Alumno: Lisandro Fernández

Carrera: Licenciatura en Música y Tecnología

Proyecto o programa acreditado en el que se inscribe el Seminario de Investigación: Cartografías Espacio-Temporales y Arte Sonoro

Profesor tutor: Pablo Riera

Contendios

1 Resumen

El presente plan propone definir una gramática formal basada en texto plano serializado[1] y descriptivo, estructurada como árbol de análisis[2] 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 MIDI[3] a partir de manipular información subscripta a dicha representación.

El desarrollo se documentará[4] para que su publicación cumpla con las premisas del software libre.[5]

Explicar estructura del texto, que se discutite en cada parte

[1] Coombs, Renear y De Rose (1987)
[2] Grela (1992)
[3] Penfold (1992)
[4] Kernighan y Plaguer (1978) Capítulo 8: Documentation (p.141-55)
[5] Varios (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

eo

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 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 recurrencía 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]

[6] Hunt y Thomas (1999) Capítulo 3: Basic Tools (pp. 72-99).
[7] Leek (2017)
[8] Raymond (1999) Capítulo 1: Context, Apartado 1: Philosophy, Sub-apartado: Basics of the Unix Philosophy (pp. 34-50)

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 Encuestas

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

Entrevistas del tipo no estructuradas, por pautas y guías.

Pautas / guias :

- 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

2.3 Motivacion / Objetivos

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.

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.

[ˆver_raymond2] [ˆver_yzaguirre] [ˆver_raymond2]: Raymond (1997) Capítulo 11: The Social Context of Open-Source Software (p. 11) [ˆver_yzaguirre]: Yzaguirre (2016)

[9] Moolenaar (2000)

2.4 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.

2.4.1 MuseData

La base de datos MuseData[10] 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.

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.

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.

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,

[10] Selfridge-Field (1997)

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 software usa estas sugerencias de impresión y sonido en el proceso de generación de documentos de partitura y archivos MIDI.

2.4.2 Humdrum

David Huron creó Humdrum[11] 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++.

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.

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.

[11] Wild (1996)

- Más bien, *su poder deriva de la flexibilidad de su kit de elementos, utilizados en combinacióin* para explotar plenamente el potencial del sistema.

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 lentos para reconocer el potencial del sistema.

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.

2.4.3 MusicXML

MusicXML[12] 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.

2.4.4 Music Markup Language

El Lenguaje de Marcado de Música (MML)[13] 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:

[12] Good (2001)
[13] Steyn (2001)

- 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.

2.4.5 Flocking

Flocking[14] 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 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.

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 bufers
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 secunecial (basado en el tiempo) eventos en un sintetizador

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.

[14] Clark y Tindale (2014)

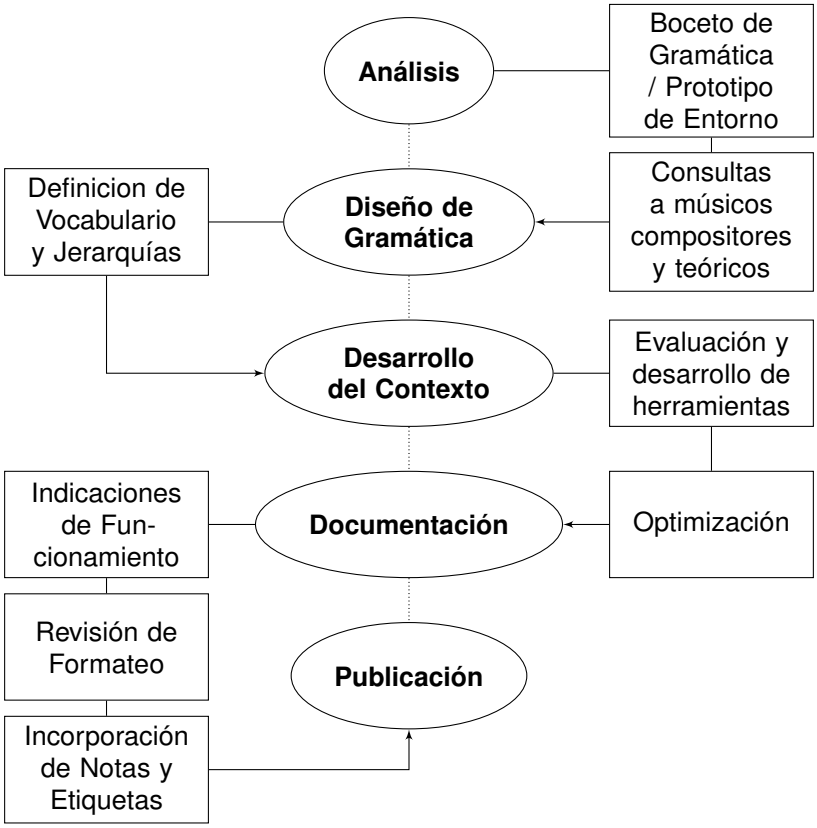
Por el contrario, la programación declarativa implica la capacidad de escribir programas que están representados en un formato que pueden ser procesados por otros programas como datos ordinarios. La familia de lenguajes Lisp es un ejemplo bien conocido de este enfoque. Paul Graham describe la naturaleza 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".[15] 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.

