

Teoría de Lenguajes

Primer Cuatrimestre de 2015

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 2

Grupo Estado Final

Integrante	LU	Correo electrónico
Gorojovsky, Román	530/02	rgorojovsky@gmail.com
Lazzaro, Leonardo	147/05	lazzaroleonardo@gmail.com

Introducción

El trabajo consiste en implementar un programa que tome un archivo escrito en el lenguaje Musileng, definido por la cátedra, y convertirlo en un archivo midi, pasando por un lenguaje intermedio que es interpretado por el programa *midicomp* para generar finalmente el archivo midi.

El problema, entonces puede dividirse en tres subproblemas:

- Convertir la descripción del lenguaje Musileng en una gramática bien definida.
- Aprender a usar alguna biblioteca preexistente para convertir esa gramática en código
- Convertir la salida del *parser* creado en los dos pasos anteriores en el lenguaje de *midicomp*

Decisiones tomadas

Se eligió usar *PLY* debido a nuestro mejor manejo del lenguaje *Python* y se usó como esqueleto del trabajo el código presentado en clase:

- `lexer.rules.py` contiene las definiciones y reglas de tokens
- `lexer.py` es el código de ejecución del *lexer* (usado principalmente para testeo).
- `parser.rules` contiene la definición de la gramática
- `expressions.py` contiene los objetos que se crean a partir del análisis sintáctico.
- `parser.py` es el código de ejecución del *parser*

Como se verá en la sección que detalla la implementación, la conversión de objetos al lenguaje *midicomp* está implementada dentro de los objetos de `expressions.py`, usando patrones de programación orientada a objetos.

Gramática

Se define la siguiente gramática, para cuya definición se priorizó la simplicidad en la construcción de los objetos que luego se usarán para generar el archivo *midicomp* por sobre la legibilidad de la gramática. En general se pusieron los terminales (puntos y coma, llaves de cierre) en las producciones “de más afuera”.

Notamos en **negrita** los tokens y en **negrita bastardilla** los tokens con algún valor, definidos más abajo.

```
Musileng → DefTempo DefCompas Constantes Voces
DefTempo → #tempo Duracion num
DefCompas → #compas num/num
Constantes → λ | constante ; constantes
Constante → const constante = num
Voces → Voz } | Voz } Voces
Voz → Voz (Var) } ListaCompases
ListaCompases → ListaCompases CompOREpe
CompOREpe → Compases | Repetir
Repetir → repetir (num) { Compases }
Compases → Compas } | Compas } Compases
Compas → compas { Figuras
Figuras → Figura ; | Figura ; Figuras
Figura → Nota | Silencio
Nota → nota (altura, Var, duracion)
Silencio → silencio (duracion)
Var → constante | num
```

Los tokens con valor son:

```
num = 0|[1-9][0-9]*
duracion = (redonda|blanca|negra|corchea|semicorchea|fusa|semifusa)(.)?
altura = (do|re|mi|fa|sol|la|si)(-|+)?
constante = [a-zA-Z]+
```

La gramática que se implementa en `parser.rules.py` tiene una pequeña diferencia con esta: no existe el no terminal “ComoOREpe”, que se agregó acá para simplificar la lectura, pero está implementado directamente en la lista de compases.

Implementación

Estructura

El lenguaje se representa con una combinación de objetos y listas de estos objetos. Como se verá más adelante, estos objetos tienen métodos para la

validación semántica del archivo y para la conversión a *midicomp*.

Un archivo parseado queda en un objeto **Musileng** que es una tupla (**DefTempo**, **DefCompas**, [**Constantes**] [**Voces**]), y que se crea al reconocer la primer producción. **DefTempo** y **DefCompas** tienen la información correspondiente a las siguientes dos producciones; las constantes se toman como una lista de (**nombre**, **valor**) y se convierten en un diccionario en el momento de construir el **Musileng**; y finalmente viene la lista de voces con el resto de la información.

