Universidad Nacional de Quilmes

Escuela de Artes

LICENCIATURA EN MÚSICA Y TECNOLOGÍA

Director de Carrera: Esteban Calcagno

Programa de Investigación

CARTOGRAFÍAS ESPACIO-TEMPORALES Y ARTE SONORO

Director: Pablo Riera

Seminario de Investigación

REPRESENTACIÓN TEXTUAL DE ESTRUCTURAS MUSICALES Y ENTORNO DE SECUENCIACIÓN

Presentado por: Lisandro Fernández

Resumen

Se propone un lenguaje formal basado en texto plano, descriptivo y serializado, capaz de representar informacion musical. Contextualiza, un conjunto de utilidades cuyo fin es producir secuencias musicales en el estándar MIDI.

Mayo 2019

Buenos Aires, Argentina

Contendios

1. Resumen

El presente trabajo propone un contexto de producción musical puramente textual.

Son el producto de esta investigación un marco de patrones y relaciones gramaticales que posibilitan la representación sintáctica de información con significado musical; un léxico y una sintaxis que definen estructuras musicales contenidas en ficheros de texto serializado[1] y autodescriptivo.

Acompaña esta propuesta un entorno de herramientas, para interprete de línea de comandos.[2] Es otro aporte importante del actual desarrollo esta cadena de procesos que consume información suscrita a dicha representación; derivando

[1] Coombs, Renear y DeRose (1987)
[2] (n.d.) (2019b)

esta manipulación en la producción de secuencias de mensajes en el formato MIDI estándar.

La primera parte de este escrito esta dedicada a justificar el objeto de estudio, presentar los motivos de las interrogantes, plantear la necesidad alternativas, también se discuten antecedentes en codificación textual de información musical.

En la segunda sección se describe el método de ejecución, detallando el procedimiento de desarrollo.

La parte central de este trabajo versa sobre el vocabulario y relaciones que conforman la gramática propuesta, se explica como dicha representación habilita que la semántica musical pueda ser materia prima de esta serie de procesos y se despliegan el resultado de algunos ejemplos a modo de demostración.

Para concluir se proyectan algunas aplicaciones posibles en diferentes escenarios (trabajo colaborativo en simultaneo y a distancia, programación en vivo) y varias disciplinas (Inteligencia artificial, modelo condicional restringido archivología).

Completando el aspecto técnico de este trabajo se incluyen el código de los módulos desarrollados para la implementación.

2. Introducción

En esta sección inaugural se enmarca la investigación, argumentando la constricción principal, la adopción de un sistema de escritura como contenedor instrucciones y medio de interacción.

Seguido se repasan las necesidades que denotan la pertinencia de este estudio, aludiendo a requerimientos externos a satisfacer.

Para concluir esta introducción se tratan trabajos semejantes de cierta relevancia a este proyecto.

2.1. Justificación

En este apartado se repasan las ventajas principales del registro de información con enunciados textuales y del empleo del lenguaje como medio de entrada de instrucciones en escenarios generales.

2.1.1. Texto Llano

> "…our base material isn't wood or iron, it's knowledge. […]. And we believe that the best format for storing knowledge persistently is plain text. With plain text, we give ourselves the ability to manipulate knowledge, both manually and programmatically, using virtually every tool at our disposal." (Hunt y Thomas 1999)

Se listan las virtudes del texto plano y legible en contraste a la codificación binaria de datos[3] o cualquier otro tipo de operación que opaque la relación con lo representado

Mínimo Común Denominador. Potencialmente cualquier herramienta de computo puede operar información almacenada en texto plano. Es soportado en múltiples plataformas, cada sistema operativo cuenta con al menos un editor de texto todos compatibles hasta la codificación de caracteres.

Fácil de manipular. Procesar cadenas de caracteres es de los trabajos mas rudimentales que pueden ser realizados por un sistema informático.

Fácil de mantener. El texto plano no presenta ninguna dificultad o impedimento ante la necesidad de actualizar información o de realizar cualquier tipo de cambio o ajuste.

Fácil de comprobar. Es sencillo agregar, actualizar o modificar datos de testeo sin la necesidad de emplear o desarrollar herramientas especiales para ello.

Liviano. Determinante cuando los recursos de sistema son limitados como por ejemplo almacenamiento escaso, velocidad de computo restringida o conexiones lentas.

Seguro contra la obsolescencia, o compatible con el avance. Los archivos de datos en formatos legibles y autodescriptivos perduran por sobre otros formatos aun cuando caduquen las aplicaciones con las hayan sido creados.[4]

2.1.2. Interprete de Comandos

Se argumenta la conveniencia de prescindir de representaciones gráficas como canal de interacción con herramientas informáticas.

Estado operativo de un ordenador inicial. Eventualmente todos los sistemas operativos permiten ser utilizados a través de este acceso previo al gerente de escritorio.

Menor utilización de recursos. No depender de un agente de ventanas interviniendo entre el usuario y el sistema libra una cantidad considerable de recursos.

Una interfaz para diferentes aplicaciones. La estructura esperada de las instrucciones en esta interfaz *aplicación - argumento - recurso* (su analogía *verbo - adverbio - sujeto*) persiste para cualquier pieza de software. Dicha recurrencia elimina el ejercicio que significa un operar distinto para cada aplicación, favoreciendo un accionar semejante en contextos y circunstancias diferentes.

Tradición. Perdura por décadas como estándar durante la historia de la informática remitiendo a los orígenes de los ordenadores basados en teletipo.

[3] Hunt y Thomas (1999) Capítulo 3: Basic Tools (pp. 72-99).
[4] Leek (2017)

Resultados reproducibles. Si bien la operación de sistemas sin mas que la entrada de caracteres requiere conocimiento y entrenamiento específico, no considerar la capa que representa la posición del puntero como parámetros de instrucciones, permite que sean recopiladas en secuencias de acciones precisas, reutilizar estos guiones en diferentes escenarios y agentes diversos.

Encadenado La posibilidad de componer rutinas complejas de manipulación concatenando resultados con procesos.[5]

Gestión remota. Mas allá del protocolo en el que se base la negociación local/remoto la interfaz de linea de comandos es la herramienta de facto para administrar un sistema a distancia.

Productividad. Valerse de herramientas pulidas como editores de texto avanzados que gracias al uso de atajos (acciones complejas asignadas a combinaciones de teclas) evitan la alternancia entre mouse y teclado, lo cual promueve un flujo de trabajo ágil.[6]

Siendo estas razones de carácter general, las mismas aplican al propósito particular que implica este estudio.

2.1.3. Interface Digital para Instrumentos Musicales (MIDI)

De carácter especifico a la producción musical, en relación directa a este proyecto es menester acreditar la adopción de un formato en particular para codificar la capa que describe y gestiona la performance entre dispositivos.[7]

El animo por que las secuencias de control a producir satisfagan las condiciones requeridas para alcanzar compatibilidad con el formato MIDI estándar[8], está fundamentado por sus virtudes de ser y proyectarse ampliamente adoptado, soportado en la mayoría de los entornos y apoyado por la industria.[9]

Si bien es ágil y se planea compatible a futuro, cualidades que comparte con el formato de texto llano, es ineludible la desventaja que significa el empleo de cualquier sistema de codificación[10], intrínseca a la dificultad que impone para interpretar a simple vista la información cifrada, ofuscación que resulta en la dependencia de herramientas especificas para cualquier manipulación.

2.2. Motivación

Este proyecto plantea la necesidad de establecer un contexto y proveer recursos para un procedimiento rudimental pero a la vez ágil y flexible de elaboración

[5] Raymond (1999) Capítulo 1: Context, Apartado 1: Philosophy, Sub-apartado: Basics of the Unix Philosophy (pp. 34-50)
[6] Moolenaar (2000)
[7] Haus y Ludovico (2007)
[8] (MMA) (1996)
[9] Penfold (1992)
[10] Cifrado condicionante para el trasporte.

discursos musicales unificando la planificación de obra con la secuenciación MIDI.

Ademas pretende exponer las ventajas de la Interfaz de Linea de Comandos para operar herramientas informáticas a la comunidad de artistas, teóricos e investigadores.

Promover la adopción de prácticas consolidadas y formatos abiertos para representar, manipular y almacenar información digital.

Fomentar el trabajo colaborativo generando vínculos con y entre usuarios.[11][12]

2.3. Antecedentes

A continuación se describen algunos desarrollos que vinculan representación y manipulación de información musical: MuseData, Humdrum, MusicXML y MML; como ejemplo de un marco de programación basada en una sintaxis declarativa se cosideró Flocking.

2.3.1. MuseData

La base de datos MuseData[13] es el sistema de codificación principal del Centro de Investigación Asistida por Computador en Humanidades (CCARH) de la Universidad de Stanford. La base de datos fue creado por Walter Hewlett.

Los archivos MuseData tienen el potencial de existir en múltiples formatos comunes de información. La mayoría de las codificaciones derivadas definen sólo algunas de las las características incluidas en el master MuseData de codificaciones. El archivo MuseData está diseñado para soportar aplicaciones de sonido, gráficos y análisis. Los formatos derivados de las codificaciones musicales de MuseData que se distribución son: MIDI1, MIDI+ y Humdrum.

2.3.1.1. Organización de archivos MuseData

Los archivos MuseData están basados en ASCII y se pueden ver en cualquier editor de texto. Dentro del formato MuseData el número de archivos por movimiento y por trabajo puede variar de una edición a otra. Estos ficheros están organizados en base a las partes. Un movimiento de una composición es típicamente encontrado dividido en varios archivos agrupados en un directorio para ese movimiento.

