

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

GRAMÁTICA FORMAL PARA PLAN DE OBRA MUSICAL Y
ENTORNO DE SECUENCIACIÓN

Presentado por: Lisandro Fernández

Abstract

Definición de gramática formal basada en texto plano serializado, estructurada como árbol de análisis para representar planes de obra musical. Acompañada por el desarrollo de un contexto de herramientas para interfaz de línea de comandos (CLI) destinada a generar secuencias musicales en el protocolo MIDI.

Mayo 2019

Buenos Aires, Argentina

Contendios

1	Resumen	2
2	Introducción	3
2.1	Necesidades / Requerimientos	3
2.1.1	¿Por qué Texto Plano?	3
2.1.2	¿Por qué Interfaz de Linea de Comandos?	4
2.2	Motivacion	4
2.3	Antecedentes	5
2.3.1	MuseData	5
2.3.1.1	Organización de archivos MuseData	5
2.3.1.2	La representación MuseData de información musical	6
2.3.2	Humdrum	7
2.3.2.1	Representación	7
2.3.2.2	Manipulación	7
2.3.2.3	De la experiencia a la apreciación	8
2.3.2.4	CLI vs GUI	8
2.3.3	MusicXML	9
2.3.4	Music Markup Language	9
2.3.5	Flocking	9
2.3.5.1	Como funciona Flocking	10
2.3.5.2	Programación declarativa	10
3	Metodología	12
3.1	Diagrama de procedimiento	12
3.2	Desarrollo	12
3.2.1	YAML	13
3.2.2	Python	13
3.2.3	midiUTIL	13
3.2.4	Otras herramientas	13
4	Resultados	14
4.1	Gramática	14
4.1.1	Estructura gramatical, representación de relaciones jerárquicas	14
4.1.2	Vocabulario	14
4.1.2.1	Propiedades de Pista	14
4.1.2.2	Propiedades de Unidad	16
4.1.2.3	Propiedades de Articulaciones	19
4.2	Implementación	21
4.2.1	Diagrama de arquitectura	21
4.2.2	Secciones de pricipales del desarrollo	22
4.2.2.1	Clase Pista	22
4.2.2.2	Recursion principal	23

4.3	Demostraciones	23
4.3.1	Melodia Simple	23
4.3.1.1	YAML	23
4.3.1.2	Partitura	23
4.3.1.3	Gráfico	23
4.3.2	Multiples Canales	23
4.3.2.1	YAML	23
4.3.2.2	Partitura	23
4.3.2.3	Gráfico	23
4.3.3	Polimetría	23
4.3.3.1	YAML	23
4.3.3.2	Partitura	24
4.3.3.3	Gráfico	24
5	Conclusiones	25
5.1	Pruebas / Entrevistas	25
6	Apéndice	25
6.1	pista.py	25
6.2	main.py	32
7	Bibliografía	41

1 Resumen

El presente plan propone definir una gramática formal basada en texto plano serializado¹ y descriptivo, estructurada como árbol de análisis² con el fin de representar planes de obra musical.

Acompañada por el desarrollo de un contexto de herramientas para interprete de línea de comandos (Command Line Interface) para producción de secuencias MIDI³ a partir de manipular información subscripta a dicha representación.

El desarrollo se documentará⁴ para que su publicación cumpla con las premisas del software libre.⁵

Explicar estructura del texto, que se discute en cada parte

¹Coombs, Renear y De Rose (1987)

²Grela (1992)

³Penfold (1992)

⁴Kernighan y Plauger (1978) Capítulo 8: Documentation (p.141-55)

⁵Varios (2001)

2 Introducción

Introducir a los temas q se discutiran en esta sección.

A continuación se argumentan los aspectos clave de este proyecto.

2.1 Necesidades / Requerimientos

Antes de discutir cualquier cosa, resumiremos algunas características / requerimientos importantes que son relevantes a nuestro trabajo.

Esto no agota todo los asuntos, y otros van aparecer mientras se vuelven relevantes pero nos da un criterio para empezar.

2.1.1 ¿Por qué Texto Plano?

“...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)

Algunas ventajas del texto plano y legible en contraste a la codificación de datos.⁶

Aprovechar. Potencialmente cualquier herramienta de computo puede operar información almacenada en texto plano.

Mínimo Común Denominador. Soportado en múltiples plataformas, cada sistema operativo cuenta con al menos un editor de texto todos compatibles hasta la codificación.

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 toda obsolescencia (o compatible con el avance). Los archivos de datos en formatos legibles y autodescriptivos perduran por sobre

⁶Hunt y Thomas (1999) Capítulo 3: Basic Tools (pp. 72-99).

otros formatos aun cuando caduquen las aplicaciones con las hayan sido creados.⁷

2.1.2 ¿Por qué Interfaz de Linea de Comandos?

Primer estado operativo de un ordenador. 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 de las instrucciones para esta interfaz *aplicación - argumento - recurso* (su analogía *verbo - adverbio - sujeto*) persiste para cualquier pieza de software. Dicha recurrencia elimina el ejercicio que significa operar de modo distinto cada aplicación, permitiendo 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.

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 (guión).

Pipeline y Automatización. La composición flujos de procesos complejos encadenando resultados con trabajos.⁸

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

Productividad. Valerse de herramientas pulidas como editores de texto avanzados (VIM / Emacs) 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.⁹

2.2 Motivacion

Este proyecto procura establecer un contexto y proveer los recursos para un procedimiento sencillo y flexible de elaboración discursos musicales unificando la planificación de obra con la secuenciación MIDI.

⁷Leek (2017)

⁸Raymond (1999) Capítulo 1: Context, Apartado 1: Philosophy, Sub-apartado: Basics of the Unix Philosophy (pp. 34-50)

⁹Moolenaar (2000)

Ademas pretende exponer las ventajas de la Interfaz de Linea de Comandos para operar sistemas informáticos 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.¹⁰¹¹

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 MuseData¹² es un proyecto y a la vez el sistema de codificación principal del Centro de Investigación Asistida por Computador en Humanidades (CCARH). 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 acomodan 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 un formato a otro así como también de una edición a otra.

Los archivos MuseData 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.

Las partes de los archivos MuseData siempre tienen la etiqueta 01 para la primera parte, 02 para la segunda parte de la partitura, etc. Conteniendo varias líneas de música, como dos flautas en una partitura de orquesta, o dos sistemas para música de piano. Archivos para diferentes los movimientos de una composición se encuentran en directorios separados que usualmente indican el número de movimiento, p. 01, 02, etc.

