MARIMBA

Diseño e Implementación de un Lenguaje

v1.0.0

1. Equipo

Nombre	Apellido	Legajo	E-mail
Julián Andrés	Bogado	63.338	jbogado@itba.edu.ar
Catalina	Müller	63.199	camuller@itba.edu.ar
Mateo	Pérez de Gracia	63.401	mperezdegracia@itba.edu.ar
Manuel	Quesada	63.580	mquesada@itba.edu.ar

2. Repositorio

La solución y su documentación serán versionadas en: **Trabajo Práctico**.

3. Introducción

En este informe se detalla el proceso de desarrollo del Proyecto Especial de la materia Autómatas, Teoría de Lenguajes y Compiladores. Inicialmente se presenta el modelo computacional, que incluye una descripción del alcance del lenguaje, así como el dominio. Luego se pueden encontrar detalles del funcionamiento del código en la sección de implementación, junto con las dificultades encontradas. Finalmente se presenta una conclusión del trabajo, con posibles futuras modificaciones.

4. Modelo Computacional

4.1. Dominio

Desarrollar un lenguaje que permita describir piezas musicales y pueda producir salidas en formato MIDI y/o PDF para su mejor comprensión. Este lenguaje tiene como fin ofrecer una propuesta más simple que las encontradas en el mercado, puesto que las mismas no están dirigidas a un público amplio, sino a un reducido nicho de músicos con conocimientos en lenguajes de programación.

4.2. Lenguaje

El lenguaje desarrollado ofrece las siguientes construcciones, prestaciones y funcionalidades:

(I). Creación de notas.
(II). Creación de patrones rítmicos.
(III). Creación de diversos sonidos que emulan instrumentos musicales.
(IV). Las variables podrán ser de tipo int, score, tempo, signature, clef, tabs y varios tipos de instrumentos.
(V). El lenguaje soportará las clases Integer, Note, Chord y Tab.
(VI). El lenguaje posee ciertas palabras reservadas que son instancias de las clases Note y Chord, las cuales son descritas por las siguientes expresiones regulares:
- `(c|d|e|f|g|a|b)(#|b)?([1-9]|10)?(\((w|h|q|i|s|t|x|o)\))?`
- `^(C|D|E|F|G|A|B)(m|7|maj7|dim|aug|sus2|sus4)?(\(([1-9]|10),(w|h|q|i|s|t|x|o)\))?(arp)?$`
- `R(\((w|h|q|i|s|t|x|o)\))?`

(VII). Para las notas, denotadas en minúscula, el # o b representan la alteración correspondiente, y entre paréntesis se define la duración de esa nota.
(VIII). Para los acordes, denotados en mayúscula, la información directamente concatenada define el tipo de acorde, y la información entre paréntesis el tono de la nota (la octava) y la duración, y *arp* si se desea reproducir como arpeggio.

(IX). Para los descansos, se utiliza con la letra R, y entre paréntesis se define la duración del mismo.

(X). El lenguaje soportará la estructura de flujo *repeat* para repetir cualquier conjunto de tabs.

(XI). El lenguaje también soportará estructuras de flujo *before* y *after*, las cuales reproducen un conjunto de tabs antes o después de otras, respectivamente.

(XII). El lenguaje soportará el método *transpose* para el *score*, el cual transpone las notas la cantidad de semitonos deseados.

Duration	Character	Decimal value
Whole	W	1.0
Half	H	0.5
Quarter	Q	0.25
Eighth	I	0.125
Sixteenth	S	0.0625
Thirty-second	T	0.03125
Sixty-fourth	X	0.015625
One-twenty-eighth	O	0.0078125

5. Implementación

5.1. Frontend

En esta sección del proyecto el programa recibe input textual del usuario, obtenido desde un archivo con formato .msh. La librería Flex-Bison se encarga de tokenizar este input con el formato que corresponde al lenguaje, es decir, se separan los símbolos y a cada uno se le asigna un valor semántico que será utilizado para completar las producciones de la gramática. A su vez, las producciones generan composiciones entre sí para lograr representar construcciones más complejas que permiten casos de uso diversos del lenguaje. Cabe resaltar que estas composiciones no producen errores de Shift-Reduce ni Reduce-Reduce[1], por lo que la gramática

[1] A. V. Aho, M.S. Lam, R. Sethi y J.D. Ullman. Compilers: Principles, Techniques, and Tools (2ª ed). USA. Addison – Wesley. 2006.

no es ambigua. Cada resultado de las producciones es almacenado en un nodo del AST[2] (Abstract Syntax Tree), hasta llegar a un resultado que sea un símbolo terminal, el cual se almacena en una hoja del árbol.

5.2. Backend

En este apartado, el programa toma como input el AST y se encarga de ingresar los nodos que corresponden en la tabla de símbolos, la cual almacena la cadena de caracteres utilizada para identificar una variable, junto con el tipo de dato que le corresponde. Esta información es utilizada para realizar chequeo de tipos donde corresponde, como en el caso de una asignación, donde asignar, por ejemplo, Chord a una variable declarada como Note debe arrojar un error.

Luego de corroborar que la sentencia es válida, se genera el output que recibirá la librería JFugue[3], es decir, se genera un string con código Java que Jfugue utilizará para devolver una salida en el formato especificado, ya sea midi o pdf.