2.3.1.2. La representación MuseData de información musical

El propósito de la sintaxis MuseData es representar el contenido lógico de una pieza musical de una modo neutral. El código se utiliza actualmente en la construcción de bases de datos de texto completo de música de varios compositores,

[11] Raymond (1997) Capítulo 11: The Social Context of Open-Source Software (p. 11)
[12] Yzaguirre (2016)
[13] Selfridge-Field (1997) Capitulo 27: MuseData: Multipurpose Representation

J.S. Bach, Beethoven, Corelli, Handel, Haydn, Mozart, Telemann y Vivaldi. Se pretende que estas bases de datos de texto completo se utilicen para la impresión de música, análisis musical y producción de archivos de sonido digitales.

Aunque el código MuseData está destinado a ser genérico, se han desarrollado piezas de software de diversos tipos con el fin de probar su eficacia. Las aplicaciones MuseData pueden imprimir resultados y partes para ser utilizadas por editores profesionales de música, así como también compilar archivos MIDI (que se pueden utilizar con secuenciadores estándar) y facilitar las búsquedas rápidas de los datos de patrones rítmicos, melódicos y armónicos específicos.

La sintaxis MuseData está diseñada para representar tanto información de notación como de sonido, pero en ambos casos no se pretende que la representación esté completa. Eso prevé que los registros MuseData servirían como archivos de origen para generar tanto documentos gráficos (específicamente de página) y archivos de performance MIDI, que podrían editarse como el usuario lo crea conveniente. Las razones de esta postura son dos:

- Cuando se codifica una obra musical, no es la partitura sino el contenido lógico de la partitura lo que codifica. Codificar la puntuación significaría codificar la posición exacta de cada nota en la página; pero nuestra opinión es que tal codificación realmente contendría más información que la que el compositor pretende transmitir.
- No se puede anticipar todos los usos a los cuales podrían darse estos datos, pero se pude estar bastante seguro de que cada usuario tendrá sus propias necesidades especiales y preferencias. Por lo tanto, no tiene sentido tratar de codificar información acerca de cómo debe verse una realización gráfica de los datos o cómo sonido que estos datos representan debe sonar.

Por otro lado, a veces puede ser útil hacer sugerencias sobre cómo los gráficos y el sonido deben ser realizados. Lo importante es identificar las sugerencias como un tipo de datos independiente, que puede ser fácilmente ignorado por software de aplicación o despojado enteramente de los datos. MuseData usa estas sugerencias de impresión y sonido en el proceso de generación de documentos de partitura y archivos MIDI.

2.3.2. Humdrum

David Huron creó Humdrum[14] en los años 80, y se ha utilizado constantemente por décadas. Humdrum es un conjunto de herramientas de línea de comandos que facilita el análisis, así como una sintaxis generalizada para representar secuencias de datos. Debido a que el conjunto de herramientas es de lenguaje de agnóstico. Muchos han empleado herramientas de Humdrum en secuencias de comandos más grandes que utilizan PERL, Ruby, Python, Bash, LISP y C++.

[14] Wild (1996)

2.3.2.1. Representación

En primer lugar, Humdrum define una sintaxis para representar información discreta como una serie de registros en un archivo de computadora.

Esta definición permite que se codifiquen muchos tipos de información. El esquema esencial utilizado en la base de datos CCARH para la altura y la duración musical es sólo uno de un conjunto abierto. Algunos otros esquemas pueden ser aumentados por gramáticas definidas por el usuario para tareas de investigación.

2.3.2.2. Manipulación

Segundo, está el conjunto de comandos, el Humdrum Toolkit, diseñado para manipular archivos que se ajusten a la sintaxis Humdrum en el campo de la investigación asistida por ordenador en la música.

El énfasis está en **asistido**:

- Humdrum no posee facultades analíticas de nivel superior per se.
- Más bien, *su poder deriva de la flexibilidad de su kit de elementos, utilizados en combinacióin* para explotar plenamente el potencial del sistema.

2.3.2.3. De la experiencia a la apreciación

Apreciación de todo el potencial de Humdrum es definitivamente a partir de la experiencia. En palabras de David Huron:

> Cualquier conjunto de herramientas requiere el desarrollo de una experiencia concomitante, y Humdrum Toolkit no es una excepción. Espero que la inversión de el tiempo requerido para aprender a usar Humdrum será más que compensado por ganancias académicas posteriores.

Los usuarios de Humdrum hasta ahora han tendido a trabajar en la percepción de la música o etnomusicología, mientras que los teóricos y los musicólogos histioriadores han sido mas lentos para reconocer el potencial del sistema.

2.3.2.4. CLI vs GUI

Humdrum u otros sistemas como él ofrecen los recursos para una marcar un paradigma para la investigación musical.

El tedio de recopilar pruebas sólidas que apoyen las propias teorías pueden ser aliviadas por la automatización, y cuanto mayor sea la cantidad de música examinada mayor será el rigor de la prueba de las hipótesis.

Sin embargo, la desafortunada posibilidad es que muchos de los musicólogos y teóricos que se benefician de una pequeña intuición asistida por la máquina es probable que sean repelidos por la interfaz totalmente basada en texto de Humdrum.

Aunque en el análisis final los comandos estilo UNIX son seguramente más flexibles y eficientes que una interfaz gráfica "amigable", pueden intimidar a principiantes, muchos de los cuales pueden resultar disuadidos de emplear herramientas de utilidad considerable.

Independientemente que teóricos de la música decidan o no aumentar su intuición musical con valiosas pruebas empíricas, los resultados basados en las cantidades máximas de datos pertinentes será un factor en la evolución de nuestra disciplina.

2.3.3. MusicXML

MusicXML[15] fue diseñado desde cero para compartir archivos de música entre aplicaciones y archivar registros de música para uso en el futuro. Se puede contar con archivos de MusicXML que son legibles y utilizables por una amplia gama de notaciones musicales, ahora y en el futuro. MusicXML complementa al los formatos de archivo utilizados por Finale y otros programas.

MusicXML se pretende un el estándar para compartir partituras interactivas, dado que facilita crear música en un programa y exportar sus resultados a otros programas. Al momento más de 220 aplicaciones incluyen compatibilidad con MusicXML.

2.3.4. Music Markup Language

El Lenguaje de Marcado de Música (MML)[16] es un intento de marcar objetos y eventos de música con un lenguaje basado en XML. La marcación de estos objetos debería permitir gestionar la música documentos para diversos fines, desde la teoría musical y la notación hasta rendimiento práctico. Este proyecto no está completo y está en progreso. El primer borrador de una posible DTD está disponible y se ofrecen algunos ejemplos de piezas de música marcadas con MML.

El enfoque es modular, varios módulos aún están incompletos y necesitan más investigación y atención. Una pieza musical serializada usando MML puede ser entregada en al menos los siguientes formatos:

- Texto: representación de notas como, por ejemplo, piano-roll (como el que se encuentra en el software del secuenciador de computadora).
- Common Western Notation: Notación musical occidental en pantalla o en papel
- MIDI-device: MML hace posible "secuenciar" una pieza de música sin tener que usar software especial. Así que cualquier persona con un editor de texto debe ser capaz de secuenciar la música de esta manera.

[15] Good (2001)
[16] Steyn (2001)

2.3.5. Flocking

Flocking[17] es un framework, escrito en JavaScript, para la composición de música por computadora que aprovecha las tecnologías e ideas existentes para crear un sistema robusto, flexible y expresivo. Flocking combina el patrón generador de unidades de muchos idiomas de música de computadora con tecnologías Web Audio para permitir a los usuarios interactuar con sitios Web entre otras potenciales tecnologías, usando un estilo declarativo de programación.

El objetivo de Flocking es permitir el crecimiento de un ecosistema de herramientas que puedan analizar y entender fácilmente la lógica y la semántica de los instrumentos digitales representando de forma declarativa los pilares básicos de síntesis de audio. Esto es particularmente útil para soportar la composición generativa donde los programas generan nuevos instrumentos de forma algorítmica, herramientas gráficas para que programadores y no programadores colaboren, y nuevos modos de programación social que permiten a los músicos adaptar, ampliar y volver a trabajar fácilmente en instrumentos existentes.

2.3.5.1. Programación declarativa

Arriba, se describió Flocking como un marco **declarativo**. Esta característica es esencial para comprender su diseño. La programación declarativa se puede entender en el contexto de Flocking por dos aspectos esenciales:

1. Enfatiza una visión semántica de alto nivel de la lógica y estructura de un programa
2. Representa los programas como estructuras de datos que pueden ser entendido por otros programas.

El énfasis aquí es sobre los aspectos lógicos o semánticos de la computación, en vez de en la secuenciación de bajo nivel y el flujo de control. Tradicionalmente los estilos de programación imperativos suelen estar destinados solo para el compilador. Aunque el código es a menudo compartido entre varios desarrolladores, no suele ser comprendidos o manipulados por programas distintos a los compiladores.

Por el contrario, la programación declarativa implica la capacidad de escribir programas que están representados en un formato que pueden ser procesados por otros programas como datos ordinarios. La familia de lenguajes Lisp es un ejemplo bien conocido de este enfoque. Paul Graham describe la naturaleza declarativa de Lisp, expresando que "no tiene sintaxis. Escribes programas en árboles de análisis… [que] son totalmente accesibles a tus programas. Puedes escribir programas que los manipulen… programas que escriben programas".[18] Aunque Flocking está escrito en JavaScript, comparte con Lisp el enfoque expresar programas dentro de estructuras de datos que estén disponibles para su manipulación por otros programas.