¹⁰Raymond (1997) Capítulo 11: The Social Context of Open-Source Software (p. 11)

¹¹Yzaguirre (2016)

¹²Selfridge-Field (1997)

La exhaustividad de la información dentro de los archivos varía entre dos niveles que en archivos MuseData llamamos Stage 1 y Stage 2. Sólo los archivos Stage 2 son recomendados para aplicaciones serias.

El primer paso en la entrada de datos (Stage 1) captura información básica como duración y altura de las notas. Por ejemplo, normalmente habría cuatro archivos (Violín 1, Violín 2, Viola, Violonchelo) para cada movimiento de un cuarteto de cuerdas. Si el movimiento del cuarteto comienza en metro binario, cambia a metro ternario, y luego vuelve a binario, cada sección métrica tendrá su propio conjunto de partes. Así habría doce archivos para el movimiento. El segundo paso en la entrada de datos (Stage 2) suministra toda la información que no puede ser capturado de forma fiable desde un teclado electrónico. Esto incluye indicaciones para ritmo, dinámica y articulación.

El juicio humano se aplica en el Stage 2. Así, cuando el movimiento del cuarteto de cuerdas citado anteriormente se convierte a la Stage 2, las tres secciones métricas para cada instrumento capturado desde la entrada del teclado se encadenará en un movimiento cada uno. El movimiento tendrá ahora cuatro archivos de datos (uno para Violín 1, otro para Violín 2, Viola, Violonchelo).

El juicio humano también proporciona correcciones y anotaciones a los datos. Algunos tipos de errores (por ejemplo, medidas incompletas) deben corregirse y así consiguen tener sentido para el usuario. Los asuntos que son más discrecionales (tales como alteraciones opcionales de los ornamentos o accidentes) por lo general no se modifica. Las decisiones discrecionales se anotan en archivos que permiten marcas editoriales.

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, 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 puede 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 software 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 Humdrum¹³ 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 es un conjunto de herramientas de línea de comandos, es el lenguaje de programa agnóstico. Muchos han empleado herramientas de Humdrum en secuencias de comandos más grandes que utilizan PERL, Ruby, Python, Bash, LISP y C++.

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.

- Su 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

¹³Wild (1996)

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 combinación* 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 historiadores han sido 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 parecer intimidantes para no programadores, muchos de los cuales pueden ser disuadidos de hacer uso de un herramienta de otra manera valiosa.

Independientemente de que los teóricos de la música decidan o no aumentar su invaluable 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

MusicXML¹⁴ fue diseñado desde cero para compartir archivos de música entre aplicaciones y para 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)¹⁵ 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. Muchos módulos aún están incompletos y necesitan más investigación y atención.

Si una pieza musical está 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.

2.3.5 Flocking

Flocking¹⁶ 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 existentes y

¹⁴Good (2001)

¹⁵Steyn (2001)

¹⁶Clark y Tindale (2014)

potenciales tecnologías. Los usuarios interactúan con Flocking 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 y puntajes 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 Como funciona Flocking

El núcleo del framework Flocking consiste en varios componentes interconectados que proporcionan la capacidad esencial de interpretar e instanciar generadores de unidades, producir flujos de muestras y programar procesos. Los principales componentes de Flocking incluyen:

1. el *Intérprete Flocking*, que analiza e instancia sintetizadores, generadores de unidad y buffers
2. el *Ecosistema*, que representa el audio general y su configuración
3. *Audio Strategies*, que son las salidas de audio conectables (vinculados a los backends como la API de audio web o ALSA en Node.js)
4. *Unit Generators* (ugens), que son funciones primitivas generadoras de las muestras utilizadas para producir sonido
5. *Synths* (sintetizadores) que representan instrumentos y colecciones en la lógica de generación de señales
6. el *Scheduler* (programador ó secuenciador), que gestiona el cambio secuencial (basado en el tiempo) eventos en un sintetizador

2.3.5.2 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. Tradicional-

mente 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.

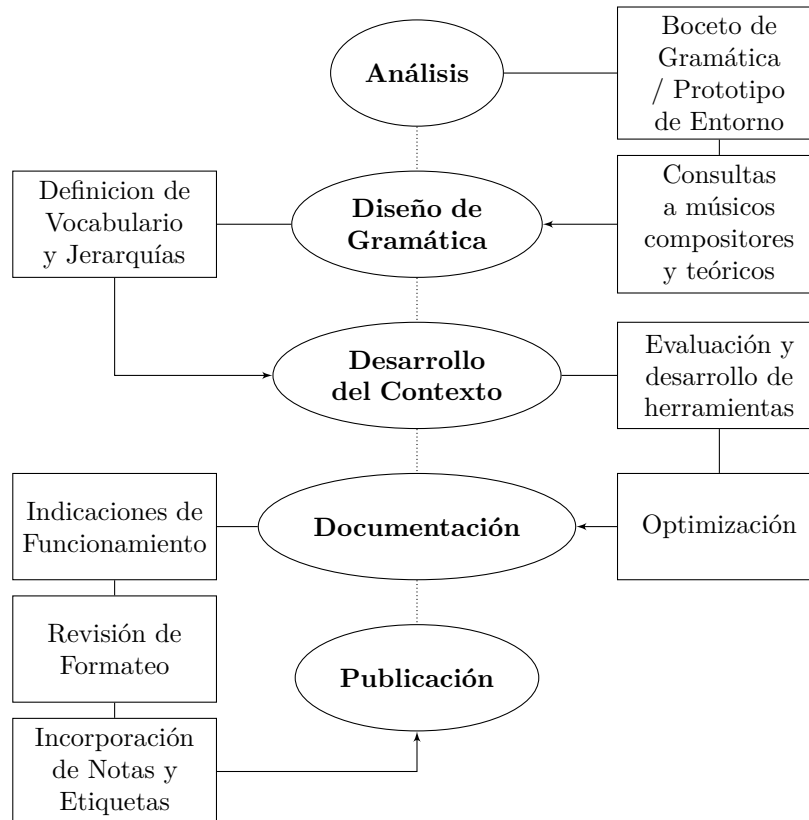
Por el contrario, la programación declarativa implica la capacidad de escribir programas que están representados en un formato que pueden ser procesados por otros programas como datos ordinarios. La familia de lenguajes Lisp es un ejemplo bien conocido de este enfoque. Paul Graham describe la naturaleza 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”.¹⁷ 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.

¹⁷Graham (2001)

3 Metodología

introduccion a la seccion, explicar que se van a discutir las herramientas usadas en cada subseccion.

