



Escuela
Politécnica
Superior

Diseño de gramática para la codificación semántica de partituras musicales basada en Humdrum



Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:

Jorge Marco Esteve

Tutor:

David Rizo Valero

Junio 2020



Universitat d'Alacant
Universidad de Alicante

Diseño de gramática para la codificación semántica de partituras musicales basada en Humdrum

Autor

Jorge Marco Esteve

Tutor

David Rizo Valero

Departamento de Lenguajes y Sistemas Informáticos



Grado en Ingeniería Informática



Escuela
Politécnica
Superior



Universitat d'Alacant
Universidad de Alicante

ALICANTE, Junio 2020

Agradecimientos

Este trabajo no hubiera sido posible sin la paciencia de mi tutor Daviz Rizo Valero que se prestó a ser mi tutor y bajo su supervisión he logrado concluir un trabajo que en ocasiones lo veía inacabable. También quiero agradecerse a todos mis compañeros, aquellos que me han ayudado y aquellos que me han puesto el camino más difícil, los cuales han hecho que me motive más. A mis amigos, tanto los que tengo cerca como los que tengo lejos, los que cuando me han visto algo deprimido me han sacado una sonrisa a cambio de nada. Todos ellos han sido mi motivación y desahogo. Por último, no puedo terminar sin poder agradecerse a mi familia, que me ha apoyado en todo este trayecto de manera incansable, dándome toda la fuerza del mundo para seguir y acabar esta etapa.

Es a ellos a quien dedico este trabajo.

*A mi familia: José, Macu, Jose, Mario y Chuck.
sin los cuales no habría acabado todo esto.*

Son pocas ojeras para tantos sueños.

Escandar Algeet.

Índice general

1. Introducción	1
1.1. Objetivos y Requisitos	1
1.1.1. Objetivos	1
1.1.2. Requisitos	1
1.2. Contexto	1
1.2.1. Historia	1
1.2.2. Ensamblador	2
1.3. Compiladores e intérpretes	2
1.3.1. Compiladores	2
1.3.2. Fases de la compilación	3
1.3.2.1. Lenguaje de programación	3
1.3.2.2. Proceso de compilación	3
1.3.3. Intérpretes	11
1.3.3.1. Estructura de un intérprete	12
1.3.3.2. Ventajas de los intérpretes.	13
1.4. Metaprogramas	13
1.4.1. Flex [1] y Bison [2]	13
1.4.2. ANTLR	13
2. Gramática para Kern [4] y Mens [5]	15
2.1. Introducción	15
2.2. Representación musical	15
2.3. Kern y Mens	15
2.4. Decisiones sobre la gramática	17
2.4.1. Analizador sintáctico descendente o ascendente	17
2.5. Notación de gramática.	19
2.5.1. Diagramas de sintaxis	19
2.6. Implementación de la gramática.	20
2.6.1. Metodología.	20
2.7. Formato de notación **kern y **mens	21
2.7.1. Alcance de nuestra gramática.	21
2.8. Especificación de la gramática	21
2.8.1. Léxico	21
2.8.2. Sintáctico	24
2.9. Explicación de nuestra gramática	28
2.9.1. **kern	28
2.9.1.1. Clave	29
2.9.1.2. Armadura de compás	30
2.9.1.3. Tiempo de compás	30

2.9.1.4.	Contenido musical	31
2.9.1.5.	Figuras	31
2.9.1.6.	Ejemplo de **kern	37
2.9.2.	**mens	40
2.9.2.1.	Ritmo de partitura	40
2.9.2.2.	Contenido musical	41
2.9.2.3.	Figuras	42
2.9.2.4.	Ejemplo de **mens	45
3.	Pruebas Unitarias	49
3.1.	Pruebas de caja blanca	49
3.1.1.	Creación de pruebas	49
3.1.1.1.	Implementación de pruebas	49
3.1.1.2.	Ejemplo de prueba	52
4.	Conclusiones	53
4.1.	Posibles proyectos futuros	53
4.2.	Conclusiones	53
4.2.1.	Código fuente del trabajo.	54
	Bibliografía	55
A.	Anexo I	57
A.1.	Introducción	57
A.2.	**kern	57
A.2.1.	Alteraciones	57
A.2.2.	Articulaciones	58
A.2.3.	Punto de aumentación	60
A.2.4.	Barra de compás	60
A.2.5.	Corcheas	61
A.2.6.	Estructura de tiempo de compás	63
A.2.7.	Armadura	64
A.2.8.	Notas	65
A.2.9.	Ornamentos	65
A.2.10.	Silencios	67
A.2.11.	Ritmo	68
A.2.12.	Ligado	69
A.2.13.	Dirección de la plica	70
A.2.14.	Ligadura	70
A.2.15.	Octavas	71
A.2.16.	Acordes	72
A.3.	**mens	73
A.3.1.	Barras de compás internas	73
A.3.2.	Puntillo	73
A.3.3.	Ligadura de expresión	73
A.3.4.	Ligature	74

A.3.5. Estructura de ritmo	74
A.3.5.1. Común	74
A.3.5.2. Perfecta	77
A.3.6. Notas	79
A.3.7. Alteración	80
A.3.8. Silencios	80
A.3.9. Tiempo de la nota	81

Índice de figuras

1.1. Proceso de ensamblador.	2
1.2. Pasos de Lenguaje de alto nivel a código máquina.	3
1.3. Estructura básica de un compilador	4
1.4. Ejemplo de diagrama de transición de un compilador donde se registran los números enteros y los reales	5
1.5. Árbol generado a partir del ejemplo para la gramática anterior.	7
1.6. Las gramáticas $LL(1)$ como subconjunto de las gramáticas libres de contexto y no ambiguas.	10
1.7. Estructura de un intérprete.	12
2.1. Representación de **kern	16
2.2. Representación de **kern	17
2.3. Comparación entre ANTLR y flex/bison	18
2.4. Ejemplo de regla de gramática.	19
2.5. Ejemplo de diagrama de sintaxis	20
2.6. Regla inicial	28
2.7. Regla que deriva a la notación **kern	28
2.8. Representación de regla **kern	29
2.9. Representación de la clave del compás.	29
2.10. Representación de las diferentes claves que podemos asignar.	29
2.11. Representación de armadura de compás.	30
2.12. Representación de armadura de compás.	30
2.13. Representación de tiempo de compás.	30
2.14. Representación de la fracción del tiempo de compas.	30
2.15. Representación de la estructura rítmica.	31
2.16. Estilos de estructura rítmica.	31
2.17. Representación de contenido musical.	31
2.18. Representación de las figuras.	32
2.19. Representación del diferente formato de notas.	32
2.20. Representación de una nota.	32
2.21. Representación visual del tiempo de la nota.	33
2.22. Representación de una nota.	33
2.23. Representación de las alteraciones.	33
2.24. Representación de las articulaciones y los ornamentos.	34
2.25. Cambio de dirección de la plica.	34
2.26. Representación de un acorde.	34
2.27. Representación del acorde de sol.	35
2.28. Representación de agrupación de notas.	35
2.29. Representación de una corchea.	35

2.30. Representación del silencio.	36
2.31. Representación de un silencio de corchea.	36
2.32. Representación de un ligadura de expresión.	36
2.33. Representación de una ligadura de expresión básica.	36
2.34. Representación de una ligadura.	37
2.35. Representación de una ligadura.	37
2.36. Representación musical del código ejemplo **kern	38
2.37. Árbol sintáctico generado.	39
2.38. Regla principal de **mens	40
2.39. Regla de tiempo de compás.	40
2.40. Estructura de compás.	40
2.41. Representación de estructuras de compás perfecta.	41
2.42. Representación de estructuras de compás común.	41
2.43. Representación de un signo mensural.	41
2.44. Contenido musical	42
2.45. Estructura de los separadores.	42
2.46. Estructura de figuras **mens	42
2.47. Representación de notas **mens	43
2.48. Representación de una nota.	43
2.49. Representación de la alteración y del puntillo en mensural.	43
2.50. Representación de una nota con alteraciones y puntillo.	43
2.51. Representación de silencios en **mens	44
2.52. Representación de un silencio de fusa.	44
2.53. Representación del ligadura de expresión en **mens	44
2.54. Representación de la ligadura en **mens	45
2.55. Representación de una ligature.	45
2.56. Representación musical del código ejemplo **mens	46
2.57. Árbol sintáctico generado de **mens	47
3.1. Ejemplo de prueba realizada.	52
3.2. Mensaje error al pasar una prueba	52
A.1. Prueba 1 de alteraciones.	57
A.2. Prueba 2 de alteraciones.	57
A.3. Prueba 1 de articulaciones.	58
A.4. Prueba 2 de alteraciones.	58
A.5. Prueba 3 de articulaciones.	58
A.6. Prueba 4 de articulaciones.	58
A.7. Prueba 5 de articulaciones.	59
A.8. Prueba 6 de articulaciones.	59
A.9. Prueba 7 de articulaciones.	59
A.10. Prueba 8 de articulaciones.	59
A.11. Prueba de punto de aumentación.	60
A.12. Prueba 1 de barra de compás.	60
A.13. Prueba 2 de barra de compás.	61
A.14. Prueba 1 de corcheas.	61

A.15.Prueba 2 de corcheas.	62
A.16.Prueba 3 de corcheas.	62
A.17.Prueba de corcheas parciales.	62
A.18.Prueba 1 de estructura de compás.	63
A.19.Prueba 2 de estructura de compás.	63
A.20.Prueba 1 de la armadura de compás.	64
A.21.Prueba 2 de la armadura de compás.	64
A.22.Prueba 3 de la armadura de compás.	64
A.23.Prueba 1 de notas.	65
A.24.Prueba 1 de notas.	65
A.25.Prueba 1 de ornamentos.	65
A.26.Prueba 2 de barra de ornamentos.	66
A.27.Prueba 3 de ornamentos.	66
A.28.Prueba 4 de barra de ornamentos.	66
A.29.Prueba 5 de barra de ornamentos.	66
A.30.Prueba 6 de ornamentos.	67
A.31.Prueba 7 de ornamentos.	67
A.32.Prueba de silencio.	67
A.33.Prueba 1 de tiempo de la nota.	68
A.34.Prueba 2 de tiempo de la nota.	68
A.35.Prueba 1 del ligado.	69
A.36.Prueba 2 del ligado.	69
A.37.Prueba 1 de la dirección de la plica.	70
A.38.Prueba 2 de la dirección de la plica.	70
A.39.Prueba 1 de ligadura.	70
A.40.Prueba 2 de ligadura.	71
A.41.Prueba 3 de ligadura.	71
A.42.Prueba 1 de octavas.	71
A.43.Prueba de cambios de configuración.	72
A.44.Prueba de acordes.	72
A.45.Prueba de barra de compás internas.	73
A.46.Prueba de puntillo.	73
A.47.Prueba de ligado de expresión.	73
A.48.Prueba de ligature.	74
A.49.Prueba 1 de estructura de ritmo común.	74
A.50.Prueba 2 de estructura de ritmo común.	74
A.51.Prueba 3 de estructura de ritmo común.	75
A.52.Prueba 4 de estructura de ritmo común.	75
A.53.Prueba 5 de estructura de ritmo común.	75
A.54.Prueba 6 de estructura de ritmo común.	75
A.55.Prueba 7 de estructura de ritmo común.	76
A.56.Prueba 8 de estructura de ritmo común.	76
A.57.Prueba 9 de estructura de ritmo común.	76
A.58.Prueba 10 de estructura de ritmo común.	76
A.59.Prueba 11 de estructura de ritmo común.	77

A.60.Prueba 1 de estructura de ritmo perfecto.	77
A.61.Prueba 2 de estructura de ritmo perfecto.	77
A.62.Prueba 3 de estructura de ritmo perfecto.	77
A.63.Prueba 4 de estructura de ritmo perfecto.	78
A.64.Prueba 5 de estructura de ritmo perfecto.	78
A.65.Prueba 6 de estructura de ritmo perfecto.	78
A.66.Prueba 6 de estructura de ritmo perfecto.	78
A.67.Prueba 8 de estructura de ritmo perfecto.	79
A.68.Prueba 9 de estructura de ritmo perfecta.	79
A.69.Prueba 10 de estructura de ritmo perfecta.	79
A.70.Prueba de notas.	79
A.71.Prueba de alteración de la nota.	80
A.72.Prueba de silencio.	80
A.73.Prueba de tiempo de las notas.	81

Índice de tablas

1.1. Gramática de una declaración de variables tipo entero.	6
1.2. Representación de análisis descendente	8
1.3. Ejemplo de gramática ambigua	9
1.4. Representación de análisis ascendente.	10
2.1. Léxico de la gramática.	21
2.2. Reglas de la gramática.	24
2.3. Código de entrada **mens	45

1. Introducción

1.1. Objetivos y Requisitos

1.1.1. Objetivos

Este trabajo pretende mostrar la realización del diseño de una gramática formal paso a paso para la codificación de partituras musicales las cuales estén basadas en Humdrum

Para ello se tratará de dar una base sobre las gramáticas explicando notaciones para representarlas y metodologías a seguir para que ésta contemple el mayor número de casos para poder representar una notación. También expondremos las diferentes utilidades que se le puede dar a ésta en un futuro.