[17] Clark y Tindale (2014)
[18] Graham (2001)

Si bien la recopilación expuesta no agota la lista de referentes pertinentes y surgirán otros que cobraran relevancia, provee un criterio para proceder.

3. Metodología

En este capitulo se introduce el procedimiento de ejecución en el que se pueden distinguir tres etapas, una preparatoria, dedicada a investigación, experimentación y pruebas, deviene la fase de producción en si que culmina en una etapa de retoques, depuración de errores y defectos.

Se aprovecha para reseñar herramientas preexistentes elegidas, se mencionan aquellas que fueron consideradas pero descartadas luego de algunos ensayos y otras periféricas vinculadas a la tarea accesoria.

3.1. Preliminares

Se explican experiencias tempranas necesarias para evidenciar y comprobar que la hipótesis formulada fuese al menos abarcable y fundamentar los pasos siguientes.

A partir de las inquietudes presentadas, se propuso como objetivo inicial establecer una lista de parámetros que asocien valores a propiedades musicales elementales (altura, duración, intensidad, etc) necesarias para definir el conjunto articulaciones constituyentes de un discurso musical, en determinado sentido rudimental, austero y moderado.

Acorde a esto se hilvanó una rutina de procesos, compuesta por un interprete , un analizador sintáctico[19] y un codificador digital[20] entre otras herramientas, que a partir de valores emita un flujo de mensajes.

3.1.1. Boceto de Gramática

El método para discretizar información, jerarquizar y distinguir propiedades de valores, se basa en el formato YAML.[21] Luego de considerar este estándar y enfrentarlo con alternativas, se concluye que cumple con las condiciones y que es idóneo para la actividad.

Implementaciones del mismo en la mayoría de los entornos vigentes[22], aseguran la independencia de la información serializada en este sistema. Se le adjudica alta legibilidad[23]. Goza de cierta madurez, por lo que fue sujeto de ajustes y mejoras[24].

[19] (n.d.) (2019a)
[20] (n.d.) (2019c)
[21] Ben-Kiki, Evans y Ingerson (2005)
[22] (n.d.) (2019e)
[23] Huntley (2019)
[24] Ben-Kiki Oren y Ingy (2009)

3.1.2. Prototipo

Se esbozó un guión de instrucciones acotado a componer cadenas de eventos a partir de la interpretación, análisis sintáctico, proyección (mapeo) y asignación de valores.

Este prototipo, que confirmó la viabilidad de la aplicación pretendida, fue desarrollado en Perl,[25] lenguaje que luego de algunas consideraciones se desestimó por Python[26] debido principalmente a mayor adopción en la producción académica.

3.2. Desarrollo

En las actividades posteriores a las comprobaciones, aunque influenciados entre si, se pueden distinguir dos agrupamientos:

- Establecer relaciones de sucesión y jerarquía, que gestionen herencia de propiedades entre segmentos musicales subordinados o consecutivos y extender el léxico admitido con el propósito de cubrir una cantidad mayor de propiedades musicales.
- Escalar el prototipo a una herramienta informática que: sea capaz de consumir recursos informáticos, interpretar series de registros, manipular valores, derivarlos en articulaciones, empaquetar y registrar secuencias, entre otras propiedades.

3.2.1. Lenguaje

Al establecer este lenguaje formal, el primer esfuerzo se concentró en definir la organización de las propiedades de cada parte musical, conseguir una estructura lógica que ordene un discurso multi-parte.

La discriminación de los datos comienza a nivel de archivo, cada fichero contiene los datos relativos a estratos individuales en la pieza. Obteniendo así recursos que canalizan la información de cada parte junto con determinadas propiedades globales (tempo, armadura de clave, metro, letras, etc), que si bien pueden alojarse en una definición de canal, son meta eventos[27] que afectaran a total de la pieza.[28]

Dicho esto se continua con la organización interna de los documentos y algunas consideraciones acerca de el léxico acuñado.

[25] Wall (1999)
[26] Rossum (2018)
[27] Selfridge-Field (1997) Capitulo 3: MIDI Extensions for Musical Notation (1): NoTAMIDI Meta-Events
[28] La limitación en cantidad de canales y el carácter global de algunas propiedades son algunas de las imposiciones del el estándar MIDI.

3.2.1.1. Formato YAML

Las definiciones de pista son regidas por YAML. Si bien el vocabulario aceptado es propio de este proyecto, todas las interpretaciones son gestionadas por dicho lenguaje. Se reseñan los principales indicadores reservados y estructuras básicas.

En el estilo de bloques de YAML, similar al de Python, la estructura esta determinada por la indentación. En términos generales indentación se define como los espacios en blanco al comienzo de la linea. Por fuera de la indentación y del contexto escalar, YAML destina los espacios en blanco para separar entre símbolos.

Los indicadores reservados pertinentes señalar son: los dos puntos ":" denotan la proyección de un valor, el guión "-" que indica un bloque de secuencia, *Ancla* el nodo para referencia futura el símbolo "&" ampersand, habilitado así subsecuentes como *alias* son invocados con el simbolo "*" asterisco.

Quizás esta presentación austera suscite una intimidación aparente, como se aprecia en los ejemplos desplegados en el capitulo siguiente, con algunas reglas sencillas este lenguaje de marcado consigue plena legibilidad, sin dejar de ser flexible ni expresivo. Para mas información acerca de otras estructuras y el tratamiento especial caracteres reservados, referirse a la especificación del formato[29].

3.2.1.2. Vocabulario

Con intensión de favorecer a la comunidad hispanoparlante el léxico que integra este lenguaje específico de dominio[30] esta compuesto, salvo contadas excepciones, por vocablos del diccionario español. De todos modos, son sencillas las modificaciones requeridas para habilitar la comprensión de términos equivalentes (en diferentes idiomas).

Para negociar con la noción inabarcable que significa dar soporte a cada aspecto musical esperado, siendo imposible anticipar todos las aplicaciones estipuladas en determinado sentido arbitrarias y/o circunstanciales, se propone un sistema de complementos de usuarios que habilita la salida y entrada de valores, para su manipulación externa a la rutina provista. Si bien en el uso este sistema se mostró prometedor, su naturaleza no excede el carácter experimental y es menesteroso promover mejoras y consideraciones adicionales.

Los componentes del léxico y el sistema de complementos son detallados en el primer apartado del capitulo siguiente.

[29] Ben-Kiki, Evans y Ingerson (2005) Apartado 5.3: Indicator Characters y Capitulo 6: Basic Structures.

[30] (n.d.) (2019d)

3.2.2. Entorno

Tanto las abstracciones desarrolladas, así como también la rutina de instrucciones principales, esta escritas para el interprete *Python 3*[31]. Ademas de incorporar al entorno varios módulos de la "Librería Estándar"[32] esta pieza de software está apoyada en otros dos complementos, el marco de trabajo "PyYAML"[33] para asistir con el análisis sintáctico, en combinación con la librería "MIDIutil"[34] encargada de la codificación.

En el mismo ánimo con el que se compuso el vocabulario, el guion de acciones hace uso intensivo de idioma español. Esta decision es en cierto aspecto cuestionable siendo mayoritarias las sentencias predefinidas en ingles impuestas por el entorno.

3.2.2.1. Analizador Sintáctico

El primer proceso en la rutina es el de consumir información subscrita, interpretarla y habilitarla para su manipulación posterior. Se confía esta tarea al analizador sintáctico *PyYAML*.

En la presentación oficial del entorno dice:

- Ser completamente capaz de analizar YAML en su version 1.1, comprendiendo todos los ejemplos de dicha especificación.
- Implementar un algoritmo referente gracias a su sencillez.
- Soportar la codificación de caracteres Unicode en la entrada y la salida.
- Analizar y emitir eventos de bajo nivel, con la posibilidad alternativa de emplear la librería de sistema LibYAML.
- Poseer una interface de programación de alto nivel sencilla para objetos nativos Python. Con soporte para todos los tipos de datos de la especificación.

3.2.2.2. Codificación de Salida

La cadena de procesos finaliza cuando la lista articulaciones resultante, hasta esta instancia abstracciones en memoria, es secuenciada en eventos, codificada y registrada en ficheros.

MIDIUtil es una biblioteca que posibilita generar piezas multi-parte en formato MIDI 1 y 2 desde rutinas de Python. Posee abstracciones que permite crear y escribir estos archivos con mínimo esfuerzo.

El autor escusa implementar selectivamente algunos de los aspectos más útiles y comunes de la especificación MIDI, argumentando tratarse de un gran documen-

[31] Rossum (2018)
[32] (n.d.) (2018b)
[33] (n.d.) (2018a)
[34] 1

to en expansión a lo largo de décadas. A pesar de ser incompleta, las propiedades cubiertas fueron suficientes para este proyecto y sirvió como marco el objetivo de dar soporte a todo aspecto comprendido por la librería[35].

3.2.2.3. Otras herramientas

Para concluir el relato de método se mencionan dos herramientas accesorias de las cuales se hizo uso intensivo, tanto en el desarrollo de la investigación, como así también en la producción de este documento.

Siendo este proyecto texto-centrista, el ecosistema está incompleto sin un editor de texto apropiado[36]. Para conseguir fluidez y consistencia la herramienta empleada para esta actividad tiene que poder operar según el contexto, manipular bloques, disponer de macros sencillos y configurables. Para estos asuntos se confió en Vim[37].