3.1 Diagrama de procedimiento



Sobre el desarrollo El entorno de producción musical que se pretende establecer estará principalmente integrando por:

descripcion general del trabajo

3.2 Desarrollo

Sobre el desarrollo como conseguir el codigo. Instalacion Uso

Sobre el desarrollo

3.2.1 YAML

El estandar YAML¹⁸ como opción para serializar las definiciones de cada parte instrumental.

3.2.2 Python

La rutina de instrucciones principales será interpretada en el lenguaje Python¹⁹ (en su ultima versión estable). Esta pieza de software estará basada en otros dos desarrollos: el módulo “*pyyaml*”²⁰ para analizar la información serializada, en combinación con la librería “*music21*”²¹ que asistirá en las tareas de musicología. Además se incorporan algunos módulos de la “*Librería Estandar*”,²² mientras que la documentación se generará con “*sphinx*”.²³

3.2.3 midiUTIL

midi

3.2.4 Otras herramientas

El editor de texto preferido para toda la actividad será VIM;²⁴ durante el desarrollo las versiones se controlarán con el sistema GIT²⁵ y el repositorio del proyecto se almacenará en un espacio online proveido por algún servicio del tipo GitLab.

¹⁸Varios (2018c)

¹⁹Rossum (2018)

²⁰Varios (2018a)

²¹Cuthbert (2018)

²²Varios (2018b)

²³Brandl y Sphinx team (2018)

²⁴Moolenaar (2018)

²⁵Torvalds (2018)

4 Resultados

[] Introduccion a los temas discutidos en cada sub seccion [] Gramatica [] Aplicacion [] Demostracion

4.1 Gramática

4.1.1 Estructura grámatical, representación de relaciones jerarquías

referir a Metodologia, YAML > La estructura principal la sintaxis gramatical de cada pista se basa en el formato de serializacion de datos YAML²⁶ el cual delimita entre clave y valor con el carácter “:” (dos puntos), mientras que la indentacion representa jerarquias, relacion de pertenencia entre parametros.

Multiples ficheros .yaml equivalen a multiples pistas en el resultado MIDI.

Describir Referencia y Recurrencia en YAML

«: *base (Para que otra pista herede estas propiedades)

4.1.2 Vocabulario

explicar q se va a describir cada palabra elegida para representar cada propiedad, etiqueta, el tipo de dato q es, un ejemplo y el valor defacto que se asigna

4.1.2.1 Propiedades de Pista

Los parametros generales de cada pista son tres: el rotulo, la paleta de unidades disponibles y el primer nivel de la forma musical. A partir del primer nivel estructural, las unidades se organizan entre ellas.

nombre (cadena de carcteres) default: empty

Titulo de la pista

```
nombre: 'Piano'
```

complementos (cadena de carcteres) default: < empty >

Ubicación de fichero con modulo de usuario.

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

forma (lista de cadenas de caracteres) default: T0 D0

Macro estructura de la pista. Lista de unidades a ser secuenciadas. Corresponde a un elemento de la paleta.

²⁶Varios (2018c)

```

forma: [
  'intro',
  'estrofa',
  'estribo',
  'coda',
]

```

unidades (diccionario)

default: T0 D0

Paleta de estructuras para secuenciar. En dos tipos de unidades, las que definen las estructuras minimas y las que invocan otras unidades ademas de sobrescribir o no alguno de sus parametros.

```

unidades:
  base: &base
  clave:
  alteraciones: -2
  modo: 1
  registracion: [
    -12,-10, -9, -7, -5, -3, -2,
    0, 2, 3, 5, 7, 9, 10,
    12, 14, 15, 17, 19, 21, 22,
    24
  ]
  alturas: [ 1, 3, 5, 8 ]
  voces:
    - [ 8, 6 ]
    - [ 5 ]
    - [ 3 ]
  transportar: 60 # C
  transponer: 0
  duraciones: [ 1 ]
  bpm: 62
  metro: 4/4
  desplazar: 0
  reiterar: 0
  dinamicas: [ 1, .5, .4 ]
  canal: 3
  programa: 103
  controladores: [ 70:80, 70:90, 71:120 ]
a: &a
  <<: *base
  metro: 2/4
  alturas: [ 1, 3,0, 5, 7, 8 ]
  duraciones: [ 1, .5, .5, 1, 1 ]
b: &b
  <<: *base

```



```

metro: 6/8
duraciones: [ .5 ]
alturas: [ 1, 2 ]
voces: 0
revertir: [ 'duraciones', 'dinamicas' ]
transponer: 3
clave:
  alteraciones: 2
  modo: 1
fluctuacion:
  min: .1
  max: .4
desplazar: -1
bˆ:
  <<: *b
  dinamicas: [ .5, .1 ]
  revertir: [ 'alturas' ]
# Unidad de unidades ( UoUs )
# Propiedades sobrescriben a las de las unidades referidas
A:
  unidades: [ 'a', 'b' ]
  reiterar: 3
B: &B
  metro: 9/8
  unidades: [ 'a' , 'bˆ' ]
  desplazar: -0.75
Bˆ:
  <<: *B
  voces: 0
  bmp: 89
  unidades: [ 'b', 'a' ]
  dinamicas: [ 1 ]
estrofa:
  unidades: [ 'A', 'B', 'Bˆ' ]
coro:
  bpm: 100
  unidades: [ 'B', 'Bˆ', 'a' ]

```

4.1.2.2 Propiedades de Unidad

Parametros por defecto para todas sas unidades, pueden ser sobrescritos.

forma (lista) default: None

Estructura de la Unidad

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

<code>clave</code> (diccionario)	default: <code>alteraciones:0, modo: 0</code>
<p>Armadura de clave Cantidad de alteraciones en la armadura de clave y modo de la escala. Los numeros positivos representan sostenidos mientras que los se refiere a bemoles con números negativos. -2 = Bb, -1 = F, 0 = C, 1 = G, 2 = D, modo: 0 (mayor) Modo de la escala, 0 = Mayor o 1 = Menor (referir midi util doc, key signature)</p>	
<hr/> <pre>clave: alteraciones: -2 modo: 1</pre> <hr/>	
<code>registracion</code> (lista de enteros)	default: <code>[1]</code>
<p>Registración fija. Secuencia de intervalos a ser recorrida por el punteros de altura.</p>	
<hr/> <pre>registracion: [-12,-10, -9, -7, -5, -3, -2, 0, 2, 3, 5, 7, 9, 10, 12, 14, 15, 17, 19, 21, 22, 24]</pre> <hr/>	
<code>transportar</code> (número entero)	default: 0
<p>Transportar Ajuste de alturas. Semitonos</p>	
<hr/> <pre>transportar: 60 # C</pre> <hr/>	
<code>transponer</code> ()	default: 0
<p>Transponer puntero de registracion Ajuste de alturas pero dentro del set registracion.</p>	
<hr/> <pre>transponer: 1</pre> <hr/>	
<code>metro</code> (cadena de caracteres)	default: 4/4
<p>Clave de compás Clave de metrica. representando una fracción (numerador / denominador).</p>	
<hr/> <pre>metro: 4/4</pre> <hr/>	
<code>desplazar</code> (número decimal)	default: 0
<p>Ajuste temporal Desfazage temporal del momento en el que originalmente comienza la unidad. offset : + / - offset con la "posicion" original 0 es que</p>	

