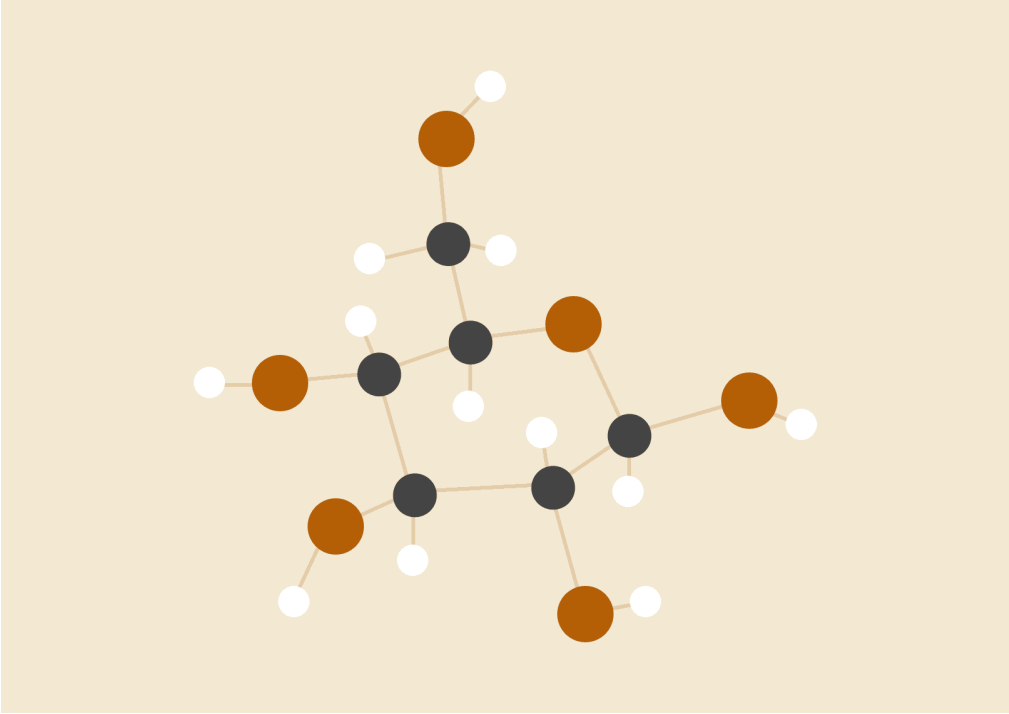
AUTÓMATAS, TEORÍA DE LENGUAJES Y COMPILADORES

TPE - Grupo 9 - 22/06/22



Gonzalo Rossin - 60135

Alberto Abancens - 62581

Mauro Daniel Sambartolomeo - 61279

Juan Negro - 61225

15/06/2022

INTRODUCCIÓN

MusiCode es un lenguaje que permite el desarrollo de melodías por medio de un lenguaje de programación. Es un lenguaje imperativo el cual el usuario puede declarar melodías y estas melodías van a englobar el conjunto de notas que van a darle identidad a cada melodía. Además, el lenguaje posee gran variedad de herramientas para modificar las melodías como modificar octavas o el BPM de la melodía. Por último, musiCode ofrece herramientas para facilitar el desarrollo en el lenguaje por medio de la utilización de ciclos while y comentarios para no tener que repetir código manualmente y la concatenación de melodías.

CONSIDERACIONES ADICIONALES

Para la implementación del lenguaje fue necesario el uso de la librería externa de Java *JFugue*, la cual utiliza el texto de nuestro lenguaje y lo traduce en audio que es el producto final de nuestro lenguaje.

Una consideración importante consiste en cómo correr nuestro compilador. Al ejecutar nuestro compilador con un programa de entrada, se generará un archivo *Output.java* dentro de un proyecto *Maven* interno al proyecto (en el caso de que el archivo de entrada sea válido según el análisis sintáctico y semántico). Luego, se deberá entrar al directorio donde está el proyecto *Maven* y correr los comandos *mvn package*, para después ejecutar el *JAR* generado. Luego, se provee un *script* de *bash* llamado *compile.sh* que recibirá como argumento el programa a compilar y el directorio donde se quiere que se copie el proyecto *Maven*. Además, la compilación y ejecución del proyecto *Maven* debe realizarse utilizando *Java 1.8*, pues se necesita para poder generar el sonido resultante.