El progreso y el respaldo en linea, fue agilizado por el sistema de control de versiones GIT.[38] Es con esta herramienta, que desde este repositorio se puede *clonar* el desarrollo, junto con las instrucciones para su instalación y uso.

Pese a que se comprenden estos temas en el dominio de usuario, se reconoce la ventaja y se sugiere el empleo de este tipo de herramientas.

4. Resultados

Se dedicada esta sección a detallar en profundidad el sistema propuesto. Comenzando por los constituyentes y preceptos, seguido los procesos de los cuales estos son objeto.

Concluyendo se expone la entrada, en formato YAML y la salida, representada en formato de partitura, de tres ejemplos: una melodía sencilla, un pieza con multiples partes y una discurso cuyos patrones son de duración no equivalente.

4.1. Gramática

Sobre la estructura sentada por el protocolo optado opera otro juego de reglas propio a este desarrollo que gobierna la combinatoria entre constituyentes. Luego de exponer estos principios se presenta el vocabulario concebido, que junto con la sintaxis completa esta gramática.

[35]

1.

[36] Moolenaar (2000)
[37] Oualline (2001)
[38] Torvalds y Hamano (2010)

4.1.1. Sintaxis

El discurso musical de cada parte se organiza en dos niveles. Se distinguen las propiedades globales que afectan a la totalidad de la pista de las que en un siguiente domino, definen cualidades particulares a cada unidad musical[39].

De la lista dispuesta en el próximo apartado, en cuestiones constitutivas se destaca el término `forma`[40]. Este indica la organización de unidades y recibe el mismo tratamiento a nivel macro que a nivel micro, en ambos casos `forma` representa una lista ordenada de elementos declarados disponibles en la paleta de `unidades`.

Si el elemento carece de este atributo ninguna otra unidad es invocada, por lo tanto se ejecuta el segmento.

Existe un secuencia de secciones musicales de primer grado que es relativa a la pista, de ahí en adelante las unidades se refieren entre ellas hasta alcanzar segmentos de ultimo grado, resultando una organización de árbol[41].

Previo a la descripción del vocabulario aceptado, relacionada a esta organización es la gestión de herencia entre unidades, la sucesión de propiedades. Unidades invocadas heredan las cualidades del referente, este sobrescribe propiedades en los referidos.

4.1.2. Léxico

A modo de referencia, se describe el léxico acuñado aprovechando la distinción expuesta anteriormente entre propiedades generales a la pista y particulares a las unidades.

Para detallar cada término evitando redundancias, se organiza la información repitiendo la misma estructura para cada uno de ellos. Se presentan en linea: el término que identifica la propiedad, el tipo de dato que se espera, el valor asignado por defecto, luego una breve descripción y por ultimo un ejemplo.

4.1.2.1. Propiedades de Pista

Los parámetros generales de cada pista son cuatro: El `nombre` de la pista define el rotulo soportado por el estándar MIDI que identifica dicha parte en la pieza, la paleta de `unidades` disponibles, la `forma` indica la secuencia de unidades de primer grado y los `complementos` de usuario.

`nombre` (cadena de caracteres) default: `<empty>`

Titulo de la pista.

[39] Grela (1992) Se adopta la terminología *unidad* para referir elementos musicales y *grado* para denotar el alcance de dicho agrupamiento.

[40] Se utiliza fuente tipográfica `monoespaciada` para resaltar el vocabulario propio de este proyecto.

[41] Pope (1986) Designing Musical Notations, Sequences And Trees.

```
nombre: 'Feliz Cumpleaños'
```

`unidades` (diccionario) default: `[ None ]`

Paleta de unidades disponibles (con sus propiedades respectivas) para ser invocadas en `forma`.

```
unidades:
  base: &base
    transportar: 72
    registracion: [ 0, 2, 4, 5, 7, 9, 11, 12 ]
    metro: 3/4
  a: &a
    <<: *base
    alturas: [ 5, 5, 6, 5 ]
    duraciones: [ .75, .25, 1, 1 ]
  b: &b
    <<: *base
    alturas: [ 8, 7 ]
    duraciones: [ 1, 2 ]
  motivoA:
    forma: [ 'a', 'b' ]
  motivoB:
    forma: [ 'b', 'b' ]
```

`forma` (lista de cadenas de caracteres) default: `[ None ]`

Macro estructura de la pista. Lista de unidades a ser secuenciadas, cada elemento corresponde a un miembro de la paleta de `unidades`.

```
forma: [ 'motivoA', 'motivoB' ]
```

`complementos` (cadena de caracteres) default: `None`

Ubicación de módulo con métodos de usuario.

```
complementos: 'enchufes.py'
```

4.1.2.2. Propiedades de Unidad

En el diccionario de unidades de la pista cada entrada representa una unidad disponible, que a su vez aloja sus cualidades. Esta es la lista de términos aceptados como propiedades para cada unidad.

`forma` (lista de cadenas de caracteres) default: `[ None ]`

Estructura de la unidad. Lista de unidades referidas a ser secuenciadas.
Cada elemento corresponde a un miembro de la paleta de unidades.

```
forma: ['A', 'B']
```

`alteraciones` (número entero) default: 0

Cantidad de alteraciones en la armadura de clave.
Números positivos representan sostenidos y números negativos indican la cantidad bemoles.

```
alteraciones: -2 # Bb
```

`modo` (número entero) default: 0

Modo de la escala.
El número 0 indica que se trata de una escala mayor, mientras que el 1 representa tonalidad menor.

```
modo: 1 # menor
```

`registracion` (lista de enteros) default: `[ 1 ]`

Conjunto de intervalos a ser indexados por el puntero de **`alturas`**.

```
registracion: [
  -12,-10, -9, -7, -5, -3, -2,
    0,  2,  3,  5,  7,  9, 10,
   12, 14, 15, 17, 19, 21, 22,
   24
]
```

`transportar` (número entero) default: 0

Ajuste de alturas en semitonos

```
transportar: 60 # C
```

`transponer` (número entero) default: 0

Transponer puntero de intervalo.
Ajuste de alturas, pero dentro de la **`registracion`** fija.

```
transponer: 3
```

`metro` (cadena de caracteres) default: 4/4

Clave de compás.

```
metro: 4/4
```

`reiterar` (número entero) default: `1`

Repeticiones, cantidad de veces q se toca esta unidad.
Esta propiedad no es transferible, no se sucede (de lo contrario se reiteraran los referidos).

```yaml
reiterar: 3
```

`canal` (número entero) default: `1`

Número de Canal.

```yaml
canal: 3
```

`afinacionNota` (diccionario) default: `None`

Afinación de nota.

```yaml
afinacionNota:
  afinaciones: [ [ 69, 50 ], [ 79, 60 ] ]
  canalSysEx: 127
  tiempoReal: true
  programa: 0
```

`afinacionBanco` (diccionario) default: `None`

Afinación banco.

```yaml
afinacionBanco:
  banco: 0
  ordenar: false
```

`afinacionPrograma` (diccionario) default: `None`

Afinacion Programa.

```yaml
afinacionPrograma:
  programa: 0
  ordenar: false
```

`RPN` (diccionario) default: `None`

Realizar llamada a parámetro numerado registrado.

```yaml
RPN:
  control_msb: 0
  control_lsb: 32
  data_msb: 2
  data_lsb: 9
  ordenar: True
```

`NRPN` (diccionario) default: `None`

Realizar llamada a parámetro numerado no registrado.

```yaml
NRPN:
  control_msb: 0
  control_lsb: 32
  data_msb: 2
  data_lsb: 9
  ordenar: True
```

`sysEx` (diccionario) default: `None`

Agregar un evento de sistema exclusivo.

```yaml
sysEx:
  fabricante: 0
  playload:  !!binary ''
```

`uniSysEx` (diccionario) default: `None`

Agregar un evento de sistema exclusivo universal.

```yaml
uniSysEx:
  codigo: 0
  subCodigo: 0
  playload:  !!binary ''
  canal: 14
  tiempoReal: False
```

`metodo-usuario` (diccionario) default: `None`

Invocar métodos declarados como `complementos` de usuario mediante su respectivo nombre, subordinando a este las propiedades de unidad que se quieran manipular y pasar como valores los argumentos que espera esta rutina.

```yaml
unidades:
  a:
    alturas: [ 5, 5, 6, 5 ]
    fluctuar: # metodo-usuario
      dinamicas: .5 # propiedad:argumentos
      duraciones: .3 # propiedad:argumentos
    desplazar: # metodo-usuario
      duraciones:  .25 # propiedad:argumentos
```

4.1.2.3. Articulaciones

Si bien estructuralmente no se distingue otra jerarquía, las siguientes propiedades actúan a nivel de articulación, se subscriben a unidades pero en vez de

modificar al segmento como conjunto, resultan en un valor por cada articulación.

Comparten la cualidad de esperar listas de valores y proceso combinatorio al cual son sujetas es similar al empleado en la técnica compositiva del motete isorrítmico[42], difiriendo en que el procedimiento no se limita a duraciones y alturas, abarca otras propiedades.

La cantidad de articulaciones producidas es equivalente al número de miembros en la serie mas extensa, se reiteran secuencialemente patrones mas cortos alineandose, hasta completar el total de articulaciones.

Es pertinente señalar la combinatoria que resulta indexando los valores de la serie de `alturas` como punteros en el conjunto intervalos de `registración` fija.