donde debe acontecer originalmente "-2" anticipar 2 pulsos o ".5" demorar medio pulso

```
desplazar: -2
```

reiterar (número entero) default: 1
Repeticiones Cantidad de veces q se toca esta unidad. Reiterarse a si misma, no es trasferible, no se hereda, caso contrario se reiterarian los referidos.

```
reiterar: 3
```

dinamicas (lista de números decimales) default: [1]
Dinámica Lista ordenada de dinámicas.

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

fluctuacion (diccionario) default: min:1, max:1
Fluctuación fluctuciones dinámicas.

```
fluctuacion:  
  min: .3  
  max: .7
```

revertir (lista de cadenas de caracteres) default: [none]
Sentido de las listas Revierte parametros del tipo lista. Deben corresponderse a la etiqueta de otro parametro del tipo lista.

```
revertir: [ 'duraciones', 'dinamicas' ]
```

canal (número entero) default: 1
Canal MIDI Número de Canal MIDI.

```
canal: 3
```

4.1.2.3 Propiedades de Articulaciones

Parametros por defecto para todas las unidades, pueden ser sobrescritos.

alturas (lista de números enteros) default: [1]
Punteros del set de registracion. Cada elemento equivale a el numero de intervalo.

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

voces (lista de números enteros) default: TO DO
Superposicion de altura Apilamiento de alturas. Lista de listas, cada voz es un lista que modifica intervalo. voz + altura = numero de intervalo.

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

duraciones (lista de números decimales) default: [1]
Duracion Lista ordenada de duraciones.

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

BPMs (número entero) default: 60
Pulso Tempo, Pulsos Por Minuto.

```
bpm: 62
```

dinamicas (lista de números decimales) default: [1]
Dinámica Lista ordenada de dinámicas.

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

programas (número entero) default: 1
Instrumento MIDI Número de Instrumento MIDI en el banco actual.

```
programa: 103
```

`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]
-

4.2 Implementación

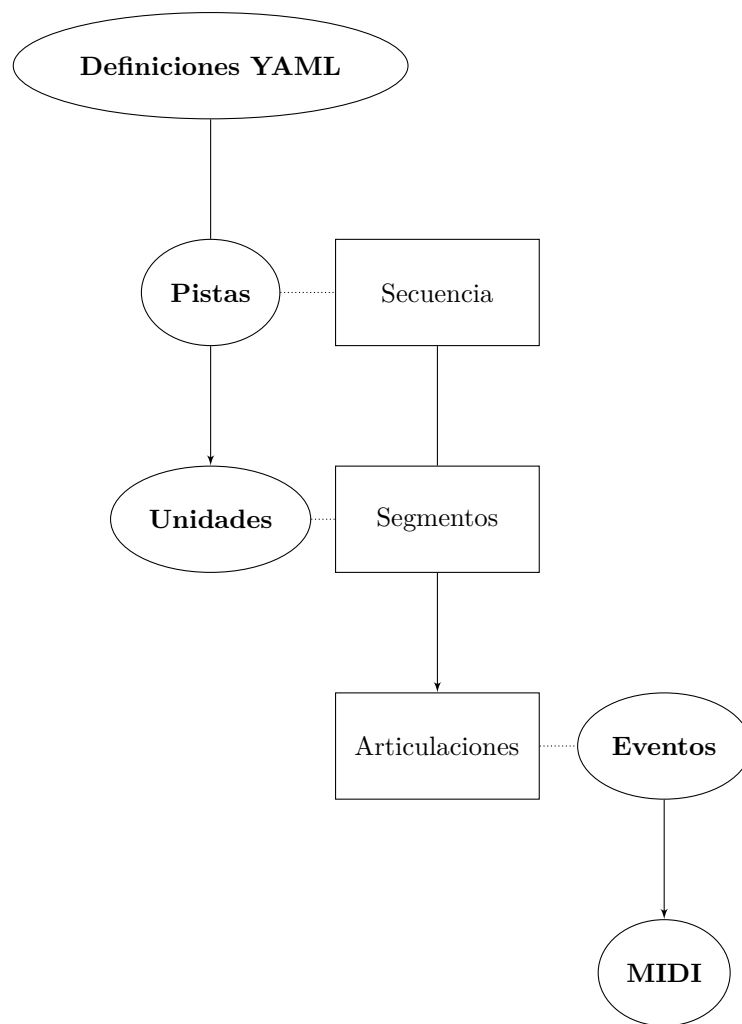
Introducción a la subsección

Aplicación y entorno de secuenciación

Lee archivos YAML como argumentos posicionales crea “pistas” a partir de ellos

explicar estructura pista como flujo de eventos agrupados en segmentos agrupados en secciones

4.2.1 Diagrama de arquitectura



4.2.2 Secciones de principales del desarrollo

Explicacion de los bloques de codigo mas representativos

4.2.2.1 Clase Pista

Clase Pista a partir de cada defefinicion de canal (.yml)

tienen un nombre parametros defaults de unidadad llamados “base” tiene una lista de unidades que se llama “macroforma” a partir de esta lista de busca en la paleta de unidades

a su vez cada unidad puede tener una lista de unidades a la que invoca arma un arbol de registros con las relaciones entre unidades arma una “sucecion” o “herencia” de parametros

repite la unidad (con sus hijas) segun parametro reiteracion agrega a los registros

Si la unidad actual tiene unidades sobrescribe los parametros de la unidad “hija” con los sucesion recursivamene busca hasta encontrar una sin unidades HIJAS
Si la unidad altual NO tiene unidades finalmente mezcla el resultado con los defaults la secuencia hace secuencia de eventos

4.2.2.2 Recursion principal

Loop principal que toma unidades previamente analizadas y llena lista de eventos.

