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

# TP2 - Musileng

Teoría de Lenguajes - TP2

**GRUPO Los Sin Grupo**

Juan Enríquez - LU 36/08 - juanenriquez@gmail.com
Damián Furman - LU 936/11 - damian.a.furman@gmail.com

**Resumen**

En el presente informe mostramos como se llevó a cabo la implementación de un lexer y un parser para el lenguaje musileng el cual nos permite una escritura amigable de archivos midi a través de la generación de un archivo intermedio que además se genera por nuestro programa.

**Keywords**

Lenguaje musical - Musileng - MIDI - Parsing - LALR - PLY

# 1. Introduccion

Dentro del marco de el reconocimiento y la generación de lenguajes vimos un número de técnicas que nos permiten estudiar la estructura sintáctica de un lenguaje. Estas técnicas nos permiten comprender un lenguaje desde su estructura a partir de una gramática. Es decir, podemos chequear (o incluso generar) un lenguaje a partir de un conjunto acotado de reglas que nos dicen como se construye dicho lenguaje.

En nuestro caso, se nos pidió realizar un programa que pueda ser capaz de *reconocer* un lenguaje llamado *Musileng*.

Musileng es un lenguaje que posibilita la realización de composiciones musicales en formato MIDI a través de una especificación dada que luego será convertida a un formato intermedio para ser a su vez procesado oportunamente por otro programa que se encargará de generar el archivo de audio en el formato antes mencionado.

# 2. Gramática

Parte del trabajo necesario para poder cumplir con la tarea asignada era poder definir la gramática del lenguaje Musileng. En nuestro caso, si bien contamos con una especificación del lenguaje y una descripción de sus estructuras de control no contamos con la definición formal de la gramática, motivo por el cual procedimos a definirla nosotros mismos. A continuación se detalla la gramática que se dedujo a partir de la información recibida.

Sea $G$ una gramática libre de contexto tal que $G :< V_n, V_t, P, S >$ con $P$ definido como:

S → ENCABEZADO CONSTANTES VOCES
ENCABEZADO → TEMPO COMPAS_DEF
TEMPO → #FIGURA numero
FIGURA → blanca | redonda | negra | corchea | semicorchea | fusa | semifusa
COMPAS_DEF → #compas numero/numero
CONSTANTES → const string = numero; CONSTANTES | $\lambda$
VOCES → DECLA_INST { MUSICA }
DECLA_INST → voz(numero)
MUSICA → COMPAS MUSICA | BUCLE MUSICA | $\lambda$
COMPAS → compas { NOTAS }
BUCLE → repetir(numero) { MUSICA }
NOTAS → FIGURA NOTAS | $\lambda$
FIGURA → NOTA | SILENCIO
NOTA → nota(ALTURA, numero, DURACION);
ALTURA → NOTAID SIMBOLO
NOTAID → do | re | mi | fa | sol | la | si
SIMBOLO → + | − | $\lambda$
DURACION → FIGURA | FIGURA •
SILENCIO → silencio(DURACION);

# 3. Implementación

Una vez que contamos con la gramática definida para el lenguaje en cuestión teníamos dos puntos fundamentales para resolver.

Por un lado implementar un parser para poder comprender la estructura sintáctica de musileng y, por otro lado, era necesario poder darle semántica a lo que estabamos tratando de comprender para poder generar ciertas estructuras y recavar cierta información útil de lo que se leía para poder generar un archivo intermedio en otro lenguaje para poder, efectivamente, componer música a través un programa externo.

Respecto de la primera tarea lo que hicimos fue utilizar por recomendación de la cátedra un *generador de parser y lexer* llamado PLY. PLY es un envoltorio para el lenguaje de programación Python de históricos lexers y parser de Unix llamados `lex` y `yacc`.

Lex es un analizador léxico. Es un programa cuya tarea es tomar una secuencia de caracteres y retornar una secuencia de *tokens*, es decir, una secuencia de símbolos pero que tienen un sentido particular. Por su parte, yacc es un generador de parsers que nos permite a partir de un conjunto de reglas (ie. la definición de una gramática) generar un parser de tipo LALR que reconoce el lenguaje descripto por las reglas mencionadas.

Para el lexer de nuestro lenguaje se procedió con la implementación de un conjunto de reglas que lo que hacen es partir el archivo de entrada en tokens con un significado relevante para el contexto de nuestro programa. En nuestro caso, queremos que los strings `do`, `re` o `sol` no sean simplemente esos strings sino que sean tokens que reflejen cierto comportamiento común de acuerdo al contexto de uso, en particular para nuestro programa son `NOTAID` que es un token que representa la ocurrencia de una nota musical dentro del programa. Lo mismo ocurre con cosas como `const grand_piano = 1;`. Esto no representa cualquier cosa sino que simboliza la definición de una constante con una *keyword* particular y un nombre con un valor y otros símbolos que permiten identificar esta construcción sintáctica.

En una etapa posterior se hizo algo similar con el parser. Yacc nos pide definir las reglas de la gramática necesaria para operar sintácticamente con el lenguaje a estudiar. Estas reglas se corresponden con la gramática definida en la sección anterior pero están definidas a partir de los tokens que se establecieron en el primer paso. Adicionalmente, las reglas que se definen permiten agregarle comportamiento al parser y poder así escribir el archivo de salida y así también poder validar ciertas cuestiones relacionadas con la composición musical como respetar la duración de los compases y validar su estructura o que las constantes que se usan estén definidas previamente.