`alturas` (lista de números enteros) default: `[ 1 ]`

Punteros de intervalo.
Cada miembro es un índice de intervalo en el conjunto: `registracion`.

```
alturas: [ 1, 3, 5, 8 ]
```

`voces` (lista de listas de números enteros) default: `None`

Superposición de alturas. Apilamiento de voces dentro de la `registracion` fija.
Lista de listas, cada miembro es una voz y cada voz es un lista de enteros que se añade al puntero de intervalo.

```
voces:
  - [ 8, 6 ]
  - [ 5 ]
  - [ 3 ]
```

`BPMs` (lista de números enteros) default: `[ 60 ]`

Pulsos por minuto.

```
BPMs: [ 75, 65, 70 ]
```

`duraciones` (lista de números decimales) default: `[ 1 ]`

Largo de articulaciones, lista de factores relativos al de pulso.

```
duraciones: [ 1, .5, .5, 1, 1 ]
```

`dinamicas` (lista de números decimales) default: `[ 1 ]`

Lista ordenada de dinámicas. Factores de 127.

[42] Variego (2018)

```
dinamicas: [ 1, .5, .4 ]
```

`programas` (lista de números enteros) default: `[ None ]`

Lista de números de instrumento MIDI en el banco actual.

```
programas: [ 25 ] # Guitarra en banco GM
```

`controles` (lista de listas de pares) default: `None`

Cambios de control. Secuencia de pares, número controlador y valor a asignar.

```
controles:
  - [ 70 : 80,  71 : 90, 72 : 100 ]
  - [ 33 : 121, 51 : 120 ]
  - [ 10 : 80,  11 : 90, 12 : 100, 13 : 100 ]
```

`tonos` (lista de enteros) default: `[ 0 ]`

Curva de entonaciones (pitch-bends).

```
tonos: [ 666, -777, 0 ]
```

`letras` (lista de cadenas de caracteres) default: `[ None ]`

Texto. Versos o anotaciones.

```
letras: [ 'Hola', 'mundo.' ]
```

4.2. Implementación

En esta apartado se diagrama el flujo de procesos, se expone la estructura de la aplicación, detallando las funciones principales de cada componente y como están conectados entre ellos.

Antes de la descripción de cada capa de abstracción, se gráfica la cadena de procesos con intensión de presentarla abarcable y facilitar su comprensión.

La rutina principal comienza leyendo, desde argumentos posicionales, definiciones de pista. Después de analizadas sintacticamente se entregan como diccionarios reunidos en un lista al modulo *Secuencia*.[43]

[43] Van Rossum y Drake Jr (1995) 10.3: Command Line Arguments, 5.5: Dictionaries, 3.1.3: Lists, 9: Classes.

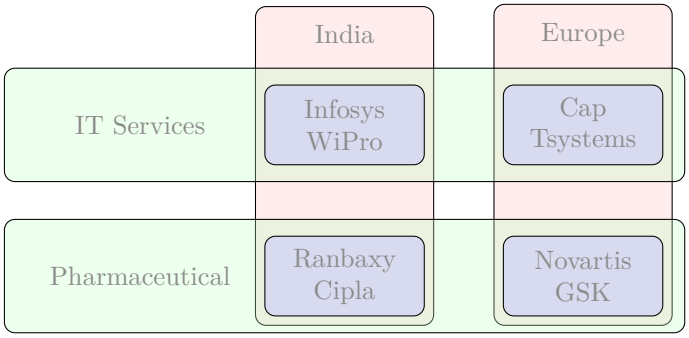

Figura 1: Clients graph

El producto de dicha manipulación es una lista ordenada de llamadas a diversas funciones que operan el codificador, el cual concluye el programa emitiendo una secuencia de mensajes MIDI.

A continuación se expone el trazado de mecanismos internos, desde el nivel de abstracción superior *Secuencia*, alcanzando la capa *Articulación* comprendida como la mas profunda.

4.2.1. Secuencia

Después que la información es recibida y consolidada como objetos de clase *Pista*, el programa recorre articulaciones y cambios de parámetros entre los elementos cada nodo, reuniendo cada pronunciamientos en un único flujo de instrucciones[44].

4.2.1.1. Pista

Este módulo es responsable por el devenir del las partes musicales. Desde el nivel macro organiza la estructura de cada instancia, discriminando entre elementos que refieren a otros, que son clasificados como *Sección*, de los que no refieren a ningún otro elemento, a los que consolida como *Segmento*.

Al mismo tiempo gestiona la sucesión de propiedades entre referente y referido. Resuelve el conjunto de propiedades resultantes de cada elemento y los dispone consecutivamente para ser consumidos en el nivel de abstracción superior.

4.2.1.2. Complemento

Importa rutinas en el módulo usuario (declarado en `complementos` de la pista) como procesos capaces de manipular propiedades de cualquier elemento de la *Pista*.

[44]El extracto de código correspondiente a cada constructor se encuentran en Apéndice 6.1: Secuencia, Apéndice 6.2: Pista, Apéndice 6.3: Complemento, Apéndice 6.4: Unidad, Apéndice 6.5 : Sección, Apéndice 6.6: Segmento y Apéndice 6.7: Articulación respectivamente.

En concordancia cada *Segmento* busca coincidencia entre cada método de usuario y clave en sus propiedades, delegando a la rutina de usuario la manipulación de los valores de la propiedad subordinada.

4.2.1.3. Unidad

Justifica esta meta clase la noción de una capa superior de agrupamiento y registro. Ademas de ahorrar alguna redundancia, habilita información de capacidad analítica en la salida verbal.

4.2.1.4. Sección

Representa conjuntos de *Secciones* y/o *Segmentos*. Es un grupo de elementos que, sin vincular articulaciones directamente y sin incidir en el discurso resultante, a nivel lógico mantiene relaciones de referencia.

4.2.1.5. Segmento

En contra partida, esta clase reúne los mecanismos para preparar y producir *Articulaciones*.

Es responsable de administrar `complementos`, distinguir actualizaciones de valor entre instancias y la gestiona alturas, trasponiendo el puntero dentro el set de registración y trasportando la nota resultante.

Completa secuencialmente patrones dispares alineándolos en relación a lista con mayor a la cantidad de miembros. Invoca una instancia de *Articulacion* por cada resultante de esta combinatoria.

4.2.1.6. Articulación

Esta capa corresponde a pronunciamientos en el discurso. Es el producto de mecanismos superiores y la materia prima interpretada en la *Secuencia* final asignando una llamada al codificador por cada instancia.

4.3. Demostraciones

Explicación de que ejemplo o demostración se va a discutir en cada sección.

4.3.1. Melodia Simple

Descripcion

4.3.1.1. YAML

Código

4.3.1.2. Partitura

Captura

4.3.2. Multiples Pistas

Descripcion

4.3.2.1. YAML

Códigos

4.3.2.2. Partitura

Capturas

4.3.3. Polimetría

Paterns con duraciones no equivalentes

4.3.3.1. YAML

Códigos

4.3.3.2. Partitura

Capturas

5. Concluciones

aplicaciones posibles en diferentes escenarios (online, archivología, livecodig) y varias disciplinas (IA, machine learning).