4.3 Demostraciones

Explicacion de que ejemplo o demostracion se va a discutir en cada seccion.

4.3.1 Melodia Simple

Descripcion

4.3.1.1 YAML

Código

4.3.1.2 Partitura

Captura

4.3.1.3 Gráfico

ploteo

4.3.2 Multiples Canales

Descripcion

4.3.2.1 YAML

Códigos

4.3.2.2 Partitura

Capturas

4.3.2.3 Gráfico

ploteos

4.3.3 Polimetría

Patterns con duraciones no equivalentes

4.3.3.1 YAML

Códigos

4.3.3.2 Partitura

Capturas

4.3.3.3 Gráfico

ploteos

5 Conclusiones

5.1 Pruebas / Entrevistas

Algunos casos de pruebas de usuarios para conseguir producir musica con este desarrollo

Entrevistas del tipo no estructuradas, por pautas y guías.

Pautas / guías :

- Background
 - Experiencia con representación de información musical textual
 - * Relación con manipulacion musical a traves de parametros.
- Predisposición a trabajar (leer/escribir) con musica que se encuentre descripta en formato textual

6 Apéndice

6.1 pista.py

```
1 from argumentos import args, verboseprint, Pifie
2 import random
3 import sys
4
5 class Pista:
6     """
7     Clase para cada definicion de a partir de archivos .yaml
8     YAML => Pista => Canal
9     """
10    cantidad = 0
11    defactos = {
12        'bpm' : 60,
13        'canal' : 1,
14        'programa' : 1,
15        'metro' : '4/4',
16        'alturas' : [ 1 ],
17        'tonos' : [ 0 ],
18        'clave' : { 'alteraciones' : 0, 'modo' : 0 },
19        'intervalos' : [ 1 ],
20        'voces' : None,
21        'duraciones' : [ 1 ],
22        'desplazar' : 0,
23        'dinamicas' : [ 1 ],
24        'fluctuacion' : { 'min' : 1, 'max' : 1 },
```

```

25     'transportar'      : 0,
26     'transponer'      : 0,
27     'controles'       : None,
28     'reiterar'        : 1,
29     'referente'       : None,
30     'afinacionNota'   : None,
31     'sysEx'           : None,
32     'uniSysEx'        : None,
33     'NRPN'            : None,
34     'RPN'             : None,
35 }
36
37 def __init__(
38     self,
39     nombre,
40     paleta,
41     macroforma,
42 ):
43     self.nombre        = nombre
44     self.orden         = Pista.cantidad
45     Pista.cantidad += 1
46
47     self.macroforma    = macroforma
48     self.paleta        = paleta
49     self.registros     = {}
50     self.secuencia     = []
51     self.ordenar()
52
53     #self.oid          = str( self.orden ) + self.nombre
54     #self.duracion    = 0
55
56     #self.secuencia = self.ordenar( macroforma )
57
58     verboseprint( '\n#### ' + self.nombre + ' ####' )
59
60 def __str__( self ):
61     o = ''
62     for attr, value in self.__dict__.items():
63         l = str( attr ) + ':' + str( value )
64         o += l + '\n'
65     return o
66
67 """
68 Organiza unidades según relacion de referencia
69 """
70 def ordenar(

```

```

71     self,
72     forma      = None,
73     nivel      = 0,
74     herencia   = {},
75 ):
76     forma = forma if forma is not None else self.macroforma
77     nivel += 1
78     """
79     Limpiar parametros q no se heredan.
80     """
81     herencia.pop( 'unidades', None )
82     herencia.pop( 'reiterar', None )
83
84     """
85     Recorre lista ordenada unidades principales.
86     """
87     error = "PISTA \"" + self.nombre + "\""
88     for unidad in forma:
89         verboseprint( '-' * ( nivel - 1 ) + unidad )
90         try:
91             if unidad not in self.paleta:
92                 error += " NO ENCUENTRO \"" + unidad + "\" "
93                 raise Pifie( unidad, error )
94             pass
95             unidad_objeto = self.paleta[ unidad ]
96             """
97             Cuenta recurrencias de esta unidad en este nivel.
98             TODO: Que los cuente en cualquier nivel.
99             """
100             recurrencia = sum(
101                 [ 1 for r in self.registros[ nivel ] if r[ 'nombre' ] == unidad ]
102             ) if nivel in self.registros else 0
103             """
104             Diccionario para ingresar al arbol de registros.
105             """
106             registro = {
107                 'nombre'      : unidad,
108                 'recurrencia' : recurrencia,
109                 'nivel'       : nivel,
110             }
111
112             """
113             Si el referente está en el diccionario herencia registrar referente.
114             """
115             if 'referente' in herencia:
116                 registro[ 'referente' ] = herencia[ 'referente' ]

```

```

117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
"""
Crea parametros de unidad combinando originales con herencia
Tambien agrega el registro de referentes
"""
sucesion = {
    **unidad_objeto,
    **herencia,
    **registro
}
"""
Cantidad de repeticiones de la unidad.
"""
reiterar = unidad_objeto[ 'reiterar' ] if 'reiterar' in unidad_objeto else 1
# n = str( nivel ) + unidad + str( reiterar )
for r in range( reiterar ):
    self.registros.setdefault( nivel , [] ).append( registro )

    if 'unidades' in unidad_objeto:
        """
        Si esta tiene parametro "unidades", refiere a otras unidades "hijas"
        recursión: pasar de vuelta por esta funcion.
        """
        sucesion[ 'referente' ] = registro
        self.ordenar(
            unidad_objeto[ 'unidades' ],
            nivel,
            sucesion,
        )

    else:
        """
        Si esta unidad no refiere a otra unidades,
        Unidad célula o "unidad seminal"
        """
        """
        Combinar "defectos" con propiedades resultantes de unidad + "herencia" y registro
        """
        factura = {
            **Pista.defectos,
            **sucesion,
        }
        """
        Secuenciar articulaciones
        """
        self.secuencia += self.secuenciar( factura )