[15] Graham (2001)

5 Metodología

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

5.1 Explicacion del proceso



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

descripcion general del trabajo

5.1.1 Codigo, Repo

Sobre el desarrollo

5.1.2 Uso / Instalacion

Sobre el desarrollo

5.2 YAML

El estandar YAML[16] como opción para serializar las definiciones de cada parte instrumental.

5.3 Python

La rutina de instrucciones principales será interpretada en el lenguaje Python[17] (en su ultima versión estable). Esta pieza de software estará basada en otros dos desarrollos: el módulo "*pyyaml*"[18] para analizar la información serializada, en combinación con la librería "*music21*"[19] que asistirá en las

[16] Varios (2018c)
[17] Rossum (2018)
[18] Varios (2018a)
[19] Cuthbert (2018)

tareas de musicología. Ademas se incorporan algunos módulos de la "*Librería Estandar*",[20] mientras que la documentación se generará con "*sphinx*".[21]

5.4 midiUTIL

midi

5.5 otras herramientas

El editor de texto preferido para toda la actividad será VIM;[22] durante el desarrollo las versiones se controlarán con el sistema GIT[23] y el repositorio del proyecto se almacenará en un espacio online proveido por algún servicio del tipo GitLab.

6 Resultados

introduccion a los temas discutidos en cada sub seccion gramatica aplicacion demostracion

6.1 Gramática

6.1.2 Estructura grámatical, representación de relaciones jerarquícas

referir a Metodologia, YAML > La estructura principal la sintaxis gramatical de cada pista se basa en el formato de serializacion de datos YAML[24] el cual delimta entre clave y valor con el cáracter ":" (dos puntos), mientras que la indentacion representa jerarquias, relacion de pertenecia 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)

6.1.3 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

6.1.3.1 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

Título de la pista

Etiqueta: nombre.

Tipo: Cadena de caracteres.

Ejemplo

```
1 nombre: 'Pista 1'
```

Defacto: nombre del fichero?.

Forma

Lista de unidades a ser sequenciadas

Etiqueta: macroforma.

Tipo: Lista de cadenas de caracteres que corresponde a un elemento de la paleta de unidades.

[20] Varios (2018b)
[21] Brandl y Sphinx team (2018)
[22] Moolenaar (2018)
[23] Torvalds (2018)
[24] Varios (2018c)

Ejemplo

```
macroforma: [
  'intro',
  'estrofa',
  'estribo',
  'estrofa',
  'coro',
  'coro',
  'inter',
]
```

Paleta de unidades

Paleta de estructuras para secuenciar

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

Etiqueta: unidades.

Tipo: Diccionario.