6. Apéndice

6.1. Secuencia

```
from .pista import Pista

class Secuencia:

  def __init__(
      self,
      definiciones,
      verbose,
      copy
    ):
    self.definiciones = definiciones
    self.pistas = []
    self.verbose = verbose
```

```
    self.copyright = copy


    for d in self.definiciones:
      r = {
        **Pista.defactos,
        **d
      }
      pista = Pista(
        nombre    = r[ 'nombre' ],
        paleta    = r[ 'unidades' ],
        forma     = r[ 'forma' ],
        plugin    = r[ 'complementos' ],
        secuencia = self,
      )
      self.pistas.append( pista )

  @property
  def eventos( self ):
    """ A partir de cada definicion agrega una Pista. """
    EVENTOS = []
    for pista in self.pistas:

      if self.verbose:
        print( pista.verbose( self.verbose ) )

      """ Generar track p/c pista """
      delta = 0
      track = pista.numero

      """ Parametros de Pista Primer articulación de la parte, agregar
      eventos fundamentales: pulso, armadura de clave, compás y programa.
      """
      EVENTOS.append([
        'addTrackName',
        track,
        delta,
        pista.nombre
      ])

      if self.copyright:
        EVENTOS.append([
          'addCopyright',
          track,
          delta,
          self.copyright
```

```
60          ])
61
62        """ Loop principal:
63        Genera una secuencia de eventos MIDI lista de articulaciones.  """
64
65        for segmento in pista.segmentos:
66          canal = segmento.canal
67          #delta += segmento.desplazar
68
69          if delta < 0:
70           raise ValueError( 'No se puede desplazar antes q el inicio' )
71           pass
72
73          """ Agregar propiedades de segmento. """
74
75          if segmento.cambia( 'metro' ):
76            EVENTOS.append([
77              'addTimeSignature',
78              track,
79              delta,
80              segmento.metro[ 'numerador' ],
81              segmento.metro[ 'denominador' ],
82              segmento.metro[ 'relojes_por_tick' ],
83              segmento.metro[ 'notas_por_pulso' ]
84            ])
85
86          if segmento.cambia( 'bpm' ):
87            EVENTOS.append([
88              'addTempo',
89              track,
90              delta,
91              segmento.bpm,
92            ])
93
94          if segmento.cambia( 'clave' ):
95            EVENTOS.append([
96              'addKeySignature',
97              track,
98              delta,
99              segmento.clave[ 'alteraciones' ],
100             1, # multiplica por el n de alteraciones
101             segmento.clave[ 'modo' ]
102           ])
103
104         if segmento.afinacionNota:
105           EVENTOS.append([
```

```
106             'changeNoteTuning',
107              track,
108              segmento.afinacionNota[ 'afinaciones' ],
109              segmento.afinacionNota[ 'canalSysEx' ],
110              segmento.afinacionNota[ 'tiempoReal' ],
111              segmento.afinacionNota[ 'programa' ],
112            ])
113
114          if segmento.afinacionBanco:
115            EVENTOS.append([
116              'changeTuningBank',
117              track,
118              canal,
119              delta,
120              segmento.afinacionBanco[ 'banco' ],
121              segmento.afinacionBanco[ 'ordenar' ],
122            ])
123
124          if segmento.afinacionPrograma:
125            EVENTOS.append([
126              'changeTuningProgram',
127              track,
128              canal,
129              delta,
130              segmento.afinacionPrograma[ 'programa' ],
131              segmento.afinacionPrograma[ 'ordenar' ],
132            ])
133
134          if segmento.sysEx:
135            EVENTOS.append([
136             'addSysEx',
137              track,
138              delta,
139              segmento.sysEx[ 'fabricante' ],
140              segmento.sysEx[ 'playload' ],
141            ])
142
143          if segmento.uniSysEx:
144            EVENTOS.append([
145             'addUniversalSysEx',
146              track,
147              delta,
148              segmento.uniSysEx[ 'codigo' ],
149              segmento.uniSysEx[ 'subCodigo' ],
150              segmento.uniSysEx[ 'playload' ],
151              segmento.uniSysEx[ 'canal' ],
```

```
152             segmento.uniSysEx[ 'tiempoReal' ],
153           ])
154
155         if segmento.NRPN:
156           EVENTOS.append([
157            'makeNRPNCall',
158             track,
159             canal,
160             delta,
161             segmento.NRPN[ 'control_msb' ],
162             segmento.NRPN[ 'control_lsb' ],
163             segmento.NRPN[ 'data_msb' ],
164             segmento.NRPN[ 'data_lsb' ],
165             segmento.NRPN[ 'ordenar' ],
166           ])
167
168         if segmento.RPN:
169           EVENTOS.append([
170            'makeRPNCall',
171             track,
172             canal,
173             delta,
174             segmento.RPN[ 'control_msb' ],
175             segmento.RPN[ 'control_lsb' ],
176             segmento.RPN[ 'data_msb' ],
177             segmento.RPN[ 'data_lsb' ],
178             segmento.RPN[ 'ordenar' ],
179           ])
180
181         for articulacion in segmento.articulaciones:
182           """ Agrega cualquier cambio de parametro,
183           comparar cada uno con la articulacion previa. """
184
185           if articulacion.cambia( 'bpm' ):
186             EVENTOS.append([
187               'addTempo',
188               track,
189               delta,
190               articulacion.bpm,
191             ])
192
193           if articulacion.cambia( 'programa' ):
194             EVENTOS.append([
195                'addProgramChange',
196                track,
197                canal,
```

```
198                 delta,
199                 articulacion.programa
200               ])
201
202             if articulacion.letra:
203               EVENTOS.append([
204                'addText',
205                track,
206                delta,
207                articulacion.letra
208               ])
209
210             if articulacion.tono:
211               EVENTOS.append([
212                  'addPitchWheelEvent',
213                  track,
214                  canal,
215                  delta,
216                  articulacion.tono
217               ])
218
219             """ Agregar nota/s (altura, duracion, dinamica).
220             Si existe acorde en la articulación armar una lista con cada voz
221             superpuesta.  o una lista de solamente un elemento.  """
222             voces = [ articulacion.altura ]
223             if articulacion.acorde:
224               voces = articulacion.acorde
225
226             for voz in voces:
227               if articulacion.dinamica:
228                 EVENTOS.append([
229                   'addNote',
230                   track,
231                   canal,
232                   voz,
233                   delta,
234                   articulacion.duracion,
235                   articulacion.dinamica
236                 ])
237               #else:
238               #  print('eo')
239
240             if articulacion.controles:
241               """ Agregar cambios de control """
242               for control in articulacion.controles:
243                 for control, valor in control.items():
```

```
244                 EVENTOS.append([
245                  'addControllerEvent',
246                   track,
247                   canal,
248                   delta,
249                   control,
250                   valor,
251                 ])
252 
253           delta += articulacion.duracion
254     return EVENTOS
```

6.2. Pista

```
import os
from argumentos import Excepcion
from .complemento import Complemento
from .seccion  import Seccion
from .segmento import Segmento

class Pista:
  """
  Clase para cada definicion de a partir de archivos .yml
  Secuencia > PISTA > Secciones > Segmentos > Articulaciones
  """
  cantidad = 0

  defactos = {
    'nombre'       : '',
    'unidades'     : [ None ],
    'forma'        : [ None ],
    'complementos' : None,
  }

  def __str__( self ):
    o  = 'PISTA ' + str( self.numero) + ': '+ str( self.nombre  )
    return o

  def verbose( self, verbosidad = 0 ):
    if verbosidad > 0:
      o = str( self ) + ' '
      o = str( self ) + ' '
      o +=  '#' * ( 70 - len( o ) )
      if verbosidad > 1:
        o += '\nELEM\tid\t#\tNIVEL\tRECUR\tNOMBRE\n'
        for e in self.elementos:
          o += e.verbose( verbosidad )
          o += '\n'
      return o

  def __init__(
    self,
    nombre,
    paleta,
    forma,
    plugin,
    secuencia,
```

```
44    ):
45      self.nombre     = nombre
46      self.paleta     = paleta
47      self.forma      = forma
48      self.secuencia  = secuencia
49      self.plugin     = plugin
50      self.numero     = Pista.cantidad
51      Pista.cantidad += 1
52
53      self.secciones  = []
54      self.segmentos  = []
55      self.seccionar( self.forma )
56
57    @property
58    def complemento( self ):
59      ubicacion = self.plugin
60      c = None
61      if ubicacion and os.path.exists( ubicacion ):
62        # TODO Tirar excepción
63        # and p not in Complemento.registro:
64        Complemento.registro.append( ubicacion )
65        c = Complemento( ubicacion )
66      return c
67
68    @property
69    def elementos( self ):
70      return sorted(
71        self.secciones + self.segmentos,
72        key = lambda x: x.numero
73      )
74
75    @property
76    def tiempo( self ):
77      # duracion en segundos
78      return sum( [ s.tiempo for s in self.segmentos ] )
79
80    """ Organiza unidades según relacion de referencia """
81    def seccionar(
82      self,
83      forma = None,
84      nivel = 0,
85      herencia = {},
86      referente = None,
87    ):
88      nivel += 1
89      """ Limpiar parametros q no se heredan.  """
```

```
90      herencia.pop( 'forma', None )
91      herencia.pop( 'reiterar', None )
92
93      for unidad in forma:
94        try:
95          if unidad not in self.paleta:
96            error = "PISTA: \"" + self.nombre + "\""
97            error += " NO ENCUENTRO: \"" + unidad + "\"  "
98            raise Excepcion( unidad, error )
99            pass
100         original = self.paleta[ unidad ]
101         sucesion = {
102           **original,
103           **herencia,
104         }
105         reiterar = 1
106         if 'reiterar' in original:
107           reiterar = original[ 'reiterar' ]
108         for r in range( reiterar ):
109           if 'forma' not in original:
110             segmento = Segmento(
111               pista       = self,
112               nombre      = unidad,
113               nivel       = nivel - 1,
114               orden       = len( self.segmentos ),
115               recurrencia = sum(
116                 [ 1 for e in self.segmentos if e.nombre == unidad ]
117               ),
118               referente = referente,
119               propiedades = sucesion,
120             )
121             self.segmentos.append( segmento )
122           else:
123             seccion = Seccion(
124               pista       = self.nombre,
125               nombre      = unidad,
126               nivel       = nivel - 1,
127               orden       = len( self.secciones ),
128               recurrencia = sum(
129                 [ 1 for e in self.secciones if e.nombre == unidad ]
130               ),
131               referente   = referente,
132             )
133             seccion.referidos = original['forma']
134             self.secciones.append( seccion )
135             elemento = seccion
```

```
136             self.seccionar(
137               original[ 'forma' ],
138               nivel,
139               sucesion,
140               seccion,
141             )
142       except Excepcion as e:
143           print( e )
```

6.3. Complemento

```
import importlib.util as importar

class Complemento:

    """ Interfaz para complementos de usuario.

        Declarar ubicación del paquete en propiedades de track.
        complementos: 'enchufes.py'

        En propiedades de segmento invocar metodo y subscribir, propiedad y
        argumentos.

        metodo: # en: enchufes.py
          propiedad_a_manipular1: argumentos
          propiedad_a_manipular2: argumentos
        fluctuar:
          dinamicas: .5
          alturas: 2
    """

    cantidad = 0
    registro = []

    def __str__(
      ubicacion,
    ):
      return self.nombre

    def __init__(
      self,
      ubicacion
    ):
     self.ubicacion = ubicacion
     self.nombre = ubicacion.split( '.' )[ 0 ]
     Complemento.cantidad += 1
     spec = importar.spec_from_file_location(
       self.nombre,
       self.ubicacion
     )
     if spec:
       modulo = importar.module_from_spec( spec )
       spec.loader.exec_module( modulo )
       self.modulo = modulo
```