```

```

163         except Pifie as e:
164             print(e)
165
166         """
167         Genera una secuencia de ariculaciones musicales
168         a partir de unidades preprocesadas.
169         """
170     def secuenciar(
171         self,
172         unidad
173     ):
174
175         """
176         Cambia el sentido de los parametros del tipo lista
177         TODO: ¿convertir cualquier string o int en lista?
178         """
179         revertir = unidad[ 'revertir' ] if 'revertir' in unidad else None
180         if isinstance( revertir , list ):
181             for r in revertir:
182                 if r in unidad:
183                     unidad[ r ].reverse()
184         elif isinstance( revertir , str ):
185             if revertir in unidad:
186                 unidad[ revertir ].reverse()
187
188         intervalos = unidad[ 'intervalos' ]
189         duraciones = unidad[ 'duraciones' ]
190         dinamicas = unidad[ 'dinamicas' ]
191         alturas = unidad[ 'alturas' ]
192         tonos = unidad[ 'tonos' ]
193         voces = unidad[ 'voces' ]
194         ganador_voces = max( voces, key = len) if voces else [ 0 ]
195         capas = unidad[ 'controles' ]
196         ganador_capas = max( capas , key = len) if capas else [ 0 ]
197
198         """
199         Evaluar que parametro lista es el que mas valores tiene.
200         """
201         candidatos = [
202             dinamicas,
203             duraciones,
204             alturas,
205             ganador_voces,
206             ganador_capas,
207             tonos,
208         ]

```

```

209 ganador = max( candidatos, key = len )
210 pasos = len( ganador )
211 secuencia = []
212 for paso in range( pasos ):
213     """
214     Consolidad "articulacion" a partir de combinar parametros: altura,
215     duracion, dinamica, etc.
216     """
217     duracion = duraciones[ paso % len( duraciones ) ]
218     """
219     Variaciones de dinámica.
220     """
221     rand_min = unidad['fluctuacion']['min'] if 'min' in unidad[ 'fluctuacion' ] else None
222     rand_max = unidad['fluctuacion']['max'] if 'max' in unidad[ 'fluctuacion' ] else None
223     fluctuacion = random.uniform(
224         rand_min,
225         rand_max
226     ) if rand_min or rand_max else 1
227     """
228     Asignar dinámica.
229     """
230     dinamica = dinamicas[ paso % len( dinamicas ) ] * fluctuacion
231     """
232     Alturas, voz y superposición voces.
233     """
234     altura = alturas[ paso % len( alturas ) ]
235     tono = tonos[ paso % len( tonos ) ]
236     acorde = []
237     nota = 'S' # Silencio
238     if altura != 0:
239         """
240         Relacion: altura > puntero en el set de intervalos; Trasponer dentro
241         del set de intervalos, luego Transportar, sumar a la nota resultante.
242         """
243         transponer = unidad[ 'transponer' ]
244         transportar = unidad[ 'transportar' ]
245         nota = transportar + intervalos[ ( ( altura - 1 ) + transponer ) % len( intervalos ) ]
246         """
247         Armar superposicion de voces.
248         """
249         if voces:
250             for v in voces:
251                 voz = ( altura + ( v[ paso % len( v ) ] ) - 1 ) + transponer
252                 acorde += [ transportar + intervalos[ voz % len( intervalos ) ] ]
253
254     """

```

```

255     Cambios de control.
256     """
257     controles = []
258     if capas:
259         for capa in capas:
260             controles += [ capa[ paso % len( capa ) ] ]
261
262     """
263     TO DO: en vez de pasar toda la unidad:
264     extraer solo los paramtros de la articulacion:
265
266     desplazar
267     changeNoteTuning
268     changeTuningBank
269     changeTuningProgram
270     sysEx
271     uniSysEx
272     NPR ( Numeroe Parametros No Registrados )
273     NRPN: Numero de Parametro No Registrado
274     """
275
276     """
277     Articulación a secuenciar.
278     """
279     articulacion = {
280         **unidad, # TO DO: Limpiar, pasa algunas cosas de mas aca...
281         # extraer parametros de unidad y agregarlos si es (1er articulacion de
282         # la unidad) o no segun corresponda
283         'unidad'      : unidad[ 'nombre' ],
284         'orden'       : paso,
285         'altura'      : nota,
286         'tono'        : tono,
287         'acorde'      : acorde,
288         'duracion'    : duracion,
289         'dinamica'    : dinamica,
290         'controles'   : controles,
291     }
292     secuencia.append( articulacion )
293     return secuencia

```

6.2 main.py

```
1 from argumentos import args, verboseprint, Pifie
2 import yaml
3 from pista import Pista
4 from datetime import datetime, timedelta
5 import math
6
7 formato_tiempo = '%H:%M:%S'
8
9 """
10 Lee ficheros YAML declarados argumentos posicionales
11 """
12 def leer_yamls():
13     defs = []
14     for archivo in args.archivos:
15         data = open( archivo.name, 'r' )
16         try:
17             y = yaml.load( data )
18             defs.append( y )
19         except Exception as e:
20             print(e)
21             print( "=" * 80)
22     return defs
23 DEFS = leer_yamls()
24
25 """
26 A partir de cada definicion agrega una "Pista"
27 """
28 PISTAS = []
29 for d in DEFS:
30     pista = Pista(
31         d[ 'nombre' ],
32         d[ 'unidades' ],
33         d[ 'macroforma' ],
34     )
35     PISTAS.append( pista )
36
37 """
38 Extrae referentes recursivamente
39 """
40 def referir(
41     refs,
42     o = None,
43 ):
```

```

44     referente = refs[ 'referente' ]     if 'referente' in refs else None
45     nombre    = refs[ 'nombre' ]       if 'nombre'   in refs else None
46     frecuencia = refs[ 'frecuencia' ]   if 'frecuencia' in refs else None
47     nivel     = refs[ 'nivel' ]         if 'nivel'    in refs else None
48     output    = o                       if o is not None else [ None ] * nivel
49     output[ nivel - 1 ] = ( nombre, frecuencia )
50     if referente:
51         referir( referente, output )
52     return output
53
54 PARTES = []
55
56 """
57 Generar canal MIDI a partir de cada pista
58 """
59 EVENTOS = []
60 for pista in PISTAS:
61     momento = 0
62     track = pista.orden
63     EVENTOS.append([
64         'addTrackName',
65         track,
66         momento,
67         pista.nombre
68     ])
69
70     EVENTOS.append([
71         'addCopyright',
72         track,
73         momento,
74         args.copyright
75     ])
76
77     # Ploteo
78     comienzo = datetime.strptime(
79         str( timedelta( seconds = 0 ) ),
80         formato_tiempo
81     )
82     parte = {
83         'orden'      : track,
84         'nombre'     : pista.nombre,
85         'comienzo'   : comienzo,
86         'etiquetas'  : [],
87     }
88     duracion_parte = 0
89

