed. In order to overcome the performance penalty due to that, and also, to implement fibers described in the next section, the parser is hard-coded without using parser generators.

The interpreter uses four operating-system threads. The command reader is executed in the original thread when the interpreter process is started. The lexical analyzer, the parser, and the interpreter body are run in a separate thread for allowing the background execution of input program code. The device input and output managers also use separate threads for improving the time accuracy of MIDI input/output.

Garbage collection is one of the important issues in language processing. To process events from the MIDI input device without pauses in the interpreter thread, traditional mark-and-sweep garbage collectors are not satisfactory. In the Takt interpreter, the incremental garbage collection algorithm proposed by Dijkstra et al. [11] is implemented.

3.2 Fibers and Scheduling

The concurrent execution of program code indicated by the '`[`s`]`' construct is realized using language-level fibers (aka coroutines). Each fiber keeps its own parser state, expression stack, and function-calling stack. Fiber switching occurs only when the execution of the current fiber is blocked, when a new fiber is created, or when such switching is explicitly indicated in the program code. The use of fibers offers better efficiency and easier mutual exclusion than using operating-system threads.

The scheduling of fibers is controlled by scheduling time ST_i associated with each fiber (here, i represents a fiber identifier). A priority queue of fibers is maintained in the interpreter body, and when fibers are switched, a fiber having the earliest scheduling time is picked for the candidate for the next execution. The candidate fiber j is executed immediately if $ST_j \leq GT$ where GT is the global time based on the operating system clock or is blocked until the condition is met otherwise. The time in the interpreter is represented in ticks (480ths of a quarter note), and mapping between ticks and seconds is maintained in the output manager.

When a transformation is applied to a phrase, a fiber is created. In order to realize the rewinding transformation described in Section 2.2.6, the fibers for the phrase and the transformation process need to be executed in advance relatively to other fibers. This "look-aheading" capability is controlled by a per-fiber parameter AD_i called *aheadness*, which determines how the execution of fiber i should be scheduled earlier than a base fiber β_i.

The scheduling time ST_i is calculated as follows. The interpreter constructs a weighted directed forest in which each vertex corresponds to a fiber and each arc is (β_i, i) with its weight being AD_i. Then, the weighted height AC_i of each vertex is calculated. The height is called *cumulative aheadness*. Finally, the scheduling time is calculated by

$$ST_i = LT_i - AC_i$$

where LT_i is the local time of fiber i.

3.3 Performance

Although the speed of the interpreter may not be a primary concern of event-level (i.e., not signal-level) music languages, it may have an impact on huge length music or pieces using time-consuming algorithms.

	Takt	Lilypond 2.18.2
Random note generation	7.4 sec.	9.4 sec.
Note generation based on the Fibonacci numbers	6.8 sec.	3.7 sec.

Table 1. Comparison of the interpreter performance.

Table 1 shows calculation time for simple benchmarking programs, comparing with Lilypond [6] with its typesetting capability disabled. The first benchmark generates a MIDI file containing 100,000 notes with random pitches on the C major scale. The second benchmark calculates the first 30 Fibonacci numbers using the naive recursive algorithm, maps them to note numbers on the C major scale, and generates a MIDI file containing 30 notes. The Lilypond input files contain embedded Scheme code for implementing such algorithms. All the programs were executed on the Intel Core2 Q9550 2.83GHz processor under the Windows Vista operating system.

As seen from the result, the Takt interpreter is faster in simple note generation; however, it is 1.8 times slower in the benchmark requiring intensive algorithmic calculation. This is partly because the Takt language pursues maximum flexibility with the power of macros, while Scheme better concentrates on performance. The author believes that, in event-level music applications, the superiority in description capability outweighs the performance penalty.

3.4 Supporting Tools

In addition to the interpreter, a translator from MIDI files to Takt and a language-specific editor based on Emacs are provided. The translator analyzes chords, polyphonic structures, and continuous parameter changes in the input MIDI file and outputs equivalent Takt code, which can be edited and converted back to a MIDI file by the interpreter. The Emacs interface enables us to enter pitch names from a MIDI keyboard and to play a described score with a feature of a score-tracking cursor for indicating the current playing position. The score-tracking cursor helps non-experts to understand the Takt score description.