6.4. Unidad

```
class Elemento():
  """
  Secuencia > Pistas > ELEMENTOS
  Metaclase base para Secciones, Segmentos
  """
  cantidad = 0
  def __str__( self ):
    o  = str( self.numero ) + '\t'
    o += str( self.orden ) + '\t'
    o += str( self.nivel )  + '\t'
    o += str( self.recurrencia ) + '\t'
    o += '+' + str( '-' * ( self.nivel ) )
    o += self.nombre
    return o

  def __init__(
    self,
    pista,
    nombre,
    nivel,
    orden,
    recurrencia,
    referente,
  ):
    self.pista  = pista
    self.nombre = nombre
    self.nivel = nivel
    self.orden = orden
    self.recurrencia = recurrencia
    self.referente = referente
    self.numero = Elemento.cantidad
    Elemento.cantidad += 1
```

6.5. Sección

```
from .elemento import Elemento

class Seccion( Elemento ):
  cantidad = 0
  """ Pista > SECCION > Segmentos > Articulaciones """

  def verbose( self, vebosidad = 0 ):
    #o =  ('=' * 70 ) + '\n'
    o = self.tipo + ''
    o += str( self.numero_seccion ) + '\t'
    o += str( self ) + ' '
    o += '=' * ( 28 - ( len( self.nombre ) + self.nivel ) )
    return o

  def __init__(
    self,
    pista,
    nombre,
    nivel,
    orden,
    recurrencia,
    referente
  ):
    Elemento.__init__(
      self,
      pista,
      nombre,
      nivel,
      orden,
      recurrencia,
      referente
    )
    self.numero_seccion = Seccion.cantidad
    Seccion.cantidad += 1
    self.nivel = nivel
    self.tipo = 'SECC'
```

6.6. Segmento

```
import math
from .elemento import Elemento
from .articulacion import Articulacion
from .complemento import Complemento


class Segmento( Elemento ):
  """
  Secuencia > Pista > Secciones > SEGMENTOS > Articulaciones
  Conjunto de Articulaciones
  """
  cantidad = 0

  defactos = {

    # Propiedades de Segmento
    'canal'    : 0,
    'revertir' : None,
    'NRPN'     : None,
    'RPN'      : None,

    # Props. que NO refieren a Meta Canal
    'metro'             : '4/4',
    'alteraciones'      : 0,
    'modo'              : 0,
    'afinacionNota'     : None,
    'afinacionBanco'    : None,
    'afinacionPrograma' : None,
    'sysEx'             : None,
    'uniSysEx'          : None,

    # Procesos de Segmento
    'transportar' : 0,
    'transponer'  : 0,
    'reiterar'    : 1,

    # Propiedades de Articulacion
    'BPMs'         : [ 60 ],
    'programas'    : [ None ],
    'duraciones'   : [ 1 ],
    'dinamicas'    : [ 1 ],
    'registracion' : [ 1 ],
    'alturas'      : [ 1 ],
```

```
    'letras'        : [ None ],
    'tonos'         : [ 0 ],
    'voces'         : None,
    'controles'     : None,

  }

  def verbose( self, vebosidad = 0 ):
    o = self.tipo + ''
    o += str( self.numero_segmento) + '\t'
    o += str( self ) + ' '
    o += '-' * ( 28 - (len( self.nombre ) + self.nivel))
    if vebosidad > 2:
      o +=  '\n' + ('.' * 70 )
      o += '\n\tORD\tBPM\tDUR\tDIN\tALT\tLTR\tTON\tCTR\n'
      for a in self.articulaciones:
        o += str( a )
      o +=  ('.' * 70 )
    return o

  def __init__(
    self,
    pista,
    nombre,
    nivel,
    orden,
    recurrencia,
    referente,
    propiedades
  ):
    Elemento.__init__(
      self,
      pista,
      nombre,
      nivel,
      orden,
      recurrencia,
      referente
    )
    self.numero_segmento = Segmento.cantidad
    Segmento.cantidad += 1
    self.tipo = 'SGMT'

    self.props = {
      **Segmento.defactos,
      **propiedades
```

```
90      }
91
92      """ PRE PROCESO DE SEGMENTO """
93      """ Cambia el sentido de los parametros de
94      articulacion """
95      self.revertir = self.props[ 'revertir' ]
96      if self.revertir:
97        if isinstance( self.revertir , list ):
98          for r in self.revertir:
99            if r in self.props:
100             self.props[ r ].reverse()
101       elif isinstance( self.revertir , str ):
102         if revertir in self.props:
103           self.props[ self.revertir ].reverse()
104
105     self.canal             = self.props[ 'canal' ]
106     self.reiterar          = self.props[ 'reiterar' ]
107     self.transponer        = self.props[ 'transponer' ]
108     self.transportar       = self.props[ 'transportar' ]
109     self.alteraciones      = self.props[ 'alteraciones' ]
110     self.modo              = self.props[ 'modo' ]
111     self.afinacionNota     = self.props[ 'afinacionNota' ]
112     self.afinacionBanco    = self.props[ 'afinacionBanco' ]
113     self.afinacionPrograma = self.props[ 'afinacionPrograma' ]
114     self.sysEx             = self.props[ 'sysEx' ]
115     self.uniSysEx          = self.props[ 'uniSysEx' ]
116     self.NRPN              = self.props[ 'NRPN' ]
117     self.RPN               = self.props[ 'RPN' ]
118     self.registracion      = self.props[ 'registracion' ]
119
120     self.programas         = self.props[ 'programas' ]
121     self.duraciones        = self.props[ 'duraciones' ]
122     self.BPMs              = self.props[ 'BPMs' ]
123     self.dinamicas         = self.props[ 'dinamicas' ]
124     self.alturas           = self.props[ 'alturas' ]
125     self.letras            = self.props[ 'letras' ]
126     self.tonos             = self.props[ 'tonos' ]
127     self.voces             = self.props[ 'voces' ]
128     self.capas             = self.props[ 'controles' ]
129
130     self.bpm = self.BPMs[0]
131     self.programa = self.programas[0]
132
133
134     """ COMPLEMENTOS
135         Pasar propiedades por metodos de usuario
```

```
136     """
137     #for complemento in self.pista.complemento:
138     if self.pista.complemento:
139       for metodo in dir( self.pista.complemento.modulo ):
140         if metodo in self.props:
141           for clave in self.props[ metodo ]:
142             original = getattr( self, clave )
143             argumentos = self.props[ metodo ][ clave ]
144             # print( metodo, ':', clave, argumentos )
145             modificado = getattr(
146                self.pista.complemento.modulo,
147                metodo,
148             )( original, argumentos )
149             setattr( self, clave, modificado )
150
151
152   @property
153   def precedente( self ):
154     n = self.orden
155     o = self.pista.segmentos[ n - 1 ]
156     return o
157
158   def obtener( self, key ):
159       try:
160         o = getattr( self, key )
161         return o
162       except AttributeError as e:
163         return e
164
165   def cambia(
166       self,
167       key
168     ):
169     este = self.obtener( key )
170     anterior = self.precedente.obtener( key )
171     if (
172       self.orden == 0
173       and este
174     ):
175       return True
176     return anterior != este
177
178   @property
179   def tiempo( self ):
180     # duracion en segundos
181     return sum( [ a.tiempo for a in self.articulaciones ] )
```

```
182
183   @property
184   def metro( self ):
185     metro = self.props[ 'metro' ].split( '/' )
186     denominador = int(
187       math.log10( int( metro[ 1 ] ) ) / math.log10( 2 )
188     )
189     return {
190       'numerador'         : int( metro[ 0 ] ),
191       'denominador'       : denominador,
192       'relojes_por_tick' : 12 * denominador,
193       'notas_por_pulso'   : 8,
194     }
195
196   @property
197   def clave( self ):
198     return {
199       'alteraciones' : self.alteraciones,
200       'modo' : self.modo
201     }
202
203   @property
204   def ganador( self ):
205     """ Evaluar que propiedad lista es el que mas valores tiene.  """
206     self.ganador_voces = [ 0 ]
207     if self.voces:
208       self.ganador_voces = max( self.voces, key = len )
209     self.ganador_capas = [ 0 ]
210     if self.capas:
211       self.ganador_capas = max( self.capas , key = len )
212
213     candidatos = [
214       self.dinamicas,
215       self.duraciones,
216       self.alturas,
217       self.letras,
218       self.tonos,
219       self.BPMs,
220       self.programas,
221       self.ganador_voces,
222       self.ganador_capas,
223     ]
224     return max( candidatos, key = len )
225
226   @property
227   def cantidad_pasos( self ):
```

```
228     return len( self.ganador )
229
230   @property
231   def articulaciones( self ):
232
233     """ Consolidar "articulacion"
234     combinar parametros: altura, duracion, dinamica, etc. """
235     o = []
236     for paso in range( self.cantidad_pasos ):
237       """ Alturas, voz y superposición voces. """
238       altura = self.alturas[ paso % len( self.alturas ) ]
239       acorde = []
240       nota   = altura
241       """ Relacion: altura > puntero en el set de registracion;
242       Trasponer dentro del set de registracion, luego Transportar,
243       sumar a la nota resultante. """
244       n = self.registracion[
245         ( ( altura - 1 ) + self.transponer ) % len( self.registracion )
246       ]
247       nota = self.transportar + n
248       """ Armar superposicion de voces. """
249       if self.voces:
250         for v in self.voces:
251           voz = (
252             altura + ( v[ paso % len( v ) ] )
253           ) + self.transponer
254           acorde += [
255             self.transportar +
256             self.registracion[ voz % len( self.registracion ) ]
257           ]
258       """ Cambios de control. """
259       controles = []
260       if self.capas:
261         for capa in self.capas:
262           controles += [ capa[ paso % len( capa ) ] ]
263       """ Articulación a secuenciar. """
264       articulacion = Articulacion(
265          segmento  = self,
266          orden     = paso,
267          bpm       = self.BPMs[ paso % len( self.BPMs ) ],
268          programa  = self.programas[ paso % len( self.programas ) ],
269          # TODO advertir si lista de duraciones vacias
270          duracion  = self.duraciones[ paso % len( self.duraciones ) ],
271          dinamica  = self.dinamicas[ paso % len( self.dinamicas ) ],
272          nota      = nota,
273          acorde    = acorde,
```

```
274           tono      = self.tonos[ paso % len( self.tonos ) ],
275           letra     = self.letras[ paso % len( self.letras ) ],
276           controles = controles,
277       )
278       o.append( articulacion )
279     return o
280
```