```

```

90     """
91     Loop principal:
92     Genera una secuencia de eventos MIDI lista de articulaciones.
93     """
94     for index, articulacion in enumerate( pista.secuencia ):
95
96         """
97         TO DO: agregar funciones de midiutil adicionales:
98         https://midiutil.readthedocs.io/en/1.2.1/class.html#classref
99         [x] addCopyright
100        [x] addPitchWheelEvent
101        [x] changeNoteTunig
102        [ ] changeTuningBank
103        [ ] changeTuningProgram
104        [x] addSysEx
105        [x] addUniversalSysEx
106        [x] makeNRPNCall
107        [x] makeRPNCall
108        """
109
110        verboseprint( articulacion )
111        precedente = pista.secuencia[ index - 1 ]
112        unidad      = articulacion[ 'unidad' ]
113        canal       = articulacion[ 'canal' ]
114        bpm         = articulacion[ 'bpm' ]
115        metro       = articulacion[ 'metro' ].split( '/' )
116        clave       = articulacion[ 'clave' ]
117        programa    = articulacion[ 'programa' ]
118        duracion    = articulacion[ 'duracion' ]
119        tono        = articulacion[ 'tono' ]
120
121        """
122        Primer articulación de la parte, agregar eventos fundamentales: pulso,
123        armadura de clave, compás y programa.
124        """
125        if ( index == 0 ):
126            EVENTOS.append([
127                'addTempo',
128                track,
129                momento,
130                bpm
131            ])
132
133        """
134        Clave de compás
135        https://midiutil.readthedocs.io/en/1.2.1/class.html#midiutil.MidiFile.MIDIFile.addTime

```

```

136     denominator = potencia negativa de 2: log10( X ) / log10( 2 )
137     2 representa una negra, 3 una corchea, etc.
138     """
139     numerador      = int( metro[0] )
140     denominador    = int( math.log10( int( metro[1] ) ) ) / math.log10( 2 )
141     relojes_por_tick = 12 * denominador
142     notas_por_pulso = 8
143     EVENTOS.append([
144         'addTimeSignature',
145         track,
146         momento,
147         numerador,
148         denominador,
149         relojes_por_tick,
150         notas_por_pulso
151     ])
152
153     EVENTOS.append([
154         'addKeySignature',
155         track,
156         momento,
157         clave[ 'alteraciones' ],
158         # multiplica por el n de alteraciones
159         1,
160         clave[ 'modo' ]
161     ])
162
163     EVENTOS.append([
164         'addProgramChange',
165         track,
166         canal,
167         momento,
168         programa
169     ])
170
171     """
172     TO DO: Crear estructura superiores a articulacion llamada segmento
173     parametros de que ahora son relativos a la articulacion #0
174     """
175     """
176     Primer articulacion de la Unidad,
177     inserta etiquetas y modificadores de unidad (desplazar).
178     """
179     if ( articulacion[ 'orden' ] == 0 ):
180         desplazar = articulacion[ 'desplazar' ]
181         # TODO raise error si desplazar + duracion es negativo

```

```

182 momento += desplazar
183
184 """
185 Compone texto de la etiqueta a partir de nombre de unidad, numero de
186 iteración y referentes
187 """
188 texto = ''
189 ers = referir( articulacion[ 'referente' ] ) if articulacion[ 'referente' ] != None else None
190 prs = referir( precedente[ 'referente' ] ) if precedente[ 'referente' ] != None else None
191 for er, pr in zip( ers , prs ):
192     if er != pr:
193         texto += str( er[ 0 ] ) + ' #' + str( er[ 1 ] ) + '\n'
194 texto += unidad
195 EVENTOS.append([
196     'addText',
197     track,
198     momento,
199     texto
200 ])
201 """
202 changeNoteTuning
203 """
204 if articulacion[ 'afinacionNota' ]:
205     EVENTOS.append([
206         'changeNoteTuning',
207         track,
208         articulacion[ 'afinacionNota' ][ 'afinaciones' ],
209         articulacion[ 'afinacionNota' ][ 'canalSysEx' ],
210         articulacion[ 'afinacionNota' ][ 'tiempoReal' ],
211         articulacion[ 'afinacionNota' ][ 'programa' ],
212     ])
213 """
214 SysEx
215 """
216 if articulacion[ 'sysEx' ]:
217     EVENTOS.append([
218         'addSysEx',
219         track,
220         momento,
221         articulacion[ 'sysEx' ][ 'fabricante' ],
222         articulacion[ 'sysEx' ][ 'payload' ],
223     ])
224 """
225 UniversalSysEx
226 """
227 if articulacion[ 'uniSysEx' ]:

```

```

228     EVENTOS.append([
229         'addUniversalSysEx',
230         track,
231         momento,
232         articulacion[ 'uniSysEx' ][ 'codigo' ],
233         articulacion[ 'uniSysEx' ][ 'subCodigo' ],
234         articulacion[ 'uniSysEx' ][ 'payload' ],
235         articulacion[ 'uniSysEx' ][ 'canal' ],
236         articulacion[ 'uniSysEx' ][ 'tiempoReal' ],
237     ])
238     """
239     Numero de Parametro No Registrado
240     """
241     if articulacion[ 'NRPN' ]:
242         EVENTOS.append([
243             'makeNRPNCall',
244             track,
245             canal,
246             momento,
247             articulacion[ 'NRPN' ][ 'control_msb' ],
248             articulacion[ 'NRPN' ][ 'control_lsb' ],
249             articulacion[ 'NRPN' ][ 'data_msb' ],
250             articulacion[ 'NRPN' ][ 'data_lsb' ],
251             articulacion[ 'NRPN' ][ 'ordenar' ],
252         ])
253
254     """
255     Numero de Parametro Registrado
256     """
257     if articulacion[ 'RPN' ]:
258         EVENTOS.append([
259             'makeRPNCall',
260             track,
261             canal,
262             momento,
263             articulacion[ 'RPN' ][ 'control_msb' ],
264             articulacion[ 'RPN' ][ 'control_lsb' ],
265             articulacion[ 'RPN' ][ 'data_msb' ],
266             articulacion[ 'RPN' ][ 'data_lsb' ],
267             articulacion[ 'RPN' ][ 'ordenar' ],
268         ])
269
270     # Ploteo
271     etiqueta = {
272         'texto' : texto,
273         'cuando' : momento,