1.1.2. Requisitos

Para la correcta comprensión de este trabajo es necesario tener una base de teoría musical ya que hablaremos de ella constantemente, aun así, trataremos de explicar de la mejor manera la mayor cantidad de términos específicos. De esta manera podremos obtener una mayor comprensión de la lectura.

También es conveniente que el lector tenga ciertos conocimientos previos de teoría de la computación para enfrentarse a este documento.

Además de los diagramas, tendremos una pila de ejemplos para apoyarnos en la comprensión del tema a tratar en su momento.

1.2. Contexto

1.2.1. Historia

A lo largo de la historia se han diseñado multitud de máquinas y computadores con el fin de intentar mejorar o acelerar la precisión de los cálculos. Podríamos empezar por Charles Babbage, inventor de la Máquina Analítica en 1840 que era capaz de realizar cálculos programables con saltos condicionales y bucles, para aquel entonces esta máquina era lo más cercano a lo que conocemos hoy en día como un computador. Pero no es hasta 1936 cuando Alan Turing formalizó la primera idea abstracta de un computador, donde una máquina lee y escribe ceros y unos de una cinta infinita, denominada actualmente como memoria, donde hay un conjunto finito de instrucciones elementales, al que hoy en día llamamos programa. La Máquina de Turing fue uno de los avances más importantes de la informática ya que esta podría considerarse como un autómata capaz de reconocer lenguajes formales. Los avances realizados por Turing provocó que en la década de 1940 hubiera una gran cantidad de desarrollo de máquinas de computación electrónicas cada vez más rápidas y precisas.

En 1945 John Von Neumann propuso una arquitectura en la que junta dos claves de los computadores: almacenamiento de un programa en memoria y un conjunto de instrucciones de procesamiento. Pero no es hasta 1948 cuando se contruye el primer computador con esta arquitectora en Manchester capaz de hacer realizar saltos condicionales y direccionamiento indirecto.

Con el tiempo los computadores y la forma de programar fue cambiando, mejorando en rendimiento y tamaño. Empezando con los computadores UNIVAC que fue la primera computadora comercial fabricada en Estados Unidos en 1951 con tarjetas perforables. Pero poco después apareció el primer lenguaje de un nivel algo mas elevado que este código máquina, el ensamblador.

1.2.2. Ensamblador

Con el ensamblador, en lugar de escribir el código en binario, el programador tenía una serie de notaciones mnemotécnicas que simplificaban y mejoraba la notación anterior. Una forma de decirlo, era un lenguaje más legible que código máquina, más sencillo de implementar y con mayor facilidad de encontrar errores en el.

Lo que el ensamblador trataba de hacer es una traducción de estas reglas mnemotécnicas(lenguaje ensamblador) a código máquina(código binario).

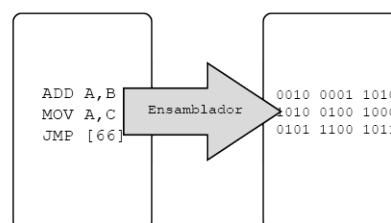


Figura 1.1: Proceso de ensamblador.

1.3. Compiladores e intérpretes

1.3.1. Compiladores

Como hemos dicho un ensamblador es un programa que se encarga de traducir el lenguaje ensamblador a código máquina. Con el paso del tiempo, se fueron desarrollando otros traductores de otros formatos de códigos/reglas que eran traducidos a ensamblador y este a su vez a código máquina.

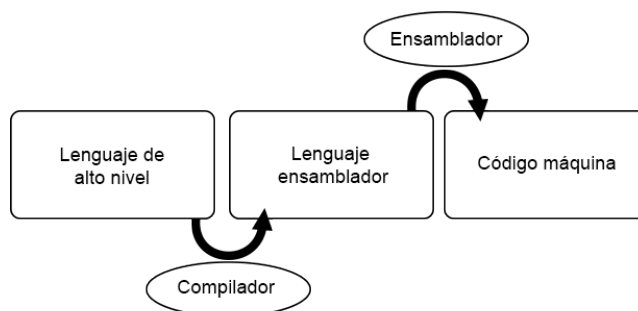


Figura 1.2: Pasos de Lenguaje de alto nivel a código máquina.

Estos traductores de código de más alto nivel se denominaron compiladores. Estos lenguajes nos permiten una manera de programar más sencilla y dependiendo el lenguaje utilizado de una manera más detallada según el ámbito en el que queramos hacerlo. Una vez desarrollado el apartado de los compiladores, comenzaron a desarrollarse diferentes lenguajes. El primero de ellos fue FORTRAN en 1956, y lo siguieron muchos otros pasando por los principales de hoy en día como son C, C++, Java o Swift.

Con el paso del tiempo los compiladores han sido desarrollados de manera más compleja, lo que ha provocado unos lenguajes con mejores prestaciones. Para poder traducir el código a ensamblador los compiladores cada vez deben realizar más tareas lo que produce que se hayan desarrollado las librerías, fragmentos de código escrito por otros programadores para simplificar la complejidad de lo que queremos implementar.

1.3.2. Fases de la compilación

Para lograr comprender las fases de la compilación, su recorrido y su finalidad debemos explicar previamente un término que hemos mencionado previamente en diversas ocasiones, qué es un lenguaje de programación.

1.3.2.1. Lenguaje de programación

Un lenguaje de programación es un lenguaje formal capaz de asimilar diferentes reglas conjuntas con el fin de poder controlar diferentes comportamientos de una computadora. Éstos tienen sus propias ‘palabras’ que las denominaremos *tokens* y con la unión de estos *tokens* rellenaremos las reglas que nos indicarán si el lenguaje en el que estamos programando está escrito de manera correcta o incorrecta. A este conjunto de reglas la llamaremos gramática, apartado donde nos centraremos más a fondo ya que es el tema principal de este trabajo.

1.3.2.2. Proceso de compilación

Como ya hemos explicado antes, un compilador es una herramienta que nos permite traducir un lenguaje de programación ‘A’ a un lenguaje de programación ‘B’. Para este proceso el compilador realiza diferentes procesos llegar al código de destino, como podría ser el lenguaje ensamblador para que este después sea ensamblado y traducido a código máquina, partiendo del código de inicio o código fuente como llamaremos a partir de ahora. Estos procesos hacen

que el compilador pase una serie de normas para comprobar que el código está correctamente escrito y no haya ningún tipo de errores, ya sean léxicos, sintácticos o semánticos. En la siguiente figura podemos observar un esquema del funcionamiento que realiza un compilador.

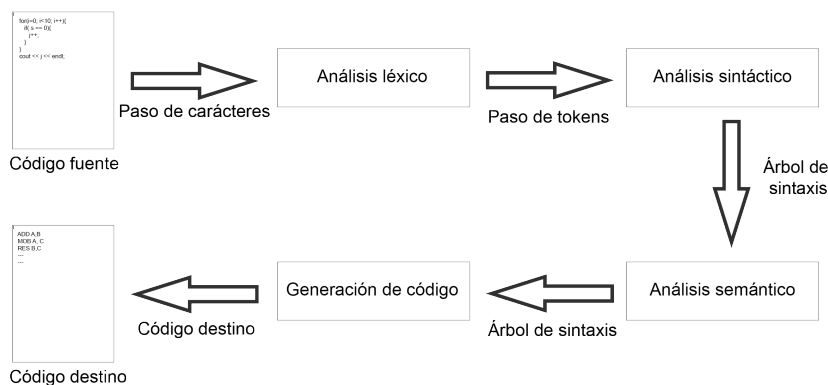


Figura 1.3: Estructura básica de un compilador

A simple vista en la imagen divisamos dos tipos de figuras, rectángulos y flechas, el primero corresponde la acción que se realiza en ese momento y la segunda el flujo de datos que vamos a enviar de una posición origen a una posición destino, donde el nodo inicial y final correspondiera al código inicial y al código destino respectivamente. A continuación, pasamos a explicar cada una de partes de la imagen anterior.

Código fuente

El código fuente es la información inicial que obtendrá el compilador para ser traducido que suele almacenarse en ficheros, ya puede ser uno o varios. Nuestro compilador tratará de leer dichos ficheros almacenando los caracteres en el computador y pasándolo a la siguiente acción.

Análisis Léxico

El analizador léxico, respecto al compilador, es el único módulo que maneja el fichero de entrada o código fuente, se encarga de abastecer al analizador sintáctico una serie de unidades lógicas denominadas *tokens* o elementos lógicos que es el resultado de la agrupación de los caracteres previamente mencionados del código fuente.

Éste suele ser una función o un método que es llamado por el analizador sintáctico cada vez que se necesitara conocer un nuevo *token* para que el proceso de traducción siga.

Existen diversidad de tipos de *tokens* como puede ser:

- Palabras reservadas. Como es el caso de los diferentes códigos más famosos de programación como C o Java, hay palabras reservadas básicas como **if**, **else** que dan lugar a operaciones lógicas.
- Símbolos especiales. Como puede ser **+** o **&&**, es decir, operadores aritméticos o lógicos entre otros

- Cadenas no específicas. Estos *tokens* suelen ser **identificadores** o **números** entre otros que suelen ser utilizados para uso de variables internas del código.

Una cadena que ya ha sido reconocida como un *token* específico es denominada como **lexema**, éste no tiene un papel estructural, ya que esto se encargará de realizarlo el siguiente apartado, pero sí tiene un punto de vista semántico y nos servirá para identificar, si lo hay, cualquier tipo de error de escritura, ya que junto al *token* almacenamos la fila y la columna en el que este se encuentra en el código fuente.

Para identificar un *token*, se deberá pasar carácter a carácter al analizador léxico el cual irá identificándolo por medio de un diagrama de transición previamente programado. Un diagrama de transición es la representación de los estados que puede tomar un sistema y muestra los eventos que implican un cambio a otro estado. Éstos parten de un estado inicial y concluyen en uno final representado por nodos, el paso de un nodo a otro se realiza mediante eventos que los representaremos como aristas y le atribuiremos la acción de estos eventos. En la siguiente imagen tenemos sencillo ejemplo para lograr entenderlo.

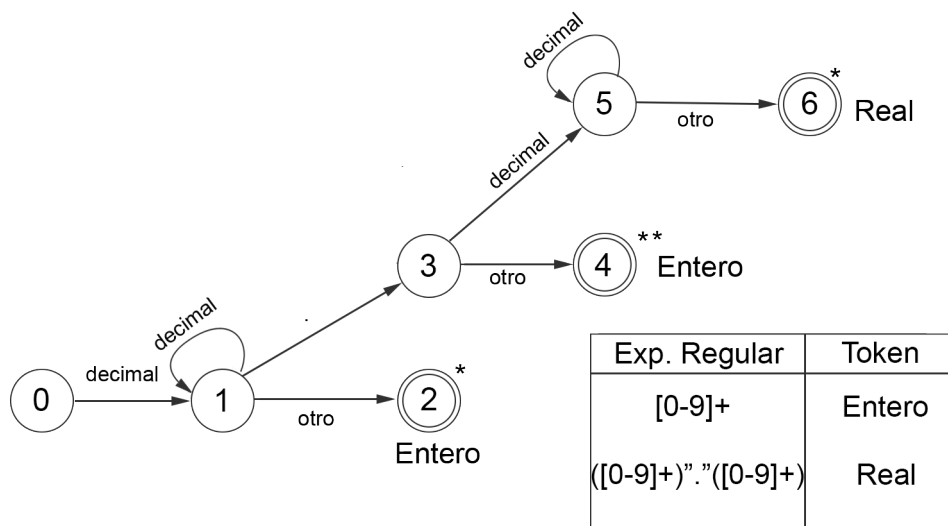


Figura 1.4: Ejemplo de diagrama de transición de un compilador donde se registran los números enteros y los reales

Dado el diagrama de transición, podemos implementar un programa (analizador léxico) capaz de identificar *tokens* que correspondieran a números reales y números enteros, sabiendo que tendremos tres tipos de entradas correspondientes a **decimal** que serán los números del 0 al 9, el punto y cualquier otro valor que lo denominamos **otro**.