Ejemplo

```
unidades:
    base: &base
      clave:
        alteraciones: -2
        modo:
      intervalos:  [
          -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 ]
      revertir: [ 'duraciones', 'dinamicas' ]
      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
      transponer: 3
      clave:
        alteraciones: 2
        modo: 1
      fluctuacion:
        min: .1
        max: .4
      desplazar: -1
    b^:
      <<: *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' , 'b^' ]
      #desplazar: -0.5
      desplazar: -0.75
    B^:
      <<: *B
      voces: 0
```

```
      bmp: 89
      unidades: [ 'b', 'a' ]
      dinamicas: [ 1 ]
    estrofa:
      unidades: [ 'A', 'B', 'B^' ]
    coro:
      bpm: 100
      unidades: [ 'B', 'B^', 'a' ]
```

6.1.3.2 Propiedades de unidad

Parametros por defecto para todas sas unidades, pueden ser sobrescritos

Armadura de clave

Catidad de alteraciones en la armadura de clave y modo de la escala.

Los numeros positivos representan sotenidos mientras que los se refiere a bemoles con números negativos. -2 = Bb, -1 = F, 0 = C, 1 = G, 2 = D, modo: 0 # Modo de la escala, 0 = Mayor o 1 = Menor

https://midiutil.readthedocs.io/en/1.2.1/class.html#midiutil.MidiFile.MIDIFile.addKeySignature

Etiqueta: clave, alteraciones y modo.

Tipo: Diccionarios de enteros.

Ejemplo

```
clave:
  alteraciones: -2
  modo: 0
```

Registración fija

Secuencia de intervalos a ser recorrida por el punteros de altura

Etiqueta: intervalos

Tipo: Lista de números enteros.

Ejemplo

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

Altura

Punteros del set de intervalos. Cada elemento equivale a el numero de intervalo.

Etiqueta: alturas.

Tipo: Lista de enteros.

Ejemplo

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

Superposicion de altura

Apilamiento de alturas. Lista de listas, cada voz es un lista que modifica intervalo. voz + altura = numero de intervalo

Etiqueta: voces.

Tipo: Lista de listas de enteros.

Ejemplo

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

Transportar

Ajuste de alturas

Etiqueta: transportar.

Tipo: Número entero.

Ejemplo

```
transportar: 60 # C
```

Tranponer

Ajuste de alturas pero dentro del set intervalos Semitonos, registración fija

Etiqueta: transponer.

Tipo: Número entero.

Ejemplo

```
transponer: 1
```

Duracion

Lista ordenada de duraciones.

Etiqueta: duraciones.

Tipo: Lista de decimales.

Ejemplo

```
[ 1, .5, .5, 1, 1 ]
```

Pulso

Tempo, Pulsos Por Minuto

Etiqueta: bpm

Tipo: Número entero.

Ejemplo

```
bpm: 62
```

Clave de compás

Clave de metrica.

Etiqueta: metro.

Tipo: Cadena de caracteres representando una fracción (numerador / denominador).

Ejemplo

```
metro: 4/4
```

Ajuste temporal

Desfazage temporal del momento en el que originalmente comienza la unidad. offset : + / - offset con la “posicion” original 0 es que donde debe acontecer originalmente “-2” anticipar 2 pulsos o “.5” demorar medio pulso

Etiqueta: desplazar.

Tipo: Número entero.

Ejemplo

```
desplazar: -2
```

Repeticiones

Catidad de veces q se toca esta unidad Reiterarse a si misma, no es trasferible, no se hereda, caso contrario se reterarian los referidos

Etiqueta: reiterar.

Tipo: Número entero.

Ejemplo

```
reiterar: 3
```

Dinámica

Lista ordenada de dinámicas

Etiqueta: dinamicas.

Tipo: Lista de número decimales.

Ejemplo

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

Fluctuación

Lista ordenada de dinámicas

Etiqueta: fluctuacion, min y max.

Default: min: 0, max: 0.

Tipo: dicionario de decimales.

Ejemplo

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

Sentido de listas

Revierte parametros del tipo lista.

Etiqueta: revertir Deben corresponderse a la etiqueta de otro parametro del tipo lista.

Tipo: Lista de cadenas de caracteres.

Ejemplo

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

Canal MIDI

Número de Canal MIDI.

Etiqueta: canal.

Tipo: Número entero.

Ejemplo

```
canal: 3
```

Instrumento MIDI

Número de Instrumento MIDI en el banco actual.

Etiqueta: programa.

Tipo: Número entero.