Por otro lado, otra consideración a tener en cuenta es que, para poder compilar correctamente nuestro código, se debe estar en el directorio base del repositorio. Esto es así pues el generador de código y el *script* para compilar que ofrecemos ambos deben referenciar otros archivos del repositorio, y no podrían hacer eso con *PATH* relativo si se está en otro directorio. Aunque es posible realizar un arreglo a esta limitación, por ejemplo utilizando variables de entorno que referencian a la base del repositorio, tomamos la decisión de no implementarlo por su dificultad de manejo en el lenguaje *C* y porque no era central al funcionamiento del programa, llegando a esta conclusión al consultar a la Cátedra.

DESCRIPCIÓN DE LA GRAMÁTICA Y SINTAXIS

Para destacar de la sintaxis están los terminales que representan las notas musicales que van desde A2 hasta G6 y pueden tener un modificador “b” o “#” para ver si está en bemol o sostenido.

Para un correcto entendimiento de nuestra gramática es necesario explicar el funcionamiento de algunos no terminales.

- MusicAssignment: Nuestro lenguaje permite modificar los campos internos de variables musicales de una manera humana y directa con declaraciones del estilo de *nota TONE C4 RYTHM whole*. Music assignment permite que se aniden estas declaraciones. También participa en la declaración de las funciones para eliminar y agregar notas, así como en la concatenación de melodías.
- MusicTypeDefinititon: Es necesario este no terminal para permitir que MusicAssignment lo tome y de esta manera en una sola línea declarar una variable y setear valores de su estado interno.

DESARROLLO DEL PROYECTO Y FASES DEL COMPILADOR

Primero para el frontend se utilizaron las herramientas *flex* y *bison* para el análisis léxico y sintáctico respectivamente y a partir de la gramática generada creamos un árbol de derivación para luego usarlo en el backend.

Para la formación del árbol optamos por no utilizar nodos genéricos para facilitar el entendimiento y recorrido. Luego, utilizamos nodos que representan a los símbolos de la gramática, y utilizamos *Enums* para distinguir cómo se crearon esos nodos. Por ejemplo, una expresión se puede obtener de varias formas, sea evaluando otra expresión o considerando una constante como una expresión. Luego, se utilizan *Enums* para distinguir estos dos casos, así al descender por el árbol para el análisis semántico y la generación de código podremos tener más información de cada nodo, y así entender mejor cómo convertirlo en el lenguaje destino.

Para el análisis semántico, como se dijo anteriormente se recorre todo el árbol de forma descendiente principalmente para validar que las asignaciones y operaciones se hayan realizado en variables que las admiten y también para validar que estas variables hayan

sido declaradas previamente antes de que sean usadas. Para hacer esto se cuenta con una pila básica en la cual se apilan diferentes tablas de símbolos que representan diferentes contextos (en nuestro caso utilizamos un array para facilitar la implementación aunque podría ser más eficiente utilizando un *Hashmap).*

De esta forma, cada vez que se utiliza un *if* o un *while* se crea un contexto nuevo y se apila un nueva tabla en la pila y esta se desapila cuando se sale de ese contexto. A partir de esto cuando se quiere agregar una nueva variable siempre se la agrega en la tabla de arriba de la pila pero cuando se quiere buscar una definición previa de una variable se chequea primero en la tabla superior y después en las anteriores, de esta manera se pueden redefinir variables en contextos anidados. Gracias a estas validaciones que extienden las validaciones del frontend, cuando pasamos a la generación de código estamos seguros de que el código generado es código válido.

La generación de código en sí está inherentemente relacionada a nuestra implementación en *Java*. Se podrá ver en "Dificultades Encontradas" cómo necesitamos de dos librerías para evaluar nuestro código, la primera hecha por nosotros para representar en *Java* las notas y melodías, y la segunda para obtener el sonido resultante. Luego, para poder generar un archivo *Output.java* que pueda utilizar nuestra librería para representar las variables como clases en *Java*, se analizaba cada caso de cada nodo y se escribía en un archivo lo necesario. Donde más se observa esta relación entre nuestro código y su representación en *Java* es en el nodo que representa asignaciones musicales, donde se deben utilizar métodos y constructores de nuestras clases para obtener el resultado final.