```

```

274         #'hasta' : duracion_unidad,
275     }
276     parte[ 'etiquetas' ].append( etiqueta )
277     # Termina articulacion 0, estos van a ser parametros de Segmento
278
279     """
280     Agrega cualquier cambio de parametro,
281     comparar cada uno con la articulacion previa.
282     """
283     if ( precedente['bpm'] != bpm ):
284         EVENTOS.append([
285             'addTempo',
286             track,
287             momento,
288             bpm,
289         ])
290
291     if ( precedente[ 'metro' ] != metro ):
292         numerador = int( metro[ 0 ] )
293         denominador = int( math.log10( int( metro[ 1 ] ) ) ) / math.log10( 2 ) )
294         relojes_por_tick = 12 * denominador
295         notas_por_pulso = 8
296         EVENTOS.append([
297             'addTimeSignature',
298             track,
299             momento,
300             numerador,
301             denominador,
302             relojes_por_tick,
303             notas_por_pulso
304         ])
305
306     if ( precedente[ 'clave' ] != clave ):
307         EVENTOS.append([
308             'addKeySignature',
309             track,
310             momento,
311             clave[ 'alteraciones' ],
312             1, # multiplica por el n de alteraciones
313             clave[ 'modo' ]
314         ])
315
316     #if programa:
317     if ( precedente[ 'programa' ] != programa ):
318         EVENTOS.append([
319             'addProgramChange',

```

```

320         track,
321         canal,
322         momento,
323         programa
324     ])
325     #midi_bits.addText( pista.orden, momento , 'prgm : #' + str( programa ) )
326
327     if ( precedente[ 'tono' ] != tono ):
328         EVENTOS.append([
329             'addPitchWheelEvent',
330             track,
331             canal,
332             momento,
333             tono
334         ])
335
336
337     """
338     Agregar nota/s (altura, duracion, dinamica).
339     Si existe acorde en la articulación armar una lista con cada voz superpuesta.
340     o una lista de solamente un elemento.
341     """
342     voces = articulacion[ 'acorde' ] if articulacion[ 'acorde' ] else [ articulacion[ 'altu' ] ]
343     dinamica = int( articulacion[ 'dinamica' ] * 126 )
344     for voz in voces:
345         altura = voz
346         """
347         Si la articulacion es un silencio (S) agregar nota sin altura ni dinamica.
348         """
349         if voz == 'S':
350             dinamica = 0
351             altura = 0
352         EVENTOS.append([
353             'addNote',
354             track,
355             canal,
356             altura,
357             momento,
358             duracion,
359             dinamica,
360         ])
361
362
363     """
364     Agregar cambios de control
365     """

```



```

366     if articulacion[ 'controles' ]:
367         for control in articulacion[ 'controles' ]:
368             for control, valor in control.items():
369                 EVENTOS.append([
370                     'addControllerEvent',
371                     track,
372                     canal,
373                     momento,
374                     control,
375                     valor,
376                 ])
377
378
379     momento += duracion
380     duracion_parte += ( duracion * 60 ) / bpm
381
382     # Ploteo
383     parte[ 'duracion' ] = datetime.strptime(
384         str( timedelta( seconds = duracion_parte ) ).split( '.' )[0],
385         formato_tiempo
386     )
387     PARTES.append( parte )

```

7 Bibliografía

Reserva de referencias:

- 27
- 28
- 29
- 30
- 31

²⁷Allen (1983)

²⁸Schaeffer (1966)

²⁹Samaruga (2016)

³⁰Lerdahl y Jackendof (1996)

³¹Shaffer (2005)

- ALLEN, J.F., 1983. Maintaining knowledge about temporal intervals. *Communications of the ACM*, pp. 832-843. ISSN 0001-0782. DOI 10.1145/182.358434.
- BRANDL, G. y SPHINX TEAM, 2018. Python Documentation Generator. [en línea]. Disponible en: <https://sphinx-doc.org/en/master>.
- 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 DE ROSE, S.J., 1987. Markup Systems and the Future of Scholarly Text Processing. *Communications of the ACM* [en línea], vol. 30, no. 11, pp. 933-47. DOI 10.1145/32206.32209. Disponible en: <http://www.xml.coverpages.org/coombs.html>.
- CUTHBERT, M.S., 2018. music21: a toolkit for computer-aided musicology. [en línea]. Disponible en: <http://web.mit.edu/music21>.
- 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. N°1.
- HUNT, A. y THOMAS, D., 1999. *The Pragmatic Programmer: From Journeyman to Master*. S.l.: The Pragmatic Bookshelf. ISBN 9780201616224.
- KERNIGHAN, B.W. y PLAGUER, P.J., 1978. *The Elements Of Programming Style*. Estados Unidos: McGraw-Hill Book Company. ISBN 9780070342071.
- 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>.
- LERDAHL, F. y JACKENDOF, R., 1996. *A Generative Theory of Tonal Music*. Estados Unidos: The MIT Press. ISBN 026262107X.
- MOOLENAAR, B., 2000. Seven habits of effective text editing. [en línea]. Disponible en: <http://moolenaar.net/habits.html>.
- MOOLENAAR, B., 2018. VIM. [en línea]. Disponible en: <https://www.vim.org/docs.php>.
- PENFOLD, R.A., 1992. *Advanced MIDI Users Guide*. United Kingdom: PC Publishing. ISBN 978-1870775397.
- 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/>.
- SAMARUGA, L.M., 2016. *Un modelo de representación y análisis estructural de la música electroacústica*. Tesis doctoral. S.l.: Universidad Nacional de Quilmes.
- SCHAEFFER, P., 1966. *Tratado de los objetos musicales*. S.l.: s.n. ISBN 9788420685403.
- SELFIDGE-FIELD, E., 1997. *Beyond MIDI: The Handbok of Musical Codes*. Estados Unidos: The MIT Press. ISBN 9780262193948.
- SHAFFER, K., 2005. Make Stunning Schenker Graphs with GNU Lilypond. [en línea]. Disponible en: <https://www.linuxjournal.com/article/8364>.
- STEYN, J., 2001. Music Markup Language. [en línea]. Disponible en: <https://steyn.pro/mml>.
- TORVALDS, L., 2018. GIT. [en línea]. Disponible en: <https://git-scm.com/docs>.
- VARIOS, A., 2001. ¿Que es el Software Libre? [en línea]. Disponible en: <https://www.gnu.org/philosophy/free-sw.es.html>.
- VARIOS, A., 2018a. PyYAML is a full-featured YAML framework for the Python programming language. [en línea]. Disponible en: <https://pyyaml.org/>.
- VARIOS, A., 2018b. The Pyhton Standar Library. [en línea]. Disponible en: <https://docs.python.org/3/library/index.html>.
- VARIOS, A., 2018c. YAML Ain't Markup Language. [en línea]. Disponible en: <http://yaml.org/>.
- 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.