Ejemplo

```
programa: 103
```

Cambios de control

Secuencia de pares número controlador y valor a asignar.

Etiqueta: controles.

Tipo: Lista de listas de tuples.

Ejemplo

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

RPN

Registered Parameter Number Call

Los bancos MIDI se alternan utilizando de RPN

https://www.mutools.com/info/docs/mulab/using-bank-select-and-prog-changes.html

http://www.andrelouis.com/qws/art/art009.htm

CC#0 numero de banco, CC#32 numero de programa

Para seleccionar el instrumento #130 = 2do banco, 3º pograma

Instrumento/programa = CC#0:2, CC#32:32

CC#0:2, CC#32:2

Ejemplo

```
controles:
  - [  0 : 2 ]
  - [ 32 : 3 ]
```

NRPN

Non Registered Parameter Number Call

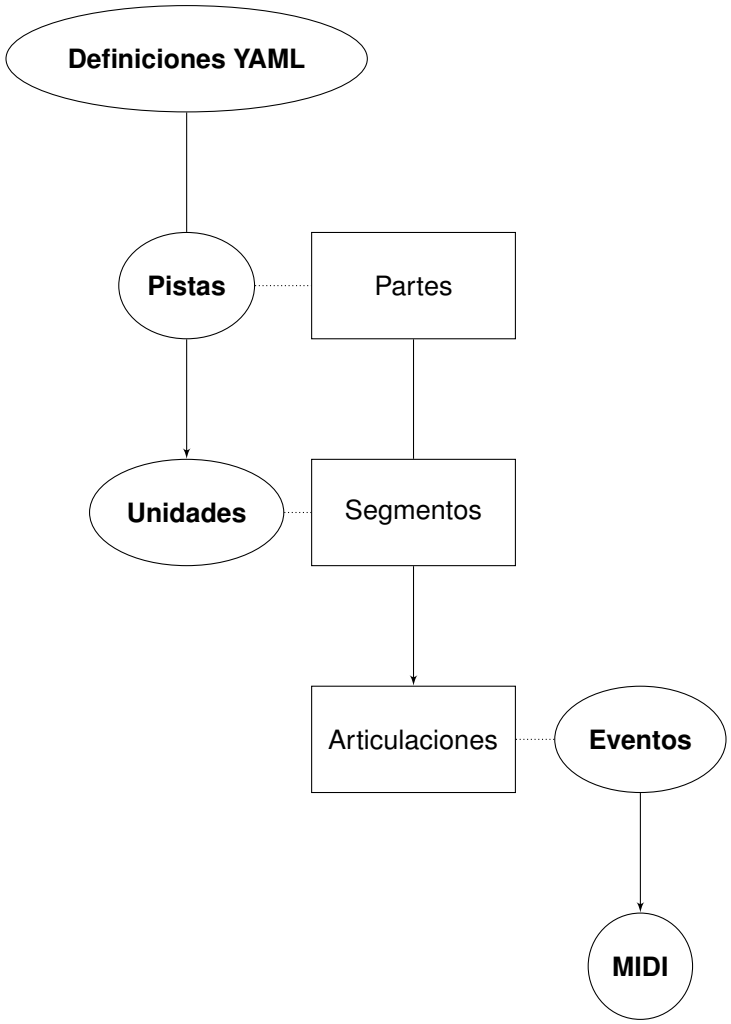
6.2 Implementación

Introduccion a la subseccion

Aplicación y entorno de secuenciación

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

6.2.1 Diagrama de arquitectura de la aplicación



6.2.2 Secciones de pricipales del desarrollo

Explicacion de los bloques de codigo mas representativos

6.2.2.1 Clase Pista (pista.py)

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

Ejemplo

```
class Pista:
  """
  Clase para cada definicion de a partir de archivos .yml
  YAML => Pista => Canal
  """
  cantidad = 0
  defactos =     'bpm'           : 60,    'canal'         : 1,    'programa'      : 1,    'metro'         : '4/4',    'altu

  def __init__(
    self,
    nombre,
    paleta,
    macroforma,
  ):
    self.nombre     = nombre
    self.orden      = Pista.cantidad
    Pista.cantidad += 1

    self.macroforma = macroforma
    self.paleta     = paleta
    self.registros  =
    self.secuencia  = []
    self.ordenar()

    #self.oid        = str( self.orden ) + self.nombre
    #self.duracion   = 0

    #self.secuencia = self.ordenar( macroforma )


  def __str__( self ):
    o = ''
    for attr, value in self.__dict__.items():
      l = str( attr ) + ':' + str( value )
      o += l + 'saltodelinea'
    return o

  """
  Organiza unidades según relacion de referencia
  Pasa cada unidad despues de analizarla por rutina para generar
  articulaciones
  """
  def ordenar(
    self,
    forma    = None,
    nivel    = 0,
    herencia = ,
  ):
    forma = forma if forma is not None else self.macroforma
    nivel += 1
    """
    Limpiar parametros q no se heredan.
    """
    herencia.pop( 'unidades', None )
    herencia.pop( 'reiterar', None )

    """
    Recorre lista ordenada unidades principales.
    """
    error =  "PISTA ¨ + self.nombre + ""
    for unidad in forma:
      verboseprint( '-' * ( nivel - 1 ) +  unidad )
      try:
        if unidad not in self.paleta:
          error +=  " NO ENCUENTRO ¨ + unidad + "¨  "
```

```
          raise Pifie( unidad, error )
          pass
        unidad_objeto = self.paleta[ unidad ]
        """
        Cuenta recurrencias de esta unidad en este nivel.
        TODO: Que los cuente en cualquier nivel.
        """
        recurrencia = sum(
          [ 1 for r in self.registros[ nivel ] if r[ 'nombre' ] == unidad ]
        ) if nivel in self.registros else 0
        """
        Dicionario para ingresar al arbol de registros.
        """
        registro =           'nombre'      : unidad,          'recurrencia' : recurrencia,          'nivel'       : nivel,

        """
        Si el referente está en el diccionario herencia registrar referente.
        """
        if 'referente' in herencia:
          registro[ 'referente' ] = herencia[ 'referente' ]

        """
        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 "defactos" con propiedas resultantes de unidad + "herencia" y registro.
            """
            factura =               **Pista.defactos,              **sucesion,
            """
            Secuenciar articulaciones
            """
            self.secuencia += self.secuenciar( factura )
      except Pifie as e:
          print(e)

  """
  Genera una secuencia de ariculaciones musicales
  a partir de unidades preprocesadas.
  """
  def secuenciar(
    self,
    unidad
  ):

    """
    Cambia el sentido de los parametros del tipo lista
    TODO: ¿convertir cualquier string o int en lista?
    """
    revertir = unidad[ 'revertir' ] if 'revertir' in unidad else None
    if isinstance( revertir , list ):
      for r in revertir:
        if r in unidad:
          unidad[ r ].reverse()
    elif isinstance( revertir , str ):
      if revertir in unidad:
        unidad[ revertir ].reverse()

    intervalos    = unidad[ 'intervalos' ]
    duraciones    = unidad[ 'duraciones' ]
    dinamicas     = unidad[ 'dinamicas' ]
    alturas       = unidad[ 'alturas' ]
    tonos         = unidad[ 'tonos' ]
    voces         = unidad[ 'voces' ]
```

```
    ganador_voces = max( voces, key = len) if voces else [ 0 ]
    capas         = unidad[ 'controles' ]
    ganador_capas = max( capas , key = len) if capas else [ 0 ]

    """
    Evaluar que parametro lista es el que mas valores tiene.
    """
    candidatos = [
      dinamicas,
      duraciones,
      alturas,
      ganador_voces,
      ganador_capas,
      tonos,
    ]
    ganador = max( candidatos, key = len )
    pasos = len( ganador )
    secuencia = []
    for paso in range( pasos ):
      """
      Consolidad "articulacion" a partir de combinar parametros: altura,
      duracion, dinamica, etc.
      """
      duracion = duraciones[ paso % len( duraciones ) ]
      """
      Variaciones de dinámica.
      """
      rand_min = unidad['fluctuacion']['min'] if 'min' in unidad[ 'fluctuacion' ] else None
      rand_max = unidad['fluctuacion']['max'] if 'max' in unidad[ 'fluctuacion' ] else None
      fluctuacion = random.uniform(
         rand_min,
         rand_max
      ) if rand_min or rand_max else 1
      """
      Asignar dinámica.
      """
      dinamica = dinamicas[ paso % len( dinamicas ) ] * fluctuacion
      """
      Alturas, voz y superposición voces.
      """
      altura = alturas[ paso % len( alturas ) ]
      tono   = tonos[ paso % len( tonos ) ]
      acorde = []
      nota = 'S' # Silencio
      if altura != 0:
        """
        Relacion: altura > puntero en el set de intervalos; Trasponer dentro
        del set de intervalos, luego Transportar, sumar a la nota resultante.
        """
        transponer  = unidad[ 'transponer' ]
        transportar = unidad[ 'transportar' ]
        nota = transportar + intervalos[ ( ( altura - 1 ) + transponer ) % len( intervalos ) ]
        """
        Armar superposicion de voces.
        """
        if voces:
          for v in voces:
            voz = ( altura + ( v[ paso % len( v ) ] ) - 1 ) + transponer
            acorde += [ transportar +  intervalos[ voz % len( intervalos ) ]  ]

      """
      Cambios de control.
      """
      controles = []
      if capas:
        for capa in capas:
          controles += [ capa[ paso % len( capa ) ] ]

      """
      TO DO: en vez de pasar toda la unidad:
      extraer solo los paramtros de la articulacion:

      desplazar
      changeNoteTuning
      changeTuningBank
      changeTuningProgram
      sysEx
      uniSysEx
      NPR ( Numeroe Parametros No Registrados )
      NRPN: Numero de Parametro No Registrado
      """

      """
      Articulación a secuenciar.
      """
      articulacion =         **unidad, # TO DO: Limpiar, pasa algunas cosas de mas aca...        # extraer parametros de se
      secuencia.append( articulacion )
    return secuencia
```

6.2.2.2 Main Loop

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

Ejemplo

```
"""
Generar eventos MIDI a partir de cada pista
"""
EVENTOS = []
for pista in PISTAS:
  momento = 0
  track = pista.orden
  EVENTOS.append([
    'addTrackName',
    track,
    momento,
    pista.nombre
  ])

  EVENTOS.append([
    'addCopyright',
    track,
    momento,
    args.copyright
  ])

  parte =      'orden'     : track,     'nombre'    : pista.nombre,     'comienzo'  : comienzo,     'etiquetas' : [],
  duracion_parte = 0

  """
  Loop principal:
  Genera una secuencia de eventos MIDI lista de articulaciones.
  """
  for index, articulacion in enumerate( pista.secuencia ):

    """
    TO DO: agregar funcciones de midiutil adicionales:
    https://midiutil.readthedocs.io/en/1.2.1/class.html#classref
    [x] addCopyright
    [x] addPitchWheelEvent
    [x] changeNoteTunig
    [ ] changeTuningBank
    [ ] changeTuningProgram
    [x] addSysEx
    [x] addUniversalSysEx
    [x] makeNRPNCall
    [x] makeRPNCall
    """

    verboseprint( articulacion )
    precedente = pista.secuencia[ index - 1 ]
    unidad     = articulacion[ 'unidad' ]
    canal      = articulacion[ 'canal' ]
    bpm        = articulacion[ 'bpm' ]
    metro      = articulacion[ 'metro' ].split( '/' )
    clave      = articulacion[ 'clave' ]
    programa   = articulacion[ 'programa' ]
    duracion   = articulacion[ 'duracion' ]
    tono       = articulacion[ 'tono' ]

    """
    Primer articulación de la parte, agregar eventos fundamentales: pulso,
    armadura de clave, compás y programa.
    """
    if ( index == 0 ):
      EVENTOS.append([
        'addTempo',
        track,
        momento,
        bpm
      ])

      """
      Clave de compás
      https://midiutil.readthedocs.io/en/1.2.1/class.html#midiutil.MidiFile.MIDIFile.addTimeSignature
      denominator  = potencia negativa de 2: log10( X ) / log10( 2 )
      2 representa  una negra, 3 una corchea, etc.
      """
      numerador        = int( metro[0] )
      denominador      = int( math.log10( int( metro[1] ) ) / math.log10( 2 ) )
      relojes_por_tick = 12 * denominador
      notas_por_pulso = 8
      EVENTOS.append([
        'addTimeSignature',
        track,
        momento,
        numerador,
        denominador,
        relojes_por_tick,
        notas_por_pulso
```

```
      ])

      EVENTOS.append([
        'addKeySignature',
        track,
        momento,
        clave[ 'alteraciones' ],
        # multiplica por el n de alteraciones
        1,
        clave[ 'modo' ]
      ])

      EVENTOS.append([
        'addProgramChange',
        track,
        canal,
        momento,
        programa
      ])

    """
    TO DO: Crear estructura superiores a articulacion llamada segmento
    parametros de que ahora son relativios a la aritulacion #0
    """
    """
    Primer articulacion de la Unidad,
    inserta etiquetas y modificadores de unidad (desplazar).
    """
    if ( articulacion[ 'orden' ] == 0 ):
      desplazar = articulacion[ 'desplazar' ]
      # TODO raise error si desplazar + duracion es negativo
      momento += desplazar

      """
      Compone texto de la etiqueta a partir de nombre de unidad, numero de
      iteración y referentes
      """
      texto = ''