El parser devuelve una estructura que contiene los datos necesarios para traducir el input a un lenguaje que pueda ser procesado por el programa *midicomp* que lo compila luego en un archivo .midi listo para ser reproducido. La estructura

que devuelve nuestro parser es la siguiente:

El parser devuelve una lista con el siguiente formato:

[ [ tempo, compas ] , compases_y_bucles ]

donde **tempo** es una lista que tiene como primer elemento a la figura y como segundo elemento la cantidad de esa figura por minuto (ejemplo: [redonda", 60])

**compas** es una lista con dos elementos: el numerador primero y el denominador del compas (por ejemplo para 2/4 es [2, 4] o para 3/4 es [3, 4])

**compases_y_bucles** es una lista de dos elementos que difieren según si se trata de un compás o de un bucle:

Si es un compás, el primer elemento de la lista es una 'C' para indicar que se trata de un compás y el segundo elemento es una lista de **diccionario_de_notas_y_silencios**

Si es un bucle, el primer elemento de la lista es una 'B' para indicar que se trata de un bucle y el segundo elemento es una lista de compases o de otros bucles

el **diccionario_de_notas_y_silencios** representa una nota o un silencio. Los campos que tiene cada diccionario son:

**duration**: es el tiempo que dura la nota ya calculado (el campo se usa para calcular la duración del compás). Este numero esta calculado de la siguiente manera: 1 / valor_de_la_figura. Es decir que si la duration de una nota es 0,5 es una blanca, si es 0,25 es una negra, si es 1 es una redonda, etc.

**type**: pueden ser dos valores: 'NOT' si el diccionario representa una nota y 'SIL' si representa un silencio

**octava**: sólo en caso de que sea nota, tiene el valor de la octava. Importante tener en cuenta que siempre es un valor numerico, no importa que haya sido declarada la nota usando directamente ese valor o usando una constante

**nota**: sólo en caso de que sea nota, tiene la nota como do, o re o mi, etc...

**desv**: también sólo existe este campo en caso de que se trate de una nota y representa la desviación. Sus posibles valores son: disminuido(-), aumentado(+) o nada. En este último caso, desv es un string vacío

Esta información es levantada por el programa musileng.py quien la procesa (por ejemplo, desarma los bucles escribiendo varias veces los compases que tienen dentro) y escribe en el archivo de salida el código listo para ser procesado por **midicomp**. Para escribir este código se basa de otra herramienta que llamamos traductor.py que provee métodos para garantizar esta escritura de manera correcta tomando como entrada los parametros manipulados por musileng.py

# 4. Conclusiones

A modo de conclusión del trabajo hay dos puntos que nos resultan importante mencionar. Por un lado, creemos que el trabajo, si bien tiene una componente

algorítmica o general de programación que es independiente al contenido de la materia, es un buen exponente de casi todos los temas que se han visto a lo largo de la cursada. Más aún, es una instanciación completamente práctica de una materia con una fuerte carga teórica y creemos que este balance es positivo porque nos permite atacar los temas desde dos frentes muy importantes.

Por otro lado, nos pusimos a pensar que hubiéramos hecho si este trabajo hubiera sido parte de otra asignatura previa a Teoría de Lenguajes. Creemos que la resolución del mismo no hubiera sido posible (o al menos de una forma elegante) sin tener las herramientas tanto teóricas como prácticas para alcanzar este objetivo. No sólo que estaríamos reinventando la rueda sino que además creemos que la solución alcanzada por fuera de este marco sería algo completamente poco robusta y con un alto costo de modificación o extensión.

Esto no implica que la tarea de leer un archivo, interpretarlo y generar otro de un significado parecido sea algo trivial. Más bien todo lo contrario, creemos que es una tarea compleja a pesar de los avances con los que contamos y que vimos a lo largo de la materia. Especialmente compleja cuando hablamos de tareas que se agregan en este proceso como por ejemplo la optimización del código que se genera y demás cosas que son importantes en el diseño de compiladores.

Por último, pudimos ver también la utilidad práctica que tienen los compiladores más allá de los algoritmos o de los lenguajes de programación en sí. Los compiladores permiten parsear y manipular texto con una facilidad increíble a partir de una serie de reglas gramaticales que describen el objetivo de la transformación que queremos realizar. Para un lenguaje de programación representan una interfaz entre la voluntad del programador y el lenguaje que una máquina puede entender y por lo tanto resultan esenciales e impresindibles. Pero con este trabajo, hemos observado que esta cualidad de interfaz o intermediario que cumplen los compiladores puede aplicarse a multiples aspectos de la vida práctica: en este caso, pasar de un lenguaje sensillo y entendible por cualquier persona a un lenguaje más complejo que puede ejecutar el midicomp.

# 5. ANEXO: Manual

Para correr el programa ejecutar python musileng.py archivo_de_entrada archivo_de_salida

En la carpeta .$^{en}$tradas_de_prueba.$^{a}$ su vez, hay varios archivos de entradas modelos que pueden utilizarse para probar el programa

También se adjuntan los archivos musileng_test.py y parser_rules_test.py que constituyen un conjunto de pruebas funcionales (musileng_test) y unitarias (parser_rules) que pueden correrse directamente desde python (en la carpeta .$^{en}$tradas_de_pruebaza tienen las entradas de prueba que toman los tests y en el caso de musileng_test las guarda en la carpeta "salidas"que debe estar creada para que los tests funcionen)