6.7. Articulación

```
class Articulacion:

  """ Secuencia > Pistas > Secciones > Segmentos > ARTICULACIONES """

  cantidad = 0

  def __str__( self ):
    o  = 'ART' + str( self.numero ) + '\t'
    o += str( self.orden ) + '\t'
    o += str( self.bpm ) + '\t'
    o += str( round(self.duracion, 2) ) + '\t'
    o += str( self.dinamica ) + '\t'
    o += str( self.altura ) + '\t'
    o += str( self.letra ) + '\t'
    #o += str( self.tono) + '\t'
    o += str( self.controles) + '\n'
    return o

  def __init__(
    self,
    segmento,
    orden,
    bpm,
    programa,
    duracion,
    dinamica,
    nota,
    acorde,
    tono,
    letra,
    controles,
  ):
    self.numero = Articulacion.cantidad
    Articulacion.cantidad += 1

    self.segmento  = segmento
    self.orden     = orden
    self.bpm       = bpm
    self.programa  = programa
    self.tono      = tono
    self._dinamica  = dinamica
    self.duracion  = duracion
    self.controles = controles
```

```
44      self.altura    = nota
45      self.letra     = letra
46      self.acorde    = acorde
47
48    @property
49    def precedente( self ):
50      n = self.orden
51      o = self.segmento.articulaciones[ n - 1]
52      if  n == 0:
53        o = self.segmento.precedente.articulaciones[ - 1 ]
54      return o
55
56    def obtener( self, key ):
57        try:
58          o = getattr( self, key )
59          return o
60        except AttributeError as e:
61          return e
62
63    def cambia( self, key ):
64        este = self.obtener( key )
65        anterior = self.precedente.obtener( key )
66        if (
67          self.segmento.orden == 0
68          and self.orden == 0
69          and este
70        ):
71          return True
72        return anterior != este
73
74    @property
75    def relacion( self ):
76        return 60 / self.bpm
77
78    @property
79    def tiempo( self ):
80      # duracion en segundos
81      return self.duracion * self.relacion
82
83    @property
84    def dinamica( self ):
85      viejo_valor = self._dinamica
86      viejo_min = 0
87      viejo_max = 1
88      nuevo_min = 0
89      nuevo_max = 126
```

```
90      nuevo_valor = (
91        ( viejo_valor - viejo_min ) / ( viejo_max - viejo_min )
92      ) * ( nuevo_max - nuevo_min) + nuevo_min
93      return int( min( max( nuevo_valor, nuevo_min ), nuevo_max ) )
```

Bibliografía

BEN-KIKI, O., EVANS, C. y INGERSON, B., 2005. Yaml ain't markup language (yaml™) version 1.1. *yaml. org, Tech. Rep*, pp. 23.

BEN-KIKI OREN, E.C. y INGY, 2009. YAML Version 1.2 Specification. [en línea]. Disponible en: http://yaml.org/spec/1.2/spec.html.

CLARK, C. y TINDALE, A., 2014. Flocking: A Framework for Declarative Music-Making on the Web. *The Joint Proceedings of the ICMC and SMC*, vol. 1, no. 1, pp. 50-57.

COOMBS, J.H., RENEAR, A.H. y DEROSE, S.J., 1987. Markup Systems and the Future of Scholarly Text Processing. *Commun. ACM* [en línea], vol. 30, no. 11, pp. 933-947. ISSN 0001-0782. DOI 10.1145/32206.32209. Disponible en: http://doi.acm.org/10.1145/32206.32209.

GOOD, M., 2001. MusicXML: An Internet-Friendly Format for Sheet Music. *Proceedings of XML* [en línea], Disponible en: http://michaelgood.info/publications/music/musicxml-an-internet-friendly-format-for-sheet-music/.

GRAHAM, P., 2001. *Beating the Averages* [en línea]. 2001. Estados Unidos: Franz Developer Symposium; www.paulgraham.com. Disponible en: http://www.paulgraham.com/avg.html.

GRELA, D., 1992. *Análisis Musical: Una Propuesta Metodológica*. 1992. Rosario, Santa Fe, Argentina: Facultad de Humanidades y Artes. SERIE 5: La música en el Tiempo. Nº1.

HAUS, G. y LUDOVICO, L., 2007. Music Representation of Score, Sound, MIDI, Structure and Metadata All Integrated in a Single Multilayer Environment Based on XML.. S.l.: s.n.,

HUNT, A. y THOMAS, D., 1999. *The Pragmatic Programmer: From Journeyman to Master*. S.l.: The Pragmatic Bookshelf. ISBN 9780201616224.

HUNTLEY, G., 2019. Interprete de Comandos. [en línea]. Disponible en: https://noyaml.com/.

LEEK, J., 2017. The future of education is plain text. [en línea]. Disponible en: https://simplystatistics.org/2017/06/13/the-future-of-education-is-plain-text.

(MMA), M.M.A., 1996. Standard MIDI Files (SMF) Specification. [en línea]. Disponible en: https://www.midi.org/specifications-old/item/standard-midi-files-smf.

MOOLENAAR, B., 2000. Seven habits of effective text editing. [en línea]. Disponible en: http://moolenaar.net/habits.html.

(N.D.), 2018a. PyYAML is a full-featured YAML framework for the Python programming language. [en línea]. Disponible en: https://pyyaml.org/.

(N.D.), 2018b. The Pyhton Standar Library. [en línea]. Disponible en: https://docs.python.org/3/library/index.html.

(N.D.), 2019a. Analizador sintáctico. [en línea]. Disponible en: https://es.wikipedia.org/wiki/Analizador_sint%C3%A1ctico.

(N.D.), 2019b. Interprete de Comandos. [en línea]. Disponible en: https://es.wikipedia.org/wiki/Int%C3%A9rprete_(inform%C3%A1tica).

(N.D.), 2019c. Interprete de Comandos. [en línea]. Disponible en: https://es.wikipedia.org/wiki/Codificaci%C3%B3n_digital.

(N.D.), 2019d. Lenguaje específico de dominio. [en línea]. Disponible en: https://es.wikipedia.org/wiki/Lenguaje_espec%C3%ADfico_de_dominio.

(N.D.), 2019e. YAML Test Matrix. [en línea]. Disponible en: https://matrix.yaml.io/valid.html.

OUALLINE, S., 2001. *Vi iMproved*. S.l.: New Riders Publishing.

PENFOLD, R.A., 1992. *Advanced MIDI Users Guide*. United Kingdom: PC Publishing. ISBN 978-1870775397.

POPE, S.T., 1986. Music Notations and the Representation of Musical Structure and Knowledge. *Perspectives of New Music* [en línea], vol. 24, no. 2, pp. 156-189. DOI 10.2307/833219. Disponible en: https://www.jstor.org/stable/833219.

RAYMOND, E.S., 1997. *The Cathedral and the Bazaar*. 1997. Estados Unidos: Linux Kongress; O'Reilly Media.

RAYMOND, E.S., 1999. *The Art of UNIX Programming*. Estados Unidos: Addison-Wesley Professional. ISBN 978-0131429017.

ROSSUM, G.V., 2018. Python 3.7. [en línea]. Disponible en: https://docs.python.org/3/.

SELFRIDGE-FIELD, E., 1997. *Beyond MIDI: The Handbok of Musical Codes*. Estados Unidos: The MIT Press. ISBN 9780262193948.

STEYN, J., 2001. Music Markup Language. [en línea]. Disponible en: https://steyn.pro/mml.

TORVALDS, L. y HAMANO, J., 2010. Git: Fast version control system. *URL http://git-scm. com*,

VAN ROSSUM, G. y DRAKE JR, F.L., 1995. *Python tutorial*. S.l.: Centrum voor Wiskunde en Informatica Amsterdam.

VARIEGO, J., 2018. *Composición algorítmica. Matemáticas y ciencias de la computación en la creación musical*. S.l.: Universidad Nacional de Quilmes. ISBN 978-987-558-502-7.

WALL, L., 1999. Perl, the first postmodern computer language. [en línea]. Disponible en: https://www.perl.com/pub/1999/03/pm.html/.

WILD, J., 1996. A Review of the Humdrum Toolkit: UNIX Tools for Musical Research, created by David Huron. *Music Theory Online*, vol. 2, no. 7.

YZAGUIRRE, G., 2016. Manifiesto del Laboratorio de Software Libre. [en línea]. Disponible en: https://labsl.multimediales.com.ar/Manifiesto_del_Laboratorio_de_Software_Libre_.html.