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

Eligimos usar *PLY* debido a nuestro mejor manejo del lenguaje *Python* y usamos 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_rulesi.py` 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

Definimos la siguiente gramática, para cuya definición priorizamos 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 pusimos 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 "CompORepe", 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 construír 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 esperado[1], 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` 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 `Musileng` lo único que

[1] Implementamos 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

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, salteando el 10[2]), 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 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 volúmen, 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 para la implementación de uno de los ejemplos, como se verá dos secciones más adelante.

Tests y ejemplos

El *testing* del programa es a tres niveles: para el *lexer* usamos un *unit test* que verifica que cada token se reconozca correctamente. En este nivel no hubo demasiadas dificultades, tan solo algunos temas de precedencia que llevaban a que, por ejemplo, "redonda" fuera interpretado como "Altura: re, Altura: do, Constante: nda" en vez de como "Duracion: redonda".

Para el parser encontramos que era muy difícil aislar cada posible producción, por lo que recurrimos a tests del sistema completo, es decir ejemplos. Estos están en el directorio `files`, y son los cuatro ejemplos de la cátedra, doce ejemplos de archivos con errores y tres ejemplos correctos. Todos los archivos siguen el formato de los de la cátedra: `<titulo>_input.txt`.

[2]Con respecto a la percusión sólo tuvimos este cuidado de no generar una pista 10. No hay implementado nada especial para manejar voces de percusión ni presentamos ejemplos con percusión

Los ejemplos erróneos están nombrados `erroneo_$i` con `$i` del 1 al 12 y son:

1. Constante definida dos veces
2. Repetir vacío
3. Repetir con N = 0
4. Sin definición de *tempo*
5. Sin definición de compas
6. Con una constante indefinida
7. Sin voces
8. Con una voz vacía
9. Con un compás vacío
10. Falta un ; en una figura
11. Un compás es demasiado corto
12. Un compás es demasiado largo

Todos producen un error a nivel sintáctico o semántico (en los métodos `validar()`). El mensaje que corresponde a los primeros intenta informar las posibles causas semánticas de la falla que reporta el parser.

En cuanto a los ejemplos correctos, hay que empezar por decir que nuestro conocimiento de escritura musical es limitado, apenas superior a lo explicado en el enunciado: Unos rudimentarios conocimientos de rítmica, la capacidad de saber qué altura es cuál si contamos las partituras y renglones y una ligera sospecha de algunos otros elementos. Con esas armas nos arriesgamos a implementar algunos ejemplos a partir de partituras buscadas en internet.

La elección del primer ejemplo a implementar nos resultó tan obvia a nosotros como a quienes nos conocen: el solo de *Ji Ji Ji*, de Patricio Rey y sus Redonditos de Ricota, que todos atribuímos al saxo, y sin embargo en el disco está grabado con la guitarra. Tuvimos que adaptar y corregir la partitura que encontramos, hasta lograr que sonara más o menos parecida a lo que escuchábamos en el disco. El ejemplo aparece titulado `jijiji`.

El sigiente ejemplo es un fragmento de otro clásico de nuestro rock: el *riff* de *Post Crucifixión* de Pescado Rabioso. Aquí nos encontramos con una limitación de *Musileng*, ya que el cierre del fragmento contiene dos *síncopas*: notas que empiezan en un compás y terminan en otro, y esto en el lenguaje no se puede implementar. Lo emparchamos definiendo los compases como $\frac{12}{8}$ en vez de la correcta $\frac{6}{8}$, con lo que una de las síncopas queda dentro del compás, aunque la última resulta insalvable. Este ejemplo se titula `post_crucifixion_riff`.

Por último se adaptó sin demasiadas dificultades la pieza *4:33* del compositor contemporáneo John Cage. Fue para lograr que este ejemplo quede correcto que se modificó la conversión de los `Silencio` a `midicomp` para que escriba todos los tiempos, pues de otra manera la pieza no tenía la duración correcta.

Uso del programa

El ejecutable principal es el script `musileng`, que implementa la interfaz definida en el enunciado. Para simplificar el testeo implementamos un script `hacer_midi.sh` que recibe el título de un ejemplo, intenta generar el `midicomp` y, si lo logra, genera el midi y lo deja en el directorio `midi`, como `<titulo>.mid`. Opcionalmente puede pasársele un `-p` para reproducir el midi generado. Dentro del script se define qué programa se usa para cada uno de estos pasos, en particular, usamos *wildmidi* para la reproducción.

Hay un script extra, `hacer_todo.sh` que ejecuta el script anterior para todos los ejemplos definidos, escribiendo en pantalla, para los ejemplos erróneos, qué error se estaría probando.