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 Musilen, 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 Musilen 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.

Musilen → 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
CopmORepe → 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 m'etodos para la

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

Un archivo parseado queda en un objeto `Musilen` 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 moment de construír el `Musilen`; 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 tambi'en 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 m'etodos `validar` que levantan una excepción con un mensaje (bastante) descriptivo cuando detectan un error. En la clase `MusiLen`, se llama al `validar()` de cada una de las voces, pasándoles la definición del compás esperado[0], más un número de voz, usado para los mensajes de error. Este m'etodo hace el pasamanos de esta información, más un número de compás, tambi'en 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 num'erica: 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` levana 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 `MusiLen` lo único que sabe hacer es escribir el header

[1] Se 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

Uso del programa

Tests y ejemplos