      ers = referir( articulacion[ 'referente' ] ) if articulacion[ 'referente' ] != None else [ ( 0, 0 ) ]
      prs = referir( precedente[ 'referente' ] ) if precedente[ 'referente' ] != None else [ ( 0, 0 ) ]
      for er, pr in zip( ers , prs ):
        if er != pr:
          texto += str( er[ 0 ] ) + ' #' + str( er[ 1 ] ) + 'saltodelinea'
      texto += unidad
      EVENTOS.append([
       'addText',
        track,
        momento,
        texto
      ])
      """
      changeNoteTuning
      """
      if articulacion[ 'afinacionNota' ]:
        EVENTOS.append([
         'changeNoteTuning',
          track,
          articulacion[ 'afinacionNota' ][ 'afinaciones' ],
          articulacion[ 'afinacionNota' ][ 'canalSysEx' ],
          articulacion[ 'afinacionNota' ][ 'tiempoReal' ],
          articulacion[ 'afinacionNota' ][ 'programa' ],
        ])
      """
      SysEx
      """
      if articulacion[ 'sysEx' ]:
        EVENTOS.append([
         'addSysEx',
          track,
          momento,
          articulacion[ 'sysEx' ][ 'fabricante' ],
          articulacion[ 'sysEx' ][ 'playload' ],
        ])
      """
      UniversalSysEx
      """
      if articulacion[ 'uniSysEx' ]:
        EVENTOS.append([
         'addUniversalSysEx',
          track,
          momento,
          articulacion[ 'uniSysEx' ][ 'codigo' ],
          articulacion[ 'uniSysEx' ][ 'subCodigo' ],
          articulacion[ 'uniSysEx' ][ 'playload' ],
          articulacion[ 'uniSysEx' ][ 'canal' ],
          articulacion[ 'uniSysEx' ][ 'tiempoReal' ],
        ])
      """
      Numero de Parametro No Registrado
      """
```

```
176       if articulacion[ 'NRPN' ]:
177         EVENTOS.append([
178          'makeNRPNCall',
179           track,
180           canal,
181           momento,
182           articulacion[ 'NRPN' ][ 'control_msb' ],
183           articulacion[ 'NRPN' ][ 'control_lsb' ],
184           articulacion[ 'NRPN' ][ 'data_msb' ],
185           articulacion[ 'NRPN' ][ 'data_lsb' ],
186           articulacion[ 'NRPN' ][ 'ordenar' ],
187         ])
188
189       """
190       Numero de Parametro Registrado
191       """
192       if articulacion[ 'RPN' ]:
193         EVENTOS.append([
194          'makeRPNCall',
195           track,
196           canal,
197           momento,
198           articulacion[ 'RPN' ][ 'control_msb' ],
199           articulacion[ 'RPN' ][ 'control_lsb' ],
200           articulacion[ 'RPN' ][ 'data_msb' ],
201           articulacion[ 'RPN' ][ 'data_lsb' ],
202           articulacion[ 'RPN' ][ 'ordenar' ],
203         ])
204
205       etiqueta =         'texto'  : texto,        'cuando' : momento,        #'hasta' : duracion_unidad,
206       parte[ 'etiquetas' ].append( etiqueta )
207       # Termina articulacion 0, estos van a ser parametros de Segmento
208
209     """
210     Agrega cualquier cambio de parametro,
211     comparar cada uno con la articulacion previa.
212     """
213     if ( precedente['bpm'] != bpm ):
214       EVENTOS.append([
215         'addTempo',
216         track,
217         momento,
218         bpm,
219       ])
220
221     if ( precedente[ 'metro' ] != metro ):
222       numerador        = int( metro[ 0 ] )
223       denominador      = int( math.log10( int( metro[ 1 ] ) ) / math.log10( 2 ) )
224       relojes_por_tick = 12 * denominador