Digamos que tenemos una traza, por ejemplo 1.2a nuestro analizador cogería el primer carácter, el decimal 1 y estaría en la posición 0 del diagrama, al ser un decimal, haríamos la transición a la posición 1 y cogeríamos el siguiente carácter, ' . ', al ser un punto la transición

la haríamos a la posición 3 y no a la dos ya que para esta debería ser cualquier otro valor que no fuera un decimal o un punto. A continuación, analizaríamos el siguiente carácter, el 2, por el mismo método que antes pasaríamos a la posición 5 ya que para la 4 debería ser cualquier caracter tipo **otro**. Ya en el 5 leeríamos el último carácter que faltara, en nuestro caso **a**, al no ser tipo decimal ni punto correspondería a un carácter tipo ‘otro’, lo que nos haría concluir el *token*, como indica el doble círculo, retrocediendo un carácter, éste es el significado del asterisco, y obteniendo un *token* **REAL** bajo el lexema 1.2.

Una vez reconocido todo los caracteres del código y pasados a *tokens* pasaremos a la la acción del analizador sintáctico.

Análisis sintáctico

Como hemos mencionado, el analizador sintáctico obtiene los *tokens* previamente mencionados llamando al analizador léxico. Éste irá leyendo dichos *tokens* a la vez que genera la traducción para el código destino, comprobando que la sintaxis, es decir, la agrupación de los *tokens* en el orden de entrada son correctos.

En definitiva, el análisis sintáctico sirve para comprobar que el orden de los *tokens* es el correcto, esta forma de comprobar que el código está escrito de manera correcta lo realiza por medio de lo que denominamos **Gramática Formal**.

Una gramática formal (‘G’) es una estructura que representa un conjunto finito de reglas que describen toda la secuencia de *tokens* pertenecientes a un lenguaje específico *L*.

La estructura algebraica de una gramática está formada por los siguientes elementos o nodos, como denominaremos a partir de ahora:

$G = \{ NT, T, S, P \}$ en el que:

- NT es un conjunto de nodos No Terminales.
- T es el conjunto de nodos Terminales.
- S es el símbolo o nodo inicial de la gramática.
- P es el conjunto de reglas de producción.

Esto quiere decir, dado un símbolo inicial **S**, el cual será un nodo no terminal, generará un número de reglas de producción **P** con nodos terminales **T** y nodos no terminales **NT**. Esos nodos no terminales a su vez derivarán otras reglas que al igual que **S** obtendremos más nodos terminales y no terminales, con una finalidad de obtener al final del árbol nodos terminales ya que éstos no generan reglas.

$$\begin{aligned} S &\rightarrow T \text{ id } R \\ T &\rightarrow \text{int} \\ R &\rightarrow \text{coma id } R \\ R &\rightarrow \text{puntoycoma} \end{aligned}$$

Tabla 1.1: Gramática de una declaración de variables tipo entero.

Éste es un ejemplo de una gramática sencilla en la cual tenemos un nodo inicial, dos nodos no terminales y cuatro nodos terminales:

$$S \rightarrow w$$

- Nodo inicial (S) = $\{S\}$
- Nodos no terminales (NT) = $\{T, R\}$
- Nodos terminales (T) = $\{id, int, coma, puntoycoma\}$

Esta gramática representaría la declaración de una o varias variables de tipo entero. Para explicar el funcionamiento de la gramática mejor, vamos a representar un ejemplo con un árbol lógico bajo el código de entrada:

```
int i, j;
```

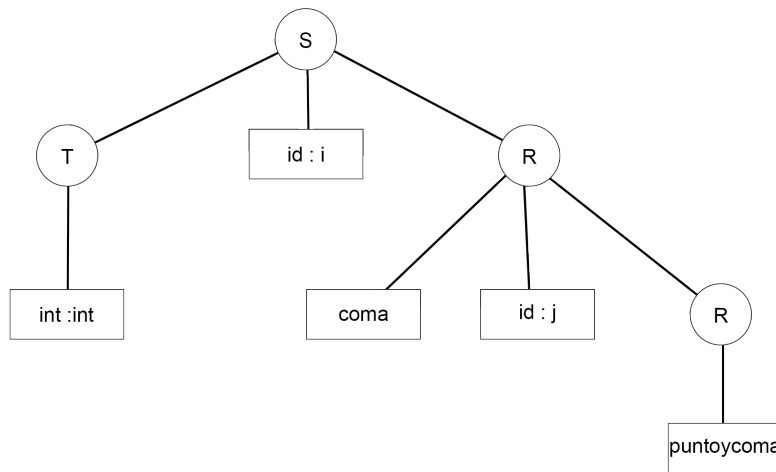


Figura 1.5: Árbol generado a partir del ejemplo para la gramática anterior.

En la figura anterior podemos observar un árbol lógico de decisiones representando el ejemplo previamente mencionado. En él podemos observar los nodos terminales y los nodos no terminales, representados respectivamente con rectángulos y círculos. Como vemos el nodo inicial o nodo raíz va derivando en otros nodos no terminales hasta llegar a la parte más baja de la ramificación con nodos terminales en los que, estos últimos, corresponderán a un tipo de *token* específico.

Una gramática libre de contexto (CFG), es una gramática formal en la que cada regla de producción está formada por un símbolo no terminal y una cadena de nodo terminales y/o no terminales. Se denomina gramática de libre contexto porque el nodo no terminal S puede ser siempre sustituido por una cadena de caracteres w sin tener en cuenta el contexto en el que ocurra. Un problema que tiene los algoritmos de un analizador sintáctico es su coste temporal, ya que gira entorno a $O(n^3)$, que es demasiado alto. Para solucionar este problema y reducir su coste temporal a lineal, $O(n)$, existen dos estrategias que vamos a tratar a continuación:

- Análisis sintáctico descendente o ASD.

- Análisis sintáctico ascendente o ASA.

Estos tipos de analizadores pueden ser implementados escribiendo a mano las reglas a través de variables y estructuras de control o usando generadores automáticos, ejemplos de este último son ANTLR y yacc/bison, los cuales hablaremos más adelante, mientras que la implementación a mano de estos analizadores son recomendables únicamente para gramáticas simples, ya que para una gramática muy compleja sería muy difícil de implementar.

Análisis sintáctico descendente

Este tipo de análisis trata de reproducir la derivación por la izquierda de la cadena de entrada. respecto a nuestro ejemplo de gramática anterior, Tabla 1.1, quedaría de la siguiente manera:

$$\begin{aligned}
 S &\rightarrow T \text{ id } R \\
 &\rightarrow \text{int id(i) } R \\
 &\rightarrow \text{int id(i) coma id(j) } R \\
 &\rightarrow \\
 &\text{int id(i) coma id(j) puntoycoma}
 \end{aligned}$$

Tabla 1.2: Representación de análisis descendente

En este caso podemos observar cómo desde un nodo inicial **S** se va derivando su regla y colocando los *tokens*. Se le denomina análisis descendente porque comienza desde el nodo más alto del árbol y va descendiendo por los nodos no terminales hasta llegar a los nodos terminales y comprobando que la sintaxis está bien hecha.

Para realizar bien el analizador sintáctico descendente la cadena de entrada debe encontrar por medio de las reglas de producción una representación de dichas reglas en el orden correcto. Para ello, solo debe existir un árbol de derivación posible para cada sentencia, es decir, la gramática no debe ser ambigua y para cada nodo en el que nos encontremos y con cualquier símbolo leído, siempre va a haber no más de una transición posible desde ese nodo con un símbolo (*determinista*). Finalmente, no puede ser una gramática recursiva por la izquierda ya que entraríamos en bucle infinito al derivar, esto quiere decir que entraría en bucle infinito al expandir un nodo no terminal a su regla de producción.

Los analizadores sintácticos descendentes, con una gramática libre de contexto, las entradas son de izquierda a derecha y las contrucciones de derivaciones son por la izquierda de una sentencia los denominamos **analizador sintáctico LL**.

Los analizadores *LL* son llamados analizadores *LL(k)* si necesitan analizar un número de *k* elementos por delante del *token* actual, es decir, si hubiera para una gramática tal analizador y pudiera analizar el código sin vuelta atrás (*Backtracking*), término que se emplea en algoritmos para buscar el retroceso en un árbol lógico y encontrar la solución deseada, entonces esta gramática la denominaríamos gramática *LL(k)*. Las gramáticas más populares son las *LL(1)* que a pesar de poder ser muy restrictivas sólo necesita ver el siguiente *token* para hacer el análisis sintáctico.

En concreto, una gramática cumple la condición *LL(1)*, si para todos los nodos no terminales, no existen símbolos comunes en los conjuntos de predicción de sus reglas, estos conjuntos

de predicción son los que ayudan a decidir que regla utilizar en cada paso. Para saber si una gramática no es $LL(1)$ se deben dar los siguientes casos:

- Que tenga recursividad por la izquierda.

$$P \rightarrow P \dots$$

$$P \rightarrow \dots$$

- Que tenga factores comunes por la izquierda.

$$P \rightarrow \alpha S_1$$

$$\dots$$

$$P \rightarrow \alpha S_2$$

- Que haya ambigüedad.

$$\begin{aligned} S &\rightarrow S + S \\ S &\rightarrow S - S \\ S &\rightarrow \text{numero} \end{aligned}$$

Esta gramática sería ambigua ya que podríamos representar de dos formas diferentes una misma entrada como vemos a continuación con la cadena de entrada $1 + 1 - 1$:

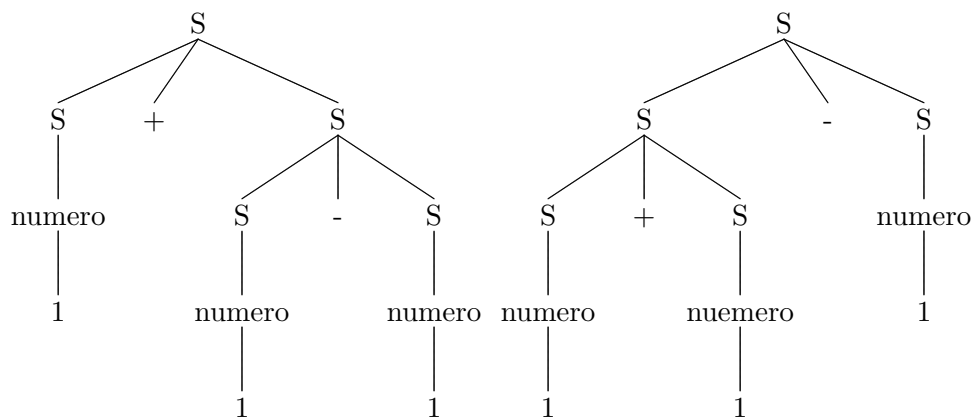


Tabla 1.3: Ejemplo de gramática ambigua

Una vez explicado esto y viendo la siguiente figura podemos saber a qué nos referimos con gramáticas $LL(1)$.

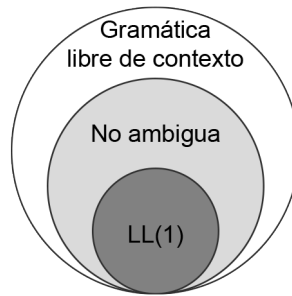


Figura 1.6: Las gramáticas $LL(1)$ como subconjunto de las gramáticas libres de contexto y no ambiguas.