DIFICULTADES ENCONTRADAS

Uno de los problemas fue definir correctamente la gramática. Dado que carecíamos del conocimiento acerca de cómo se desarrollaba el backend, la gramática que habíamos hecho inicialmente terminó siendo más compleja de lo que tal vez hubiera sido si hubiéramos sabido toda la teoría del proyecto de antemano. Llegamos a esta conclusión durante el desarrollo del backend.

Por otro lado, también fue dificultoso encontrar la forma de generar el sonido final después de compilar. Después de investigación sobre cómo ejecutar sonidos en distintos lenguajes de programación, encontramos la librería ***JFugue***, basada en *Java*, que permitía reproducir variables de tipo *String* a código, si es que los caracteres seguían ciertas reglas. Luego, se tomó la decisión de lograr que nuestro código se traduzca a *Java*,

y de esta manera poder generar el *String* dinámicamente y reproducirlo. Luego, se utilizó la librería de *JFugue* junto a la documentación para aprender a reproducir sonidos, y así poder entender el formato del *String* necesario para generar los sonidos deseados. Además, generamos nuestro propio código en *Java* que establecía las melodías y notas como distintas clases para poder ir analizando el código nuestro y convertirlo en un código *Java*.

El mayor de los problemas encontrados en la traducción fue la variable de tipo *Melody*. Por cómo se había diseñado nuestra gramática, se debe poder concatenar dos melodías distintas de forma natural, y el *BPM* de la segunda melodía debe ser respetado. Es decir, concatenar dos melodías debe resultar en otra melodía que cambie su *BPM* después de cierta nota. Además, también se debe poder insertar una melodía dentro de otra, y se desea que la melodía interna mantenga su *BPM*. Luego, se utilizó la clase *MelodyWrapper*, que contiene una lista de melodías, que en sí mantienen cada una su *BPM*. De esta manera, concatenar melodías consiste en agregar elementos a la lista, e insertar melodías dentro de otras consiste en dividir la melodía externa en dos y concatenar la melodía interna en donde se desee. Luego, podemos lograr todo lo intencionado de forma clara y sencilla.

Finalmente, el *String* requerido se obtiene en ejecutar el método *toString* de una *MelodyWrapper*, que concatena el método *toString* de cada *Melody*, que agrega su *BPM* y concatena el método *toString* de cada *Note*, que en sí se resuelve según lo indicado en la documentación de *JFugue*. Con las clases programadas por nosotros, utilizar nuestro lenguaje se convierte en algo más sencillo, y también se deja mucho lugar para extensiones aprovechando todo el funcionamiento de *JavaX* y *JFugue*.

FUTURAS EXTENSIONES

Hay muchas formas que se podría extender nuestro código para tener más funcionalidades. Aquí presentamos algunas de ellas:

1. Poder declarar acordes, que serían conjuntos de notas que suenan al mismo tiempo.
2. Poder lograr una melodía compuesta por otras dos melodías que suenan independientemente, por lo tanto podría suceder que ciertas notas suenen al unísono.
3. Poder agregar instrumentos que interpretan las notas con su respectivo timbre.
4. Exportar la melodía hacia un archivo en forma de partitura.

CONCLUSIÓN

Como conclusión podemos decir que si bien la nueva modalidad nos permitio desarrollar el proyecto a conciencia y sin apuro por llegar a una fecha de entrega, no tener los conocimientos nos dio varios inconvenientes ya que no estábamos seguros de cómo comenzar a implementar elementos como el árbol de sintaxis o la tabla de símbolos. En resumen, la nueva modalidad tuvo sus pros y contras pero la clase de consulta fue de mucha ayuda para abordar estos problemas y poder realizar un proyecto completo y bien desarrollado.

REFERENCIAS

1. JFugue: La librería que se utilizó para convertir las notas dadas por nuestro lenguaje a audio