Una voz, representada en la clase **Voz**, consiste en la identificación del instrumento y una lista de compases. Los **repetir** (N) se implementan en el momento del parseo (en **parser_rules.py** en vez de **expressions.py**) generando N veces la lista interna de compases. Cada **Compas** consiste en una lista de objetos **Figura**, que pueden ser de la subclase **Nota** o **Silencio**.

Para definir las figuras (y también para las definiciones de los compases y del *tempo*) hay dos clases: **Duracion** y **Altura**, que representan respectivamente el ritmo (redonda, blanca, negra, con o sin puntillo, etc.) y la altura (do, re, mi, sostenido, bemol, etc.) de cada figura. Obviamente los silencios sólo tienen duración.

Validación

La validación de la “partitura” se hace en forma *top-down* llamando a una colección de métodos **validar** que levantan una excepción con un mensaje (bastante) descriptivo cuando detectan un error. En la clase **Musileng**, se llama al **validar()** de cada una de las voces, pasándoles la definición del compás esperado¹, más un número de voz, usado para los mensajes de error. Este método hace el pasamanos de esta información, más un número de compás, también para los mensajes de error.

El **validar()** de cada compás hace la suma de las duraciones de sus figuras y la compara con la duración esperada por la definición de compás. Cada objeto **Duracion** “sabe” cuál es su duración numérica: 1 para las redondas, 1/2 para las blancas, 1/4 para las negras, etc. multiplicado por 1.5 en caso de haber puntillo. El **Compas** levanta una excepción si la suma es mayor o menor a lo esperado.

Conversión a formato *midicomp*

Una vez más, la implementación de esta tarea queda repartida en diversas clases, en las funciones **get_midicomp()**. La clase **Musileng** lo único que sabe hacer es escribir el header, pidiéndole la definición del *tempo* en clicks a **DefTempo**, que implementa la fórmula del enunciado. Luego se van anexando los resultados del **get_midicomp()** de cada voz.

Los objetos **Voz** reciben su id (un número asignado secuencialmente, saltando el 10²), los clicks por redonda, calculados según el enunciado y los pulsos por compás. De esta información, lo único que usa **Voz** es el id, para escribir el

¹Se implementó la verificación de que todas las voces tengan la misma cantidad de compases, pero dado que uno de los ejemplos de la cátedra viola esta regla, la verificación quedó comentada

header. Los demás datos, los pasa hacia cada uno de sus compases, junto con un nro. de compás, también generado secuencialmente. Cada compás genera su `midicomp` y este se va anexando al de la voz.

La clase **Compas** debe mantener, además del *string* con el texto generado, el pulso y click actual dentro del compás. Para esto la clase **Figura** recibe en su `get_midicomp()` el pulso y click actual, además de los clicks por redonda y pulsos por compás definidos “arriba” en **Musileng**. A partir de estos datos, cada **Figura**, sabe dónde empieza y puede calcular dónde termina: primero se calcula el total de clicks que ocupará la figura en función de su duración y los clicks por redonda; a partir de este valor, el pulso donde termina la nota es

$$pulso_{final} = pulso_{inicial} + \lceil clicks_{total} / clicks_por_pulso \rceil$$

y el click es

$$clicks_{final} = clicks_{total} \% clicks_por_pulso$$

donde $clicks_por_pulso = 384$.

Se hace la excepción de que si $pulso_{final} = pulsos_por_compas$, se escribe que la figura finaliza en el 00:000 del siguiente compás. El `midicomp` de cada figura se completa con la altura correspondiente a cada nota y el volumen, 70 en el caso de las notas y 0 en el de los silencios. En una primer versión de esta conversión los objetos **Silencio** solamente avanzaban pulsos y clicks, generando un **string** vacío, pero fue necesario cambiar esto para la implementación de uno de los ejemplos, como se verá dos secciones más adelante.

Uso del programa

Tests y ejemplos

²Con respecto a la percusión sólo se tuvo este cuidado de no generar una pista 10. No se implementó nada especial para manejar voces de percusión ni se implementaron ejemplos con percusión