Trabajo Práctico Especial

Segundo Cuatrimestre de 2022

Asignatura: 72.39 - Autómatas, Teoría de Lenguajes y Compiladores

Grupo 9 - Hello Melody!

Autores:

Banfi, Malena	**(61008)**
Fleischer, Lucas	**(61153)**
Perez Rivera, Mateo	**(61170)**
Szejer, Ian	**(61171)**

Índice

Introducción con idea subyacente y objetivos del lenguaje

El objetivo de este trabajo fue desarrollar un lenguaje de programación y su compilador en su totalidad. Dicho esto, se construyó la gramática del lenguaje y el analizador léxico y sintáctico del compilador.

Hello Melody! es un lenguaje que permite desarrollar sonidos por medio de un lenguaje de programación. Es un lenguaje programático que permite generar sonidos musicales, simulando ser un piano o una flauta. El lenguaje permite utilizar todo el rango de notas y algunas variaciones para no limitar la capacidad artística del programador. Es un lenguaje muy amplio, con muchas herramientas para modificar la melodía, como por ejemplo, variar el tempo de una canción, la duración en segundos de la misma, y las octavas de notas entre otras. Al compilar el programa se genera un archivo ejecutable con el sonido esperado.

Las tecnologías utilizadas para realizar el trabajo fueron Flex, Bison y Java (en especial la librería JFugue).

Consideraciones

Para la implementación del lenguaje, fue necesario la integración de la librería JFugue, la cual transforma el texto que le pasamos (el formato de la canción) y lo traduce en audio, que es la salida de nuestro lenguaje.

"*Debido a que este ejemplo se desarrolla en Java, se deja en manos del cliente, la responsabilidad de disponer una instalación del runtime, es decir, de una JVM (Java Virtual Machine), con una versión que soporte el programa generado.*"[1]

Una vez armado el programa por el usuario, si el mismo es aceptado, se generará un archivo ejecutable (Out.java) el cual contendrá todo el código necesario para que, con solo ejecutar dicho archivo, el usuario pueda escuchar la canción que él mismo generó. Dicho archivo se encuentra en: *"java/src/main/java/com/example/java/Out.java"*.

Se proveen tres archivos de prueba de sonidos conocidos, como el "felizCumpleanios", "suspenso" y "lordOfTheRings?".

Para mas información, leer el README.md

Desarrollo

Para desarrollar el proyecto, lo primero que necesitábamos era una idea. Gracias a las ideas propuestas por la cátedra, nos orientamos por el lado de los sonidos.

La primera idea fue generar una salida auditiva de resultados de expresiones matemáticas resueltas, pero luego, al descubrir la librería "JFugue" de Java, orientamos la idea del proyecto a sonidos musicales.

[1] (2022-11-13, v0.1.0) Backend - M. Agustin Golmar

Una vez finalizada la etapa de investigación previa y ya con la idea definida, empezamos a trabajar en la parte del Frontend del proyecto. Continuamos con la etapa del desarrollo de la gramática, en donde todo el grupo colaboró, ya que creímos que era una gran base para que el resto del proyecto sea claro y más sencillo. Para esta etapa, tuvimos que utilizar tanto Flex como Bison (para el análisis sintáctico y léxico). Para poder comenzar, tuvimos que buscar información y leer sobre el funcionamiento de los mismos (gran mayoría provista por la cátedra) pero gracias al proyecto base que se nos dio, se nos hizo más fácil la conexión entre los archivos de cada uno y cómo se relacionaban entre sí.

Luego, vino la etapa de generación de sonidos literales, en donde se construyo el Backend. En dicha sección, lo primero que realizamos fue la implementación de los nodos del árbol, los cuales fueron generados en el abstract-syntax-tree.h, siguiendo mucho la lógica de nuestra gramática y pensando en sus usos. Luego, se generó una tabla de símbolos sencilla, en la cual guardamos el tipo de variable y su nombre para luego realizar chequeos de las mismas. Esta se ordenó con una lista (la idea de un mapa fue considerada, pero al ser una tabla tan simple, no fue implementada). Dichos chequeos, fueron utilizados en funciones como la “adición de notas” o “eliminación de notas” ya que, no se pueden agregar ni eliminar notas de cualquier variable, por lo tanto era necesario verificar que este bien el par de variables, y esto fue posible gracias a la tabla (si se quiere agregar una nota a otra nota, tirara error, ya que las notas pueden ser agregadas únicamente a una track).