225       notas_por_pulso = 8
226       EVENTOS.append([
227         'addTimeSignature',
228         track,
229         momento,
230         numerador,
231         denominador,
232         relojes_por_tick,
233         notas_por_pulso
234       ])
235
236     if ( precedente[ 'clave' ] != clave ):
237       EVENTOS.append([
238         'addKeySignature',
239         track,
240         momento,
241         clave[ 'alteraciones' ],
242         1, # multiplica por el n de alteraciones
243         clave[ 'modo' ]
244       ])
245
246     #if programa:
247     if ( precedente[ 'programa' ] != programa ):
248       EVENTOS.append([
249          'addProgramChange',
250         track,
251         canal,
252         momento,
253         programa
254       ])
255     #midi_bits.addText( pista.orden, momento , 'prgm : #' + str( programa ) )
256
257     if ( precedente[ 'tono' ] != tono ):
258       EVENTOS.append([
259          'addPitchWheelEvent',
260         track,
261         canal,
262         momento,
263         tono
264       ])
265
```

```
266
267     """
268     Agregar nota/s (altura, duracion, dinamica).
269     Si existe acorde en la articulación armar una lista con cada voz superpuesta.
270     o una lista de solamente un elemento.
271     """
272     voces = articulacion[ 'acorde' ] if articulacion[ 'acorde' ] else [ articulacion[ 'altura' ] ]
273     dinamica = int( articulacion[ 'dinamica' ] * 126 )
274     for voz in voces:
275       altura = voz
276       """
277       Si la articulacion es un silencio (S) agregar nota sin altura ni dinamica.
278       """
279       if voz == 'S':
280         dinamica = 0
281         altura = 0
282       EVENTOS.append([
283         'addNote',
284         track,
285         canal,
286         altura,
287         momento,
288         duracion,
289         dinamica,
290       ])
291
292
293     """
294     Agregar cambios de control
295     """
296     if articulacion[ 'controles' ]:
297       for control in articulacion[ 'controles' ]:
298         for control, valor in control.items():
299           EVENTOS.append([
300            'addControllerEvent',
301             track,
302             canal,
303             momento,
304             control,
305             valor,
306           ])
307
308
309     momento += duracion
310     duracion_parte += ( duracion *  60 ) / bpm
311
312   PARTES.append( parte )
```

6.3 Demostraciones

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

6.3.1 Melodia Simple

descripcion

YAML

código

Partitura

Captura

Gráfico

ploteo

6.3.2 Multiples Canales

descripcion

YAML

codigos

Partitura

Capturas

Gráfico

ploteos

6.3.2 Polimetría

Paterns con duraciones no equivalentes

YAML

codigos

Partitura

Capturas

Gráfico

ploteos

9 Bibliografía

Reserva de referencias:,[25],[26][27],[28]

[25] Allen (1983)
[26] Schaeffer (1966)
[27] Samaruga (2016)
[28] Lerdahl y Jackendof (1996)

10 Apéndice

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.coverpagess.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. Nº1.

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 Elemenets Of Programing 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.

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., 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.