4. RELATED WORK

Interactive programming tools for algorithmic composition or real-time MIDI processing have been developed over the decades. Some of them are well matured and widely accepted as standard tools. They can be categorized into two types: visual language based and textual language based.

Max [12], Pd [13] and OpenMusic [14] belong to the former category. In general, visual languages are easy to learn and convenient for creating interactive control programs using graphical components such as buttons and sliders. However, developing large-scale programs in visual languages is generally considered to be difficult and that is why textual languages are mainly used today for developing software applications as well as hardware systems. As a remedy for this problem, each tool provides a way

for creating new graphical objects using an external textual language (for example, JavaScript in Max or LISP in OpenMusic). However, those external languages are weak in score description, and thus, they are not convenient for defining phrases, transforming them, and organizing them with an algorithmic flow.

SuperCollider [15] is a popular tool based on a textual language. It mainly focuses on audio synthesis; however, it also has capabilities for MIDI processing. The language is a newly-designed one supporting object-oriented features and coroutines. Nevertheless, it does not support note-by-note description like '`c d e`', and therefore, the seamless fusion of note-by-note and algorithmic descriptions as provided in Takt is not possible.

5. CONCLUSION

This paper described an interactive command-line environment for composers, educators, and researchers. In the environment, note-by-note direct description and algorithmic representation are unified in one language and therefore the system is considered to be optimal for compositions with the mixed use of non-algorithmic and algorithmic approaches. In future, I would like to extend this project to support graphical interfaces such as a piano-roll editor for event buffers. I would also like to investigate the possibilities of applying this environment for other purposes such as real-time control of robots or illumination.

6. REFERENCES

[1] L. Smith, "Score — A Musician's Approach to Computer Music," *Journal of Audio Engineering Society*, vol. 20, no. 1, pp. 7–14, 1972.

[2] C. Roads, *The Computer Music Tutorial*. Addison-Wesley, 1996, ch. 17.

[3] E. Selfrideg-Field, Ed., *Beyond MIDI*. MIT Press, 1997.

[4] H. Taube, "Common Music: A Music Composition Language in Common Lisp and CLOS," *Computer Music Journal*, vol. 15, no. 2, pp. 21–32, 1991.

[5] P. Hudak, T. Makucevich, S. Gadde, and B. Whong, "Haskore Music Notation - An Algebra of Music," *Journal of Functional Programming*, vol. 6, pp. 465–483, 1995.

[6] H.-W. Nienhuys and J. Nieuwenhuizen, "LilyPond, a system for automated music engraving," in *Proceedings of the XIV Colloquium on Musical Informatics*, 2003.

[7] C. Walshaw, "ABC Music Notaion," URL: `http://abcnotation.com/`.

[8] S. Nishimura, "PMML: A Music Description Language Supporting Algorithmic Representation of Musical Expression," in *Proc. of the 1998 International Computer Music Conference*, 1998, pp. 171–174.

[9] M. Balaban, "The Music Structures Approach to Knowledge Representation for Music Processing," *Computer Music Journal*, vol. 20, no. 2, pp. 96–111, 1996.

[10] L. Wall and R. L. Schwartz, *Programming Perl*. O'Reilly Media, 1991.

[11] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, "On-the-Fly Garbage Collection: An Exercise in Cooperation," *Communications of the ACM*, vol. 21, no. 11, pp. 966–975, 1978.

[12] M. Puckette, "The Patcher," in *Proc. of the 1988 International Computer Music Conference*, 1988, pp. 420–429.

[13] ——, "Pure Data," in *Proc. of the 1996 International Computer Music Conference*, 1996, pp. 269–272.

[14] G. Assayag, C. Rueda, M. Laurson, C. Agon, and O. Delerue, "Computer-Assisted Composition at IRCAM: From PatchWork to OpenMusic," *Computer Music Journal*, vol. 23, no. 3, pp. 59–72, 1999.

[15] S. Wilson, D. Cottle, and N. Collins, Eds., *The SuperCollider Book*. MIT Press, 2011.