Luego se continuó con el cambio de las funciones del flex-actions.c y bison-actions.c, ya que al cambiar los parámetros de entrada de las funciones (los cuales para la primer entrega se simplificaron como ints) hubo que hacer cambios a la hora de alocar, liberar y asignar memoria para los mismos. Una vez finalizadas, se integraron funciones de la tabla de símbolos en varias funciones del bison-actions.c, más que nada para las verificaciones de pertenencia de variables en la lista, y para la adición de símbolos en la tabla (con sus respectivos nombres).

Continuamos con el generator.c, en el cual tuvimos que copiar las salidas de nuestro código en c, al código en java, para que una vez finalizada la lectura del archivo que se quiere copiar, poder ejecutarlo en Java con funciones de la librería JFugue.

Una vez finalizada la etapa previa, comenzó la etapa final de testeos. Al finalizar las pruebas parciales de sonidos, decidimos retomar los casos de prueba (tanto los 10 aceptados como los 5 rechazados) propuestos en la primera entrega y reformarlos. Algunos de ellos, como los de duración de sonidos, aumento o disminución de velocidad, no habían podido ser testeados correctamente ya que aún no teníamos las herramientas necesarias para hacerlo. Luego de algunos ajustes, se proponen los 15 casos de uso mostrando su testeo correctamente (se agregó un caso 11 de aceptación, en el cual se como dos notas se tocan a la misma vez) .

Pequeño manual de uso de JFugue

Como se mencionó anteriormente en la introducción, hay varias herramientas para modificar el sonido musical producido. Para que no haya dudas al respecto de como modificarlo desde Java, se provee una pequeña guía de uso. Cabe destacar que antes de cada

función irá un "**outputUtils.**FuncionAUsar" seguido del nombre de función que se quiera usar. El uso de comillas en los parámetros de funciones es literal.

- Para la creación de notas, tracks y songs se utilizan las siguientes funciones. Estas tres variables son strings.
  - Para una nota, simplemente se declara la variable. No hay necesidad de inicializarla. (Ej: String noteUno).
  - Para una track se utiliza la función "defineTrack()".
  - Para una song se utiliza la función "defineSong()".

- Para modificar una nota, es decir, asignar una valor en específico (do, re, mi, …), un ritmo y un acorde se utiliza la función:
  - mofifyNote("valoDeNota", "valorDelRitmo", "valorDelAcorde"); Si no se quiere especificar un acorde, no se pasa dicho parámetro. Asimismo, si no se quiere especificar ni ritmo ni acorde, no se pasan dichos parámetros.
    Los valores aceptados de las notas son: "C(do), D(re), E(mi), F(fa), G(sol), A(la), B(si), R(sleep)".
    Los valores aceptados para los ritmo son: "q(q), qqq(i), w(w), h(h)".
    Los valores aceptados para los acordes son: "CNumero" donde el número puede ir de 0 a 10, a medida que se incrementa el número se incrementa la agudez. C1 es muy grave y C10 muy agudo.
- Para agregar una nota a una track, o una track a una canción, se utilizan las siguientes funciones:
  - Para agregar una nota a una track "trackAddNote(nombreTrack, nombreNota).
  - Para agregar una track a una canción "songAddTrack(nombreCancion, nombreTrack);
- Para quitar la última aparición de una nota y/o quitar todas las apariciones de una nota de una track se usan las siguientes funciones:
  - Para quitar la última aparición se usa "trackSubstraction(nombreTrack, nombreNota);
  - Para quitar todas "tracktrackDivision(nombreTrack, nombreNota);
- Para combinar el sonido de dos notas se utiliza la función:
  - "combine(nombreNota1, nombreNota2); ambas se reproducen simultáneamente en el sonido.
- Para aumentar y disminuir la velocidad de una track se utiliza:
  - "trackTempo(nombreTrack, double); donde el valor numérico indica la velocidad. Si el mismo es menor a 1 la velocidad disminuye, si es mayor aumenta.
- Para repetir una track x veces se usa:
  - "trackMultiply(nombreTrack, veces a repetir);
- Para cambiar de instrumento (por el momento se trabaja con piano y flauta):
  - "changeInstrument("nombreTrack", int); donde int representa el número de instrumento este número puede ser cero o uno, el cero representa el piano y el uno la flauta.
- Para finalizar, si se desea cambiar la duración total de la canción se puede usar:

➢ “setSongDuration(“nombreCancion”, int); donde el int representa el tiempo en segundos.

Gramática

Para entender correctamente nuestra gramática, pasaremos a explicar dos no terminales utilizados.

- instruction: se definen dos posibles tipos de instrucciones, las unarias tanto como las binarias.
- unaryExpression: se usa para modificar solamente a una variable.
- binaryExpression: se usa para modificar 2 variables, o más.

Para más información ver aquí

Dificultades Encontradas

A la hora de trabajar con el token “note” de nuestra gramática, encontramos dificultades ya que utilizamos una variable “note_values” la cual contiene las notas posibles que se le pueden asignar a una nota (do, re, mi, etc…). Esta confusión de nombres y tokens nos llevó a dudar de qué parámetros y cómo debíamos pasar a varias funciones que manejan estas variables. Todo se soluciono gracias a la respuesta rápida de la cátedra.

Otra dificultad que tuvimos fue el tema de la alocación de memoria. En un principio encaramos por el lado de utilizar mallocs (lo que nos trajo muchos problemas y segmentations fault) pero luego decidimos utilizar calloc (por recomendación de la cátedra), ya que nos garantiza que no hay accesos a buffers no asignados correctamente, strings que no garantizan tener un 0 al final, o accesos a memoria liberados erróneamente.

Otro problema relacionado con la memoria del programa, fue la dificultad a la hora de trabajar con punteros y casteos de variables. Tuvimos varios warnings y errores por descuidos a la hora de pasar tipos de variables mezclados e inconsistentes. Por suerte, todos pudieron ser solucionados, salvo 5 warnings que fueron dejados ya que el tipo de variable (aunque está mal casteada) es necesaria para el out.java tal y como esta.

Futuras Extensiones

A la hora de pensar en qué extensiones puede tener nuestro código, nos dimos cuentas de que hay un gran abanico de opciones pero creo que las más características son las siguientes:

1. Poder imprimir la melodía en formato de partitura
2. Poder agregar más instrumentos
3. Agregarle voz que acompañe a la melodía
4. Correr dos canciones en simultáneo
5. Poder concatenar definiciones e instrucciones en una misma sentencia

Conclusión

Para concluir, gracias a la implementación de este trabajo práctico, pudimos terminar de plasmar los conocimientos adquiridos tanto en las clases prácticas como en las teóricas. Se construyó por completo un lenguaje y su compilador, generando un lenguaje propio, experimentando de punta a punta el proceso de diseño y el desarrollo del mismo.

Más allá de las dificultades y errores encontrados en el camino, creemos que gracias a nuestro trabajo práctico nos llevamos una gran experiencia y enseñanza del mismo.

Cabe aclarar que decidimos que dos integrante del grupo solamente haga los commits al repositorio (ya que dichos integrantes tienen como sistema operativo Linux y facilitaba a la hora de ejecutar el trabajo), y trabajar a través de la extensión de Visual Studio Code llamada *LiveShare* para poder trabajar de forma colaborativa y tener una buena división de tareas.

Bibliografía y Referencias

1. Golmar, M. Agustin, Flex-Bison-Compiler
2. JFugue: Music Programming for Java™ and JVM Languages