5.3. Dificultades Encontradas

Durante el desarrollo del proyecto nos encontramos con numerosas dificultades, a continuación mencionamos las que consideramos más importantes.

5.3.1. Definir sintaxis del lenguaje

Inicialmente se tornó dificultoso definir una sintaxis para el lenguaje ya que no conocíamos las restricciones de la librería Flex-Bison. A su vez no conocíamos el comportamiento tanto del frontend como del backend, por ende algunas funcionalidades debieron ser adaptadas, por ejemplo, eliminamos la funcionalidad **`.along()`**, ya que no era compatible con la escritura de partituras.

Luego, en la sección de backend tuvimos que lidiar con decisiones que se tomaron en el frontend sin conocer el comportamiento del back, lo que también generó algunos conflictos, como el hecho de cambiar en el front el orden en que se recibían los id's, para poder realizar el chequeo de tipos en el back.

También fue un desafío adaptar las funcionalidades para hacerlas accesibles a usuarios sin conocimiento de programación, ya que frecuentemente la primer implementación de alguna funcionalidad resultaba sesgada debido a nuestra afinidad con este entorno, y era necesario cambiarla por versiones más simples.

5.3.2. Producciones

Definir las producciones también fue un desafío ya que la sintaxis del lenguaje permitía una gran cantidad de variaciones posibles en cuanto a combinaciones de tipos de datos y sentencias. Además, mientras menos restricciones se le imponen al usuario, es

[2] A. V. Aho, M.S. Lam, R. Sethi y J.D. Ullman. Compilers: Principles, Techniques, and Tools (2ª ed). USA. Addison – Wesley. 2006.

[3] Librería de Java utilizada en el proyecto.

necesario considerar más producciones, con todas las funciones y código adicional que eso involucra.

5.3.3. Pipeline de ejecución

A su vez una dificultad que se presentó, que si bien es inherente al trabajo práctico, queremos mencionar en este apartado ya que estuvo muy presente a lo largo de todo el desarrollo. Descifrar el orden de ejecución de los distintos módulos fué un gran desafío hasta el último momento, ya que era necesario considerar qué input le llegaba a cada módulo, y cuál era su output, para tenerlo en cuenta para el siguiente.

No aportó mucho el hecho de que Flex-Bison es en sí mismo bastante críptico, sin mencionar los bloques de código autogenerados, que en muchos casos son cajas negras que se asumen funcionales.

6. Futuras Extensiones

En futuras iteraciones de este trabajo se podrían agregar las siguientes funcionalidades:

- Pipes para validar duraciones por compás.
- Preview del resultado a medida que se escribe el código.
- Brindar más formatos de output, como mp3, por ejemplo.

7. Tutorial de uso

Para lograr parsear un archivo .msh primero se debe contar con un entorno de compilación apto. Para esto es necesario clonar el repositorio de git de Marimba#. Una vez se cuenta con el proyecto, es necesario correr el siguiente comando (se asume un entorno de ejecución de linux):

```
<project_root>/script/ubuntu/build.sh
```

Se debe reemplazar <project_root> con el path hasta el directorio root del proyecto.

Dicho comando realiza la compilación de los archivos para poder ejecutarlos.

Posteriormente se debe correr el siguiente comando (ubicado en la root del proyecto):

```
./script/ubuntu/generate.sh <path_to_file>
```

Donde <path_to_file> es el path al archivo .msh que queremos parsear.

Este comando genera un archivo con el código Java equivalente al input recibido en lenguaje Marimba#. También se genera el archivo pom.xml que se corresponde con las dependencias de Maven necesarias para correr el proyecto en un entorno Maven de Java. El resultado son dos archivos, un .pdf y un .midi que representan el input ingresado por el usuario ya procesado, los cuales se encuentran en el directorio `./marimbash`.

8. Conclusiones

Para finalizar es pertinente realizar una pequeña recapitulación del trabajo práctico: Inicialmente se realizó una entrega con el diseño del lenguaje, en la segunda entrega se realizó el frontend, que involucra recibir input del usuario y generar un AST, finalmente en la tercer y última entrega se parsea el AST y se genera la salida en código Java que utiliza las dependencias de Jfugue. A lo largo del desarrollo se aplicaron conceptos teóricos vistos en clase como gramáticas, producciones, análisis léxico, etc.

Aunque fué complicado entender el funcionamiento de la librería Flex-Bison, es innegable la potencia que proporciona a la hora de realizar análisis de un input, transformarlo y brindar un output, permitiendo generar salidas en archivos de cualquier tipo, gracias al uso del lenguaje C. Y, si bien por momentos la percepción era la de estar programando algo abstracto y poco palpable, una vez obtenido el output todo quedó mucho más claro y fué evidente el por qué del uso del entorno propuesto por la cátedra.

9. Bibliografía

https://www.cs.utexas.edu/ftp/novak/cs315/jfugue-chapter2.pdf

https://github.com/agustin-golmar/Flex-Bison-Compiler

http://www.jfugue.org/

A. V. Aho, M.S. Lam, R. Sethi y J.D. Ullman. Compilers: Principles, Techniques, and Tools (2ª ed). USA. Addison – Wesley. 2006.