En definitiva, una gramática $LL(1)$ es una gramática la cual está libre de contexto y, además, no es ambigua.

Análisis sintáctico ascendente

Los analizadores sintácticos ascendentes se le denominan de esta manera (en inglés *bottom-up*) porque su pretensión es la de construir un árbol sintáctico para una determinada cadena de entrada, empezando por sus nodos terminales y construyendo el árbol con sus nodos no terminales hasta llegar a la raíz o nodo inicial. También se le puede considerar a este proceso como la reducción de una cadena de símbolos al símbolo inicial de la gramática, es decir, una derivación en sentido inverso. Para detallar un poco más, en cada paso del proceso, una cadena que coincida con la parte derecha de la producción se reemplaza por el símbolo no terminal de la parte izquierda de la producción. A continuación vamos a observar un ejemplo de ello por medio de la gramática de el Tabla 1.1:

Para el ejemplo: `int i, j;`

int	id(i)	coma	id(j)	puntoycoma	←
T	id(i)	coma	id(j)	puntoycoma	←
T	id(i)	coma	id(j)	R	←
T	id(i)	R			←
S					

Tabla 1.4: Representación de análisis ascendente.

En el ejemplo podemos ver que los símbolos encuadrados serán los que vamos a intentar simplificar a una regla, la cual en el siguiente paso la marcaremos de rojo, así paso tras paso hasta llegar al nodo raíz que nos verifique el correcto uso de la gramática.

Este tipo de analizadores son de tipo reducción/desplazamiento, analizador LR, los cuales reconocen lenguajes realizando dos operaciones, cargar y reducir. Lo que hacen es leer los *tokens* de la entrada e ir cargandolo en una pila, de forma que se puedan explorar los *tokens* que contiene ésta y ver si se puede corresponder con la parte derecha de alguna de las reglas de la gramática. Si es así se realiza una reducción, la cual consiste en sacar de la pila esos *n tokens* y en su lugar colocar el símbolo o nodo no terminal que le corresponde

a esa regla. En caso contrario se carga en la pila el siguiente *token* y una vez hecho esto se vuelve a intentar reducir hasta llegar al símbolo inicial de la gramática.

Cabe mencionar que este tipo de analizadores deben saber identificar muy bien las reglas de producción para que no se quede bloqueado el análisis, ni tenga que hacer retroceso (*backtracking*). Al igual que existen las gramáticas $LL(k)$ que hemos mencionado anteriormente, también están las gramáticas $LR(k)$ las cuales utilizan k elementos léxicos (*tokens*) de búsqueda hacia delante.

Estas gramáticas realizan eficientemente el análisis ascendente en el que no hay retroceso, ya que son reconocidas por un analizador $LR(k)$. Para construir un analizador de este tipo necesitamos un programa analizador LR, una tabla de análisis, que sirve para analizar el recorrido que puede llegar a realizar la derivación de todas las reglas de una gramática, y una pila en los que se van cargando los estados que pasa el analizador y los símbolos de la gramática que se van leyendo.

Análisis semántico

El análisis semántico del árbol de la sintaxis que recibimos empieza por detectar las incoherencias a nivel conceptual lo cual este análisis trata de generar los errores que el código fuente incorpora. Estos analizadores se les suelen denominar parsers, los cuales buscan de una forma más concienzuda los *tokens* que se generan del código inicial, y no puede ‘gestionar’ el analizador sintáctico.

Generación de código

Este apartado consiste en que, una vez comprobado que el árbol está ordenado correctamente y que todos los *tokens* tienen su lógica, el código es traducido por medio de sus *tokens* en el cual tenemos su lexema, y que correspondiera una traducción específica, al código destino finalizando, de esta manera, el proceso habrá finalizado.

1.3.3. Intérpretes

Un intérprete es un programa que analiza y ejecuta simultáneamente un programa escrito en un lenguaje fuente.

En la siguiente figura se representa lo que sería el esquema general de un intérprete, este lo podemos asemejar a una caja negra. En estos podemos identificar dos entradas: un programa **P** escrito en un lenguaje fuente **LF** junto con los datos de entrada. A partir de estos datos de entrada y mediante un proceso de interpretación se va produciendo unos resultados.

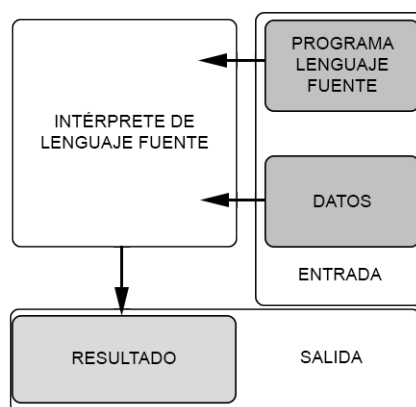


Figura 1.7: Estructura de un intérprete.

La diferencia entre compiladores e intérpretes es que los compiladores transforman el programa a un programa equivalente en un código diferente y éste es ejecutado dando los resultados en fase de ejecución.

1.3.3.1. Estructura de un intérprete

Cuando vamos a contruir un intérprete es conveniente utilizar una representación interna del lenguaje fuente a analizar. De esta manera, la organización interna de la mayoría de los intérpretes se descompone en varios módulos:

Conversión a representación interna: La entrada del código del programa P en Lenguaje Fuente, es analizado y se transforma a la representación interna del programa P.

Representación interna: Para crear esta representación se utilizan árboles sintácticos que obteniendo la traducción mediante una **Tabla de símbolos** se va rellenando con la traducción de estos símbolos de una manera ordenada y capaz de poder representar el lenguaje inicial.

Tabla de símbolos: Durante el proceso de traducción, es conveniente ir creando una tabla con información relativa a los símbolos que aparecen. La información que almacenaremos en esta tabla será un conjunto de **tokens** en el que su complejidad residira en la complejidad del lenguaje inicial.

Evaluador de Representación Interna: A partir de la implementación de la parte anterior se evaluan los datos de entrada y se lleva a cabo las acciones ne cesarias con el fin de obtener los resultados. Durante este proceso es necesario contemplar la aparición de errores.

Tratamiento de errores: Durante la evaluación se pueden general diferentes errores que el intérprete debe contemplar, errores que no son capaces de contemplar en la parte de la representación interna.

Dependiendo de la complejidad del código a analizar, el intérprete tiene la posibilidad de contener módulos similares a los del compilador tradicional: análisis léxico, análisis sintáctico y análisis semántico.

1.3.3.2. Ventajas de los intérpretes.

En general, la utilización de compiladores permite construir programas más eficientes que los correspondientes interpretados, ya que durante la ejecución de código compilado no es necesario realizar complejos análisis, además, un buen compilador es capaz de detectar errores y optimizar el código generado.

Por otro lado, los intérpretes, proporcionan una mayor flexibilidad que permite modificar y ampliar las características del lenguaje fuente. Ejemplos como APL, Prolog o Lisp surgieron en primer lugar como sistemas interpretados y posteriormente aparecieron compiladores.

Facilitan la metaprogramación, programas que crean otros programas, lo cual ayuda a la implementación de sistemas de aprendizaje automatizado.

Debido a que no existen etapas intermedias de compilación, los sistemas interpretados facilitan el desarrollo rápido de prototipos, potencian la utilización de sistemas interactivos y facilitan las tareas de corrección de errores.

1.4. Metaprogramas

Como ya hemos dicho anteriormente, hay dos formas de implementar analizadores, a mano, donde debes implementar tú mismo todas las funcionalidades de *estos*, o usando generadores automáticos, como es el caso de ANTLR y Flex/Bison, las herramientas más utilizadas hoy en día para la creación de compiladores, los cuales, a partir de un conjunto de reglas, ellos mismos generan un analizador sintáctico, aunque de maneras diferentes que explicaremos a continuación.

1.4.1. Flex [1] y Bison [2]

En el caso de Flex y Bison, creados por Vern Paxson y Robert Corbett respectivamente, son dos herramientas en las que, la primera trata el léxico de la gramática produciendo los *tokens* y pasándoselo al segundo que es el que crea el analizador sintáctico. Éste crea un analizador LALR, que es una mejora de los analizadores tipo LR(1). Como hemos explicado anteriormente, este tipo de analizadores sintácticos son ascendentes, los cuales generan los árboles de la sintaxis de abajo hacia arriba (down-top).

1.4.2. ANTLR

ANTLR [3] es un software desarrollado en *Java* por Terence Parr, que implementa de una manera eficiente analizadores LL(k), esto quiere decir que el analizador sintáctico generado por ANTLR es descendente (top-down), creando el árbol sintáctico de arriba a abajo, es decir, desde el nodo inicial de la gramática y los no terminales. Además es capaz de generar el análisis lexico y el propio código destino, por medio del análisis semántico y traduciéndolo. En pocas palabras, puede realizar prácticamente todas las fases de un compilador, pasándole una gramática e implementando los *parsers* y las traducciones necesarias.

2. Gramática para Kern [4] y Mens [5]

2.1. Introducción

Como hemos dicho al inicio la creación de nuestro trabajo se va a basar en la creación de una gramática formal en la que podamos representar el lenguaje musical. Para ello necesitamos, también, entrar un poco en contexto sobre la representación musical.

2.2. Representación musical

Los códigos musicales han sido utilizados desde sus inicios para facilitar la transcripción de sonidos, desde los neumas que se representaban el canto en monasterios medievales, al solfeo de música pedagógica del siglo actual. Muchos códigos musicales son de uso común, particularmente en la música pedagógica, solfeo, cuyo propósito principal es enseñar los procesos musicales mediante diferentes notaciones.

La digitación de los dedos en piano, los ideogramas los cuales denominamos tablaturas para guitarra y otros instrumentos con trastes, las notas de notación moderna y muchos símbolos especiales diseñados para transmitir formas particulares de tocar la batería son códigos de un tipo u otro. En este ámbito, la notación más utilizada y completa para representar la música es la notación musical común o notación musical moderna.

Entre el s.XIV y el s.XX fueron inventados diferentes códigos con el fin de hacer poder alcanzar objetivos que historiadores musicales, himnógrafos y etnomusicólogos pedían. Pero, desde el siglo XX a la actualidad se ha contemplado la creación de notación para computadores de una manera sólida, ya que debido a su eficiencia y su memoria resulta una oportunidad codiciosa para, por ejemplo, representar largas obras musicales con tantos atributos como queramos codificar, pudiéndolas compartir de una manera más sencilla que escritas a mano.

2.3. Kern y Mens

Actualmente hay varios sistemas de representación musical como puede ser DARMS [6], SCORE [7], EsAC [8]. Estos sistemas de representación han sido usados para codificar simbólicamente partituras, ya sea para renderizarlas a PDF o para reproducirlas, obteniendo de dichos símbolos una melodía.

Humdrum [9] es un software creado por David Huron en la década de los 80s. Dicho software es empleado con el fin de obtener diferentes recursos musicales a nivel computacional. Junto a su tipo de notación, `**kern`, se puede obtener PDFs de partituras e incluso se puede reproducir la melodía previamente implementada. Pero lo más importante de Humdrum es que es una herramienta para realizar diversas tareas de musicología digital. Esta notación, `**kern`, es una representación secuencial musical de manera vertical en la que cada columna representa una voz de cada melodía con el fin de representar finalmente una partitura.

Figura 2.1: Representación de **kern

En esta representación podemos implementar notación moderna occidental además de poder manipular los siguientes apartados:

- El tono: alteraciones musicales, claves, posición de claves, armadura, fórmula de compás
- La duración: silencios, puntos, ligaduras, ligaduras de expresión.
- Articulaciones y ornamentos: staccato, tenuto, pizzicato.

Como hemos dicho la representación de **kern es vertical, en el que cada voz de la melodía representa una columna, denominadas *spines* en **kern se puede representar varias voces en una misma melodía, cada vez que añadamos una columna con la iniciación de **kern será añadida una voz nueva como observaremos en el ejemplo siguiente.

**kern	**kern
*staff2	*staff1
*clefF4	*clefG2
*k[b-]	*k[b-]
*M3/4	*M3/4
=1-	=1-
2.r	8r
.	8dL
.	8g
.	8b-
.	8g
.	8dJ
=2	=2
8r	4dd
8GGL	.
8BB-	4r
8D	.
8BB-	4r
8GGJ	.
=3	=3
4GWw	8r
.	8ddL
8GGL	8b-
8BB-	8g
8D	8gg
8GJ	8b-J
=4	=4
4D	8aL
.	8gg
4d	8ff
.	8ee
4D	8ff
.	8a-J
=5	=5
*-	*-

Figura 2.2: Representación de ****kern**

Como podemos comprobar en la Figura 2.2 tenemos una representación de dos voces diferentes de una partitura por medio de ****kern**, en el código podemos observar dos *spines* diferentes los cuales, el de la izquierda genera la voz de arriba y la de la derecha la voz de abajo. Gracias a esto se pueden hacer escritos musicales de más de una voz con gran facilidad, únicamente añadiendo una columna nueva que ejercera de una nueva voz instrumental.

Hay más estilos de columnas en esta notación como ****text** el que podemos escribir bajo un pentagrama, pero este tema no lo vamos a abordar en este trabajo.

Una variante que si nos interesa de la representación ****kern** es ****mens**. Este tipo de representación acoge notación mensural blanca con muchas coincidencias con ****kern** a la hora de representarse, ya que este tipo de notación también es vertical y se gestiona de una forma muy parecida a la de notación moderna.

El defecto que lleva a esta representación es la falta de una gramática formal que es el tema correspondiente a este trabajo y que explicaremos a continuación.

2.4. Decisiones sobre la gramática

Para poder implementar una gramática hay que hacer cierto balance de cómo va a ser la complejidad de ésta y discutir de qué manera puede ser la más correcta u óptima de realizarla.

2.4.1. Analizador sintáctico descendente o ascendente

Para tomar una elección sobre qué analizador sintáctico escoger, el ascendente o el descendente, vamos a barajar sus ventajas e inconvenientes.

Como ya hemos explicado anteriormente con más detalle, un analizador sintáctico descendente genera el árbol de sintaxis desde el nodo inicial a las hojas (nodos terminales) derivando estos primeros nodos en reglas de producción compuestas de nodos terminales y nodos no terminales, mientras el analizador sintáctico ascendente genera este árbol de abajo hacia arriba, donde los nodos terminales se juntan creando reglas de producción has llegar al nodo inicial.

Para la elección de qué tipo de analizador vamos a escoger debemos entender qué ventajas e inconvenientes tiene cada uno. Por un lado, el analizador sintáctico ascendente es más eficiente respecto a la detección de errores respecto al descendente ya que va generando reglas conforme reconoce tokens hasta llegar al nodo raíz. Además, tiene un reconocimiento muy rápido, aunque no tanto como el descendente. Por otro lado, el analizador sintáctico descendente una gran ventaja reside en la facilidad de ser implementado, además, como ya hemos dicho, es un método de reconocimiento más rápido que el ascendente.

Dadas las ventajas y desventajas que hemos anotado, también debemos exponer qué herramienta nos conviene más utilizar, *ANTLR* o *Flex/Bison*, ya que serán las herramientas que necesitaremos para la creación de nuestra gramática.

Bien, como sabemos, una gran diferencia que reside entre *ANTLR* y *flex/bison* es que son analizadores sintácticos descendentes y ascendentes, respectivamente. Otra gran diferencia entre ambos es el apartado que engloba cada una de estas herramientas, ya que con la primera podemos obtener mejor flexibilidad a la hora de creación del lenguaje que con *flex/bison*, porque el primera engloba más fases de compilación que el segunda.

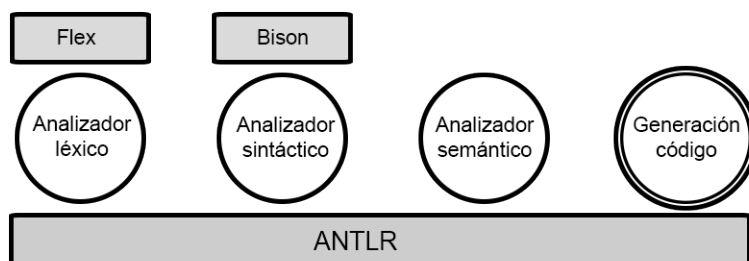


Figura 2.3: Comparación entre ANTLR y flex/bison

Sabiendo estos detalles que hemos explicado, la elección del formato de analización sintáctica descendente es la correcta para este trabajo, ya que la gran cantidad de *tokens* y de reglas a formar hace que sea una gramática compleja, además la utilización de ANTLR puede provocar que en un futuro mejore la semántica de ésta ya que será posible de implementar e incluso fabricar traductores a otros tipos de representación musical, como DARMS o SCORE por ejemplo.

Además, ANTLR soporta la notación basada en EBNF, el cual es una extensión del notación BNF (*Backus-Naur Form*) que es la notación formal para definir la sintaxis de un lenguaje. EBNF soporta símbolos que simplifican la especificación de las reglas de producción. En el siguiente apartado explicaremos como es la utilización de este notación.

2.5. Notación de gramática.

Como ya hemos dicho, para la generación de nuestra gramática vamos a usar una notación EBNF, la cual es una extensión en BNF, lo que produce una simplificación de las reglas a implementar con el uso de diferentes símbolos. Para comprender la notación deberemos saber el uso de los diferentes símbolos.

```
numero ::= digito + ( '.' digito + ) ?
digito ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Figura 2.4: Ejemplo de regla de gramática.

En la figura superior nos muestra un ejemplo de una regla de gramática con notación EBNF. Donde implementamos la gramática de un numero decimal o real.

Como podemos observar hay diferentes símbolos que puede ser que no reconozcamos para la creación de la notación de una gramática:

- `::=` : este símbolo se utiliza para definir una regla atribuida a un nodo no terminal inicial, es decir:
la regla 'numero' derivará en `(digito + ('.' digito +)`
- `?`: cuando utilizamos este símbolo significa que la regla puede tener o no el símbolo, regla o conjunto de éstos. En nuestro ejemplo lo podemos observar englobando a todo lo que tenemos dentro del paréntesis, esto querrá decir que la regla será valida aparezca o no los símbolos de dentro del paréntesis.
- `+` o `*`: estos dos símbolos son utilizados para expresar que hay 1 o más, o ninguno o más símbolos, en nuestro caso podemos observar `+` lo que significa que habrá, al menos, un `digito` inicial.
- `|`: es un símbolo para la representación *or* de una regla. En nuestro ejemplo lo podemos observar en la regla `digito` donde el resultado de ésta puede ser cualquier número comprendido entre 0 y 9.

La forma que utilizaremos de representar la gramática será por medio de los diagramas de sintaxis (*Railroad Diagram*) que describimos a continuación.

2.5.1. Diagramas de sintaxis

Los diagramas de sintaxis son una forma de representar una gramática libre de contexto, éstas representan una alternativa gráfica a la EBNF. Estos diagramas son visuales y resultan más sencillos de entender que la notación EBNF.

Para su construcción podemos diferenciar diferentes partes:

- La flecha inicial, donde comienza la regla de producción, con un flujo de izquierda a derecha
- Los rectángulos que representarán nodos no terminales de la gramática.

- Los rectángulos con bordes redondeados, estos serán los nodos terminales, escritos en negrita.
- Fin de la regla donde veremos punta de flecha hacia las dos direcciones.

Todos estas partes van unidas con una línea, con un flujo hacia la izquierda, pero en el que se pueden dar casos de bifurcaciones e incluso recursividad.



Figura 2.5: Ejemplo de diagrama de sintaxis

En la Figura 2.5 podemos observar un ejemplo de diagrama de sintaxis en el cual, va de izquierda a derecha y el primer paso que damos nos lleva un nodo no terminal denominado **no_terminal**, éste puede introducir a una recursividad que generará, tantas veces como haga el bucle, la regla **no_terminal**, una vez acabe tiene la posibilidad, o no, de encontrar el nodo terminal **terminal** incluso recorrerlo varias veces y tras ello finalizar.

2.6. Implementación de la gramática.

Una vez ya explicado los términos necesarios para lograr entender cómo se puede realizar el trabajo, toca ver que metodología seguiremos para la implementación de esta gramática.

2.6.1. Metodología.

Dado a que nos guiaremos por la notación de Humdrum el método más eficiente en nuestro caso será seguir una metodología de desarrollo guiado por pruebas el cual consiste en los siguientes pasos:

- Elegir un requisito: nuestro requisito sería sistematizar las reglas que generamos a partir de la documentación de ****kern** y ****mens**.
- Escribir una prueba: obtenido el requisito implementamos nuestra prueba unitaria con el fin de lograr resolverla.
- Verificar el fallo de la prueba: en caso de no fallar el requisito que hemos obtenido está paliado, mientras que si falla deberemos modificar las reglas para que supere la prueba correctamente.
- Escribir la implementación correcta: modificación de la regla de manera correcta con el fin de que la prueba sea verificada correctamente.
- Ejecutar las conjunto de pruebas: una vez superadas cada una de manera unitaria, deberemos ejecutar todas en su conjunto ya que al modificar unas puede haber repercutido en otras, si al llegar al fin no hemos logrado solucionarlo puede que necesitemos una refactorización de estas.

Este tipo de metodología requiere de unas reglas base y a partir de esta ir construyendo las diferentes reglas de producción que probaremos con los diferentes test y conforme vamos completandolas se irán dando errores de regresión que deberemos ir corrigiendo.

2.7. Formato de notación ****kern** y ****mens**

Antes de comenzar a explicar con detalles la creación de la gramática cabe explicar la estructura de ambas notaciones.

Como ya hemos explicado antes ****kern** y ****mens** son dos notaciones las cuales, la primera representa la notación musical moderna y la segunda la notación mensural blanca. La peculiaridad que tiene esta notación es su estructura, ya que se implementa de forma vertical (en columnas). Es una peculiaridad porque si hemos tenido una base previa a lenguajes de programación por lo general para implementar estos código se suele implementar de forma horizontal, es decir, en filas.

Con este formato, cada columna representará una voz nueva pudiéndose así crear partituras de varias voces. Al representarse cada voz en cada columna la visualización del código al programador podría resultar más entendible, ya que puedes ir implementando varias voces al mismo tiempo ya que en una misma fila obtendremos, por ejemplo, dos voces diferentes en el mismo momento de la partitura.

2.7.1. Alcance de nuestra gramática.

Nuestra gramática va a englobar tanto la notación ****kern**, como la notación ****mens**, es decir, vamos a poder implementar código con la finalidad de que nuestro analizador sintáctico descendente verifique que el código esta correctamente implementado. Dado en algunos aspectos ambas gramáticas son parecidas algunas de las reglas servirán para ambos tipos de notación, pudiendo compartirlas entre sí.

Además, hemos desechado ciertos aspectos que nuestra gramática no puede comprender y que se podrían tener en cuenta para trabajos futuros, como es el caso de la aplicación de varias voces, ya que deberíamos cambiar la forma de lectura de nuestra gramática a una forma horizontal. Más adelante, expondremos algunas de las mejoras y las posibles aplicaciones que puede obtener la gramática.

2.8. Especificación de la gramática

2.8.1. Léxico

Tabla 2.1: Léxico de la gramática.

TOKEN	DEFINICIÓN
LETTER_L	`L'
LETTER_M	`M'
PERTFECT	`0'
ONE	`1'
FOUR	`4'

THREE	`3'
TWO	`2'
FIVE	`5'
SIX	`6'
SEVEN	`7'
EIGHT	`8'
NINE	`9'
ZERO	`0'
SHARP	`#'
DOUBLESARP	`##'
FLAT	`-'
DOUBLEFLAT	`--'
NATURAL	`n'
SPACE	` '
GREATER	`>'
LESS	`<'
LEFTPAR	`('
; RIGHTPAR	`)'
LEFTBRACKET	`['
RIGHTBRACKET	`]'
LEFTCURBRACES	`{'
RIGHTCURBRACES	`}'
LETTER_K	`K'
LETTER_k	`k'
LETTER_S	`S'
LETTER_s	`s'
LETTER_u	`u'
LETTER_m	`m'
LETTER_T	`T'
LETTER_t	`t'
LETTER_W	`W'
LETTER_w	`w'
LETTER_U	`U'
LETTER_p	`p'
LETTER_i	`i'
LETTER_v	`v'
LETTER_r	`r'
LETTER_R	`R'
LETTER_I	`I'
LETTER_P	`P'
LETTER_A	`A'
LETTER_B	`B'
LETTER_C	`C'
LETTER_D	`D'
LETTER_E	`E'

LETTER_F	`F'
LETTER_G	`G'
LETTER_X	`X'
LETTER_a	`a'
LETTER_b	`b'
LETTER_c	`c'
LETTER_d	`d'
LETTER_e	`e'
LETTER_f	`f'
LETTER_g	`g'
LETTER_J	`J'
WORD_CLEF	`clef'
WORD_KERN	`kern'
WORD_MENS	`mens'
WORD_MET	`met'
DOT	`.'
BARLINE	` '
EQUAL	`='
SLASH	`/'
BACKSLASH	`\\'
ASTERISK	`*'
CIRCUNFLEX	`^'
APOSTROPHE	`\''
SEMICOLON	`;'
COLON	`:'
COLOURED	`~'
GRAVE_ACCENT	`''
EXCLAMATION	`!'
COMA	`,'
SLURS_COUNT	`&'
TOKEN_FINISH	`*-'

2.8.2. Sintáctico

Al representar dos notaciones diferentes con una sola gramática, compartiendo ciertas reglas entre ellas, la dificultad que ello representa es elevada. Como observamos en la tabla de abajo, que describe la sintaxis de la gramática en su totalidad tiene una notación EBNF en la que los nodos terminales estan en negrita.

Nuestra gramática parte de una regla (**startRule**) a la que bifurcará hacia dos reglas que representará las dos notaciones que vamos a representar, ****kern** y ****mens**. Cada una de estas notaciones son diferentes pero poseen ciertas estructuras muy parecidas en la que sabiendo representar una es sencillo aprender la otra.

Tabla 2.2: Reglas de la gramática.

startRule	kern_notation+ mens_notation+
mastercleff	ASTERISK WORD_CLEF clef
keysignature	ASTERISK LETTER_k LEFTBRACKET note_signature* RIGHTBRACKET
note_signature	noteName (FLAT SHARP)
notesuffix	SHARP DOUBLESARP FLAT DOUBLEFLAT NATURAL
pitch	noteName+ noteNameCl+
stem_direction	SLASH BACKSLASH
kern_notation	ASTERISK ASTERISK WORD_KERN mastercleff keysignature? timesignature? musicalcontent TOKEN_FINISH
timesignature	(ASTERISK LETTER_M fraction metter)?
fraction	number SLASH number
number	digit+
metter	ASTERISK WORD_MET LEFTPAR common_met RIGHTPAR
common_met	LETTER_c (BARLINE)?
musicalcontent	barlines? measure+ items
measure	items barlines
items	item+

item	notes rest changeconfiguration slurs ties
changeconfiguration	(mastercleff timesignature keysignature)+
slurs	LETTER_U? SLURS_COUNT* LEFTPAR SLURS_COUNT* RIGHTPAR
ties	LETTER_U? LEFTBRACKET note note RIGHTBRACKET LEFTBRACKET note RIGHTBRACKET
notes	beaming note chord
beaming	((note ties) chord) LETTER_L+ ((note ties) chord) LETTER_J+ ((note ties) chord) LETTER_L* partial_beaming ((note ties) chord) LETTER_J* partial_beaming
note	time DOT* pitch notesuffix? edit_accident? not_hide? ornaments? articulations? stem_direction?
time	digit +
rest	time DOT* LETTER_r
barlines	EQUAL digit* FLAT? (EQUAL digit*)? doubleBarline (EQUAL digit*)? leftRepeatBarline (EQUAL digit*)? rightRepeatBarline endBarline (EQUAL digit*)? doubleRepeatBarline
chord	(note ties) (SPACE (note ties))+
articulations	APOSTROPHE CIRCUNFLEX GRAVE_ACCENT COLOURED APOSTROPHE COLOURED APOSTROPHE CIRCUNFLEX CIRCUNFLEX CIRCUNFLEX SEMICOLON
ornaments	LETTER_t LETTER_T LETTER_m LETTER_M LETTER_W LETTER_w
partial_beaming	LETTER_K LETTER_k
edit_accident	(bracket parenthe)
doubleBarline	BARLINE BARLINE
rightRepeatBarline	COLON BARLINE EXCLAMATION

leftRepeatBarline	EXCLAMATION BARLINE COLON
doubleRepeatBarline	COLON BARLINE EXCLAMATION BARLINE COLON
endBarline	EQUAL EQUAL
mens_notation	ASTERISK ASTERISK WORD_MENS mastercleff keysignature? m_timesignature? m_musicalcontent TOKEN_FINISH
m_timesignature	ASTERISK WORD_MET LEFTPAR mensural_signs RIGHTPAR
m_musicalcontent	m_measure+ m_items
m_measure	m_items+ m_barlines
m_items	m_item+
m_item	m_note m_rest m_slurs m_ligature
m_note	m_rhythm m_dot? (m_perfect m_imperfect)? COLOURED? pitch (m_notesuffix (bracket parenthe)?)? stem_direction?
m_barlines	EQUAL FLAT EQUAL BARLINE BARLINE
m_rest	m_rhythm LETTER_r
m_slurs	SLURS_COUNT* LEFTPAR SLURS_COUNT* RIGHTPAR
m_ligature	LESS GREATER
m_dot	LETTER_p COLON
m_notesuffix	LETTER_U? notesuffix
mensural_signs	signs_c signs_p
signs_c	(LETTER_C) (BARLINE (THREE TWO SLASH (TWO THREE) LETTER_r)? TWO THREE DOT (BARLINE)? LETTER_r)?
signs_p	(PERTFECT ((TWO THREE BARLINE (THREE)? SLASH THREE DOT)? THREE SLASH TWO THREE TWO))
m_perfect	LETTER_p

m_imperfect	LETTER_i
noteName	(LETTER_a LETTER_b LETTER_c LETTER_d LETTER_e LETTER_f LETTER_g)
noteNameCl	(LETTER_A LETTER_B LETTER_C LETTER_D LETTER_E LETTER_F LETTER_G)
m_rhythm	(LETTER_X LETTER_L LETTER_S LETTER_M LETTER_U LETTER_s LETTER_m LETTER_u)
not_hide	LETTER_X
bracket	LETTER_I
parenthe	LETTER_P
digit	(ONE TWO THREE FOUR FIVE SIX SEVEN EIGHT NINE ZERO)
clef	(LETTER_G TWO LETTER_F TWO LETTER_F THREE LETTER_F FOUR LETTER_C FIVE LETTER_C FOUR LETTER_C THREE LETTER_C TWO LETTER_C ONE LETTER_G ONE LETTER_G TWO)

2.9. Explicación de nuestra gramática

Como punto de inicio tendremos una regla inicial, que denominamos en nuestro caso `startRule`, esta creará una bifurcación entre ambas notaciones (`**kern` y `**mens`). Cabe mencionar que en las reglas podremos visualizar dos estilos de símbolos los que están entre comillas y los que no. Los primeros representarán a los nodos terminales mientras que los otros a los nodos no terminales.

```
startRule ::= kern_notation + | mens_notation +
```

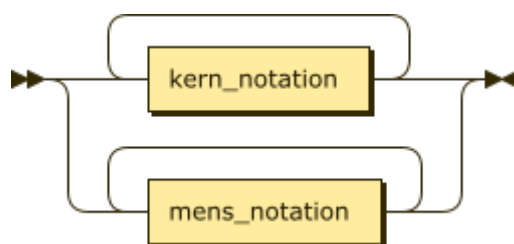


Figura 2.6: Regla inicial

Como hemos dicho previamente y podemos observar en la Figura 2.6, esta regla crea la separación entre ambas notaciones dándole la importancia de la elección de una notación u otra, además una vez elegida podemos observar que no puede cambiar a la anterior, será o `**kern` o `**mens`.

2.9.1. `**kern`

Una vez elegida la notación a utilizar pasamos a explicar una de ellas, `**kern` [10], esta notación representa a la que utilizamos actualmente en música, la notación moderna occidental.

Para ello debemos establecer una cabecera y un final que abarque a este tipo de notación y además la identifique como que es `**kern`:

```
kern_notation ::= '*' '*' 'kern' mastercleff keysignature ?  
timesignature ? musicalcontent '*-'
```

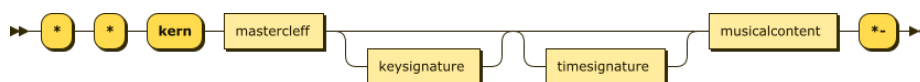


Figura 2.7: Regla que deriva a la notación `**kern`

Esta regla lo que conseguimos es realizar una división de lo que sería el pentagrama inicial en las cuatro piezas más básicas y fundamentales: clave, armadura de compás, tiempo de compás y el contenido musical.



Figura 2.8: Representación de regla `**kern`

Como podemos observar en la figura anterior de una manera más visual, la regla inicial de `**kern` divide la clave que contendrá el pentagrama (resaltado mediante un recuadro azul, Figura 2.8), la armadura de dicho pentagrama (resaltado mediante un recuadro roja), el tiempo de compás que tendrá la composición (resaltado mediante un recuadro verde) y el contenido musical (resaltado mediante un recuadro magenta), además algunas de estas partes puede que no nos las encontremos ya que no son imprescindibles en la notación.

2.9.1.1. Clave

Para este apartado, la regla es muy sencilla [11].



Figura 2.9: Representación de la clave del compás.

Dado un encabezado con `*clef` y asignando una de las claves que mostramos en la Figura 2.10 (regla `clef`) de la notación podemos formar la clave de nuestro pentagrama.

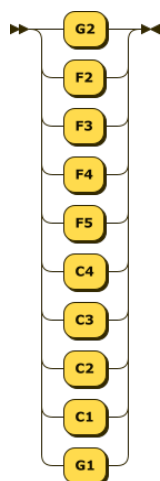


Figura 2.10: Representación de las diferentes claves que podemos asignar.

Este apartado representaría el recuadro azul de la Figura 2.8. Un ejemplo de ello, sería `*clefG2`.

2.9.1.2. Armadura de compás

La aplicación de la armadura [12] del compás se realizará con el *token* * seguido del *token* k entre corchetes los cuales contendrán las notas con sus sostenidos o bemoles.

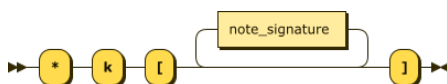


Figura 2.11: Representación de armadura de compás.

Viendo el sencillo formato que representa la figura anterior, nos queda por saber como especificar que notas queremos que la armadura las convierta en bemol o sostenido.

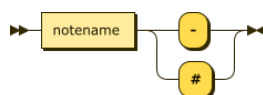


Figura 2.12: Representación de armadura de compás.

Esta regla, Figura 2.12, sirve para especificar si la armadura será en bemol o en sostenido y no tener la opción de alternar entre ambos. Para poder completar esta regla únicamente nos quedaría elegir las notas de la armadura (a, b, c, d, e, f, g), de esta manera podríamos completar el recuadro rojo del pentagrama de la Figura 2.8. Un ejemplo de armadura sería *k[a-b-c-], lo que significaría que las notas *la*, *si*, y *do*, se tocarían con bemoles.

2.9.1.3. Tiempo de compás

El tiempo del compás [13] lo definimos con una fracción precedido de los *tokens* * M.

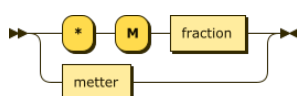


Figura 2.13: Representación de tiempo de compás.

Como observamos en la figura anterior tenemos una bifurcación en la que podemos elegir si queremos un tiempo de compás por fracción o por signo de compás.

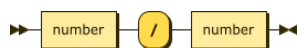


Figura 2.14: Representación de la fracción del tiempo de compás.

Esta figura, marca una elección de la regla de la Figura 2.13 haciendo una fracción, un ejemplo de esta representación sería *M6/8, este tiempo de compás representaría al recuadro verde de la Figura 2.8.

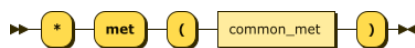


Figura 2.15: Representación de la estructura rítmica.

La otra opción que tenemos es la de la Figura 2.15. Esta opción trata de estructuras rítmicas que representan un tiempo de compás, como el *compasillo*, por ejemplo. En la notación de `**kern` tenemos dos formas de estructuras rítmicas como observaremos en la Figura 2.16

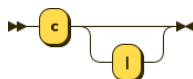


Figura 2.16: Estilos de estructura rítmica.

Una manera de poder representar una estructura rítmica en nuestra gramática es, `*met (C)`.

2.9.1.4. Contenido musical

Ya hemos representado tres de los cuatro recuadros de nuestro pentagrama, Figura 2.8. A continuación, pasamos a representar el último recuadro. Esto representa la parte más amplia de nuestra gramática en torno a la notación de `**kern`. En este apartado es donde tendremos gran parte de la representación musical: ritmo, notas, alteraciones...

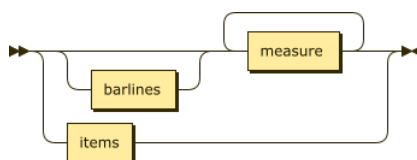


Figura 2.17: Representación de contenido musical.

En esta figura observamos las dos formas diferentes que vamos tener para representar la notación. Un compás (*measures*) es una medida en la que mediante con barras de compas (*barlines*) hay una ristra de notas de tal manera que respete signo del compás.

El otro tipo de representación no contempla barras de compás, el cual será una acumulación de notas hasta llegar a su fin.

2.9.1.5. Figuras

A lo largo del contenido musical vamos a disponer de diferentes figuras [15] (*items*) para poder representarlo.

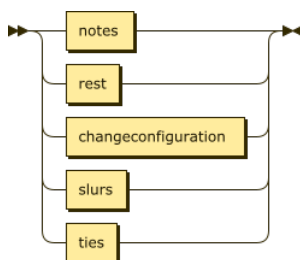


Figura 2.18: Representación de las figuras.

Cada una de estas figuras representa una parte fundamental para poder crear notación completa. Principalmente, tenemos las notas [20] que representan la altura y la duración relativa del un sonido, aunque éstas pueden ir agrupadas en diferentes tipos.

Notas

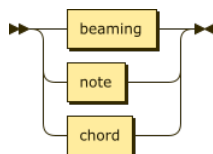


Figura 2.19: Representación del diferente formato de notas.

Hemos especificado tres tipos de notas diferentes: nota (*note*), acorde (*chord*) y la agrupación rítmica de corcheas o notas con tiempos más cortos (*beaming*). Debemos decir que, tanto los acordes como estas agrupaciones rítmicas están formadas por diferentes notas.

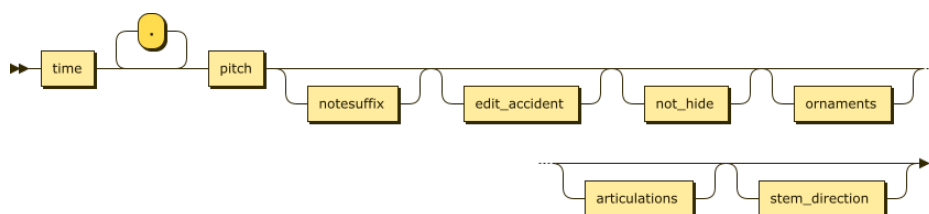


Figura 2.20: Representación de una nota.

Una nota en nuestra gramática corresponde a una figura musical que tiene una gran cantidad de formas para representarse. Tiene un tiempo [21], el cual en ****kern** se representará con números siendo múltiplo de dos las notas empezando con la redonda (valor 1), siguiéndole la blanca (valor 2), después la negra (valor 4) y así hasta las notas más cortas, también hay más largas, como la cuadrada por ejemplo (valor 0). Nuestra gramática acepta todos los números, pero el analizador semántico el que debe comprobar los números seleccionados.

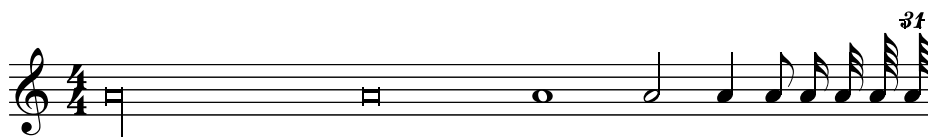


Figura 2.21: Representación visual del tiempo de la nota.

Además, existe una forma de alargar una nota que controlamos en nuestra gramática como es el puntillo [22] (*dot*) que como vemos en la Figura 2.20, se representa con el *token* `.` y puede insertarse los que consideremos.

Para representar nuestra nota es fundamental darle un tono que representamos con `a`, `b`, `c`, `d`, `e`, `f`, `g` (notación inglesa), equivalente a *la*, *si*, *do*, *re*, *mi*, *fa*, *sol*. Sabiendo la altura, ya podríamos crear notas. Por ejemplo, para representar una negra en la tono de sol con un puntillo sería de esta manera:

4.g



Figura 2.22: Representación de una nota.

Además, podemos complementar la nota con alteraciones [23] (Figura 2.23) en bemol, sostenido o natural.

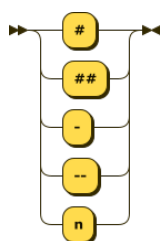


Figura 2.23: Representación de las alteraciones.

Estas alteraciones podran ser editadas con una `I` o `P` para ponerlos entre paréntesis o corchetes.

Además tenemos una gran serie de detalles para la modificación de las notas como puede ser las articulaciones (Figura 2.24a), los ornamentos (Figura 2.24b).

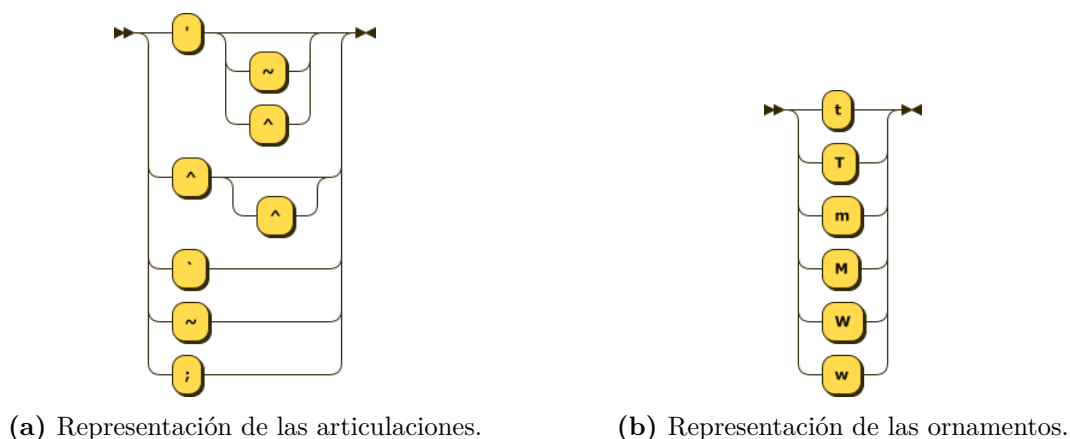


Figura 2.24: Representación de las articulaciones y los ornamentos.

Un cambio a más agudo o más grave con la representación de una nota es la subida o bajada de *octava* [24], en nuestra gramática también podemos representarla con la agregación de la misma nota repetidas veces para subir de octava y con la utilización de mayúsculas para bajarla. Por ejemplo, si tenemos una *g* (sol) y queremos subir una octava, simplemente deberíamos añadir otra *g*, *gg*. Por otro lado si queremos bajar deberíamos sustituir esa *g* por su mayúscula *G* y aumentar más mayúscula cuanto más queramos bajar.

Por último, podemos modificar la dirección de la plica.

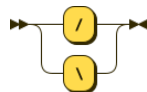


Figura 2.25: Cambio de dirección de la plica.

Podemos cambiar la dirección de la plica, hacia arriba o hacia abajo con los *tokens* `\` y `/` como muestra la Figura 2.25. Con esto tendríamos la información necesaria para poder generar diferentes notas dada la forma de representar de la Figura 2.20.

Acordes

La creación del acorde es muy sencilla una vez conocida la base de una nota.

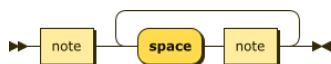


Figura 2.26: Representación de un acorde.

Como podemos observar en la Figura 2.26, la representación de un acorde es la agrupación de notas, separadas por un espacio, por ejemplo, el acorde de *sol* (sol, mi, do) en **negrita** se representaría de la siguiente manera:

4g 4e 4c



Figura 2.27: Representación del acorde de sol.

Agrupación de corcheas o figuras de ritmo mayor

La creación de corcheas o figuras con mayor ritmo simboliza la unión de dos notas o más con más ritmo que la negra, nuestra gramática no comprende que la nota sea corchea u otra. Pero si comprende la agrupación entre figuras como observamos en la siguiente figura.

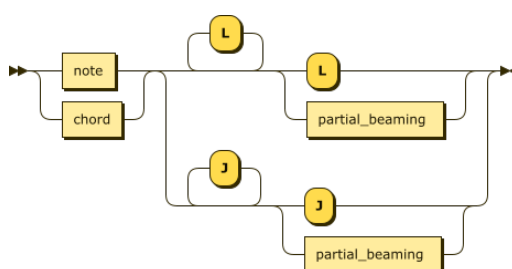


Figura 2.28: Representación de agrupación de notas.

Como observamos en la Figura 2.28, la agrupación se realiza por medio de los *tokens* L y J el primero que da comienzo a la agrupación y el último da finalidad. Puede ser repetido cada uno de estos dependiendo la agrupación que hagamos y siempre la escribimos detrás de la nota o acorde (también podemos agrupar acordes). Un ejemplo sencillo de este tipo de agrupación podría ser el siguiente:

8gL
8eJ



Figura 2.29: Representación de una corchea.

Donde agruparemos dos corcheas, un sol y un mi. Por último, podemos observar lo que es una agrupación parcial de forma que la haz que de suelta. Para representarlo, únicamente debemos deberemos añadir una k o K para que sea representada.

Silencios

La representación del silencio es sencilla y se asemeja mucho a la representación de una nota simple, a él, únicamente, debemos añadirle la medida de tiempo que tiene y el *token* que lo define es 'r'.

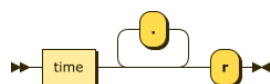


Figura 2.30: Representación del silencio.

Algo que no hemos mencionado, como podemos observar en la Figura 2.30 es que al silencio también le podemos agregar puntillos. Un ejemplo de éste sería:

8.r

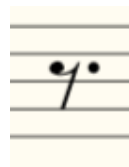


Figura 2.31: Representación de un silencio de corchea.

Este ejemplo representaría al silencio de una corchea.

Ligadura de expresión

la ligadura de expresión [25] o *slur* es una alteración musical de una serie de notas las cuales, las afectadas, se deben interpretar sin separación entre ellas. Como vemos en la Figura 2.32, la representación se basa en el uso de corchetes enumerándolos con la cantidad de necesaria para poder identificarlos, el inicio del ligadura de expresión corresponderá al *token* "(" mientras que el final a su opuesto.

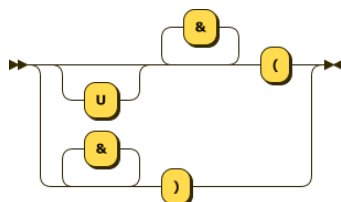


Figura 2.32: Representación de un ligadura de expresión.

Ejemplo de la Figura 2.32 sería el siguiente:

(4g
2e)



Figura 2.33: Representación de una ligadura de expresión básica.

Donde en 4g comenzaría la ligadura de expresión y concluiría con 2e ya que entre ambos aparecen los corchetes.

Ligadura

La ligadura [26] o *tie* se produce cuando pretendemos alargar dos notas, esto visualmente en la notación moderna occidental se representa con una ligadura de articulación entre dos notas de la misma tonalidad. En la notación de ****kern** se representa con la inserción de los corchetes entre dos notas.

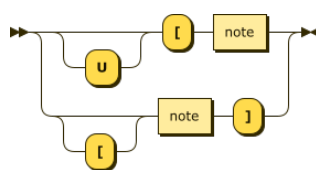


Figura 2.34: Representación de una ligadura.

Una cosa que podemos observar entre la Figura 2.32 y la Figura 2.34 es que en ambas aparece el *token* U como opcional, hemos insertado este *token* en la gramática para simbolizar dónde queremos la ligadura de expresión o la ligadura. Si representamos cualquiera de los dos con U irá en la parte de arriba de la nota, (*up*) y si no lo representamos automáticamente irá hacia abajo.

Un ejemplo de la Figura 2.34 sería el siguiente:

[4g
2g]



Figura 2.35: Representación de una ligadura.

Donde alargaremos las notas ya que la ligadura englobará a las dos notas con la misma tonalidad ya que los corchetes las engloba.

2.9.1.6. Ejemplo de ****kern**

Visto todo lo anterior, deberíamos ser capaces de entender el código en notación ****kern** ya que hemos visto las reglas más importantes para que pueda ser representada. Para tener una mejor base de la gramática con esta notación, a continuación, visualizaremos un ejemplo de la misma, donde dado una notación de entrada generaremos un árbol sintáctico el cual representará una pequeña pieza musical.

```
**kern
*clefG2
*M4/4
*met(c)
(8eL
8fJ
4g 4b 4dd)
=
*-
```



Figura 2.36: Representación musical del código ejemplo `**kern`.

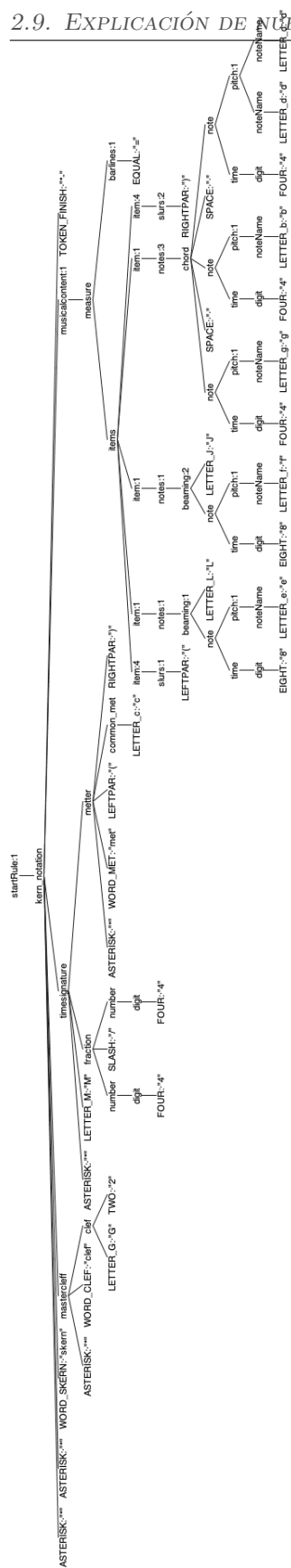


Figura 2.37: Árbol sintáctico generado.

En el ejemplo anterior observamos la generación de un árbol sintáctico a partir de un código de entrada simple. Vemos como desde un nodo inicial va generando ramificaciones a nodos no terminales y finalizando dicha ramificación con nodos terminales. De esta manera nuestro código es analizado sintácticamente obteniendo como resultado una implementación correcta de la notación ****kern**. Junto a la previa explicación de la gramática el ejemplo no debe resultar complejo.

2.9.2. ****mens**

Una vez explicada la parte de ****kern**, vamos a pasar a la notación mensural, ****mens**. Hemos de decir que esta notación tiene reglas bastante parecidas a las de ****kern** salvando algunas pequeñas estructuras o añadidos.

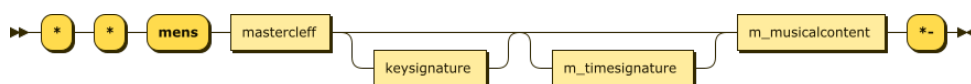


Figura 2.38: Regla principal de ****mens**

En esta figura observamos que la estructura principal de esta notación es prácticamente la misma que la principal de ****kern**, Figura 2.7, las reglas que no anteceden en su nombre un ‘m_’ y son las mismas que la otra notación.

A continuación nos saltaremos las reglas que no varían entre ****kern** y ****mens** ya que han sido explicadas previamente.

2.9.2.1. Ritmo de partitura

Como para la notación mensural no había tiempos de compás en base a la fracción sino lo que se denomina signos mensurales, hemos desechado la regla de las fracciones, Figura 2.39.



Figura 2.39: Regla de tiempo de compás.

Dejando únicamente las estructuras rítmicas [16] que para esta notación hay un abanico muy amplio de ellas.

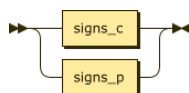


Figura 2.40: Estructura de compás.

Como podemos ver en la figura anterior, estas estructuras tienen dos tipos, *perfectas* e *imperfectas*, cada una de ellas las podemos visualizar por separado de la siguiente manera, Figura 2.41 y 2.42.

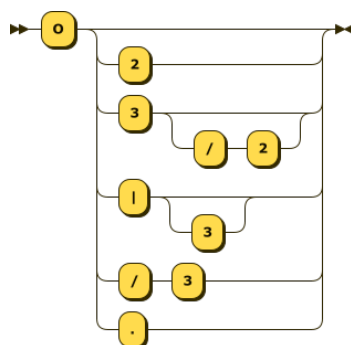


Figura 2.41: Representación de estructuras de compás perfecta.

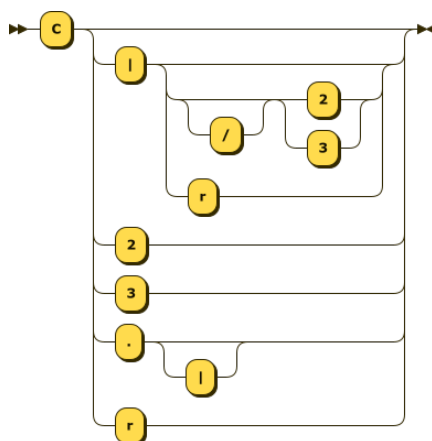


Figura 2.42: Representación de estructuras de compás común.

Para ejemplificar esta regla, un ejemplo de ésta sería:

`*met(C|)`



Figura 2.43: Representación de un signo mensural.

La diferencia con la estructura rítmica de `**kern` reside en que en `**mens` tenemos muchos más signos de métrica y que el tipo del signo se representa en mayúscula en vez de en minúscula: `C` en vez de `c`.

2.9.2.2. Contenido musical

A partir de este apartado, la diferencia de ambas notaciones se acentúan, puesto que en la notación musical no tenemos compases, las figuras son expresadas de manera diferente y tenemos otros recursos.

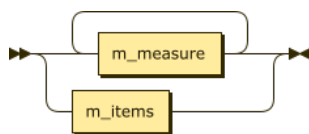


Figura 2.44: Contenido musical

En el párrafo anterior hemos dicho que en esta notación no tiene compases, pero como podemos observar en la Figura 2.44 tenemos la regla **measure**, estas medidas en este caso no representa a compases, simplemente, representa una notación para que el que implemente el código tenga una referencia, adoptando separadores (**barlines**) a un conjunto de figuras.

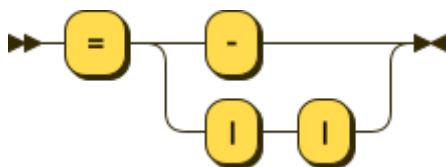


Figura 2.45: Estructura de los separadores.

Como observamos en la Figura 2.45, tenemos dos formas de indicar estos separadores, uno por medio del *token* `=-` y otro para finalizar con `=||`.

2.9.2.3. Figuras

El contenido musical se basa fundamentalmente en un conjunto de figuras musicales como representa la Figura 2.44 y cada figura es representada como indica la Figura 2.46.

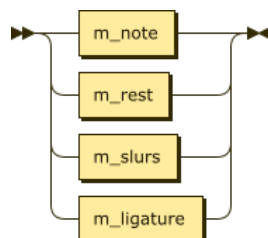


Figura 2.46: Estructura de figuras ****mens**.

En la figura anterior observamos cuatro tipos de representación de figuras, de arriba a bajo serían, las notas propias de esta notación, los silencios, la ligadura de expresión y la ligadura de la notación mensural.

Notas

Para representar las notas tenemos **a**, **b**, **c**, **d**, **e**, **f**, **g** como en la notación de ****kern**, este nos permite saber el tono de la nota. Una de las cosas que cambia ****mens** es en la representación de la duración de las notas [17] que son, de más a menos duración; **X**, **L**, **S**,

M, U, s, m, u, representado, respectivamente a la duración de *maxima*, *longa*, *breve*, *minima*, *fusa*, *semi-breve*, *semi-minim* y *semi-fusa*.

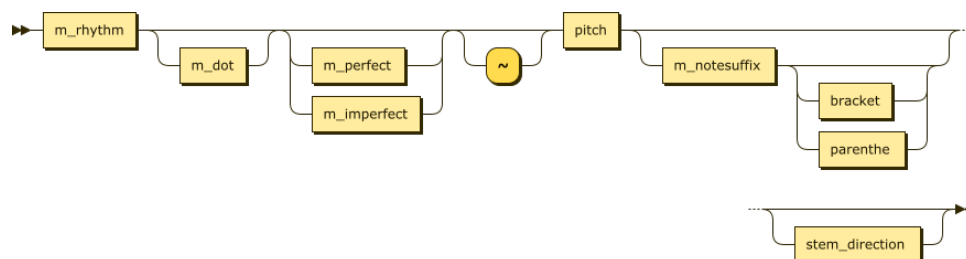


Figura 2.47: Representación de notas `**mens`.

Un ejemplo básico de una nota la cual representaría una fusa en sol sería:



Figura 2.48: Representación de una nota.

Como podemos observar, la Figura 2.47, es la representación de una nota en notación mensural. En ella podemos editar el tono, la duración (también si lleva puntillo [18]), las alteraciones que pueda poseer, incluso la dirección de la plica.

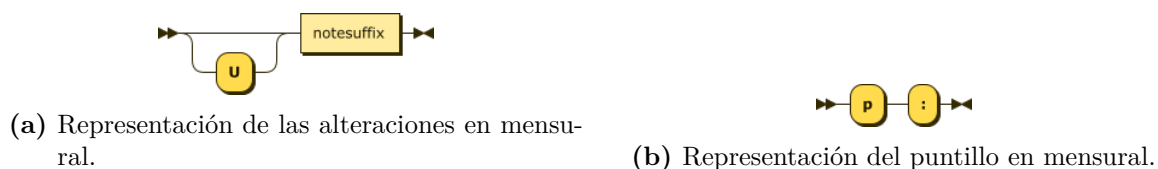


Figura 2.49: Representación de la alteración y del puntillo en mensural.

Como observamos en la Figura 2.49a podemos utilizar el *token* U para determinar si la alteración va al lado o encima de la nota, cosa que en `**kern` no teníamos disponible. Seguidamente, la alteración se representa igual que en la otra notación.

Por otro lado, el puntillo, como representa la Figura 2.49b, también varía a la notación `**kern` siendo su simbología de la siguiente manera p:



Figura 2.50: Representación de una nota con alteraciones y puntillo.

El ejemplo anterior corresponde a una fusa de **1a** sostenida con puntillo.

También podemos ver si la nota es perfecta o imperfecta [19], que en la notación moderna supone el punto de aumento si la nota es perfecta y sin punto de aumento si la nota es imperfecta, con la representación de los *tokens* **i** y **p** respectivamente.

Silencios

Al igual que ****kern**, ****mens** también posee sus propias formas de expresar los silencios como podemos observar en la Figura 2.51.

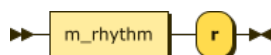


Figura 2.51: Representación de silencios en ****mens**.

Esta regla es bastante sencilla ya que para implementarla solo necesitamos la duración del silencio (**X**, **L**, **S**, **M**, **U**, **s**, **m**, **u**) y el *token* **r** que representa al silencio (*rest* en inglés).

Ejemplo de un silencio de fusa sería:



Figura 2.52: Representación de un silencio de fusa.

Ligadura de expresión

La ligadura de expresión o *slur* en inglés representa a la continuidad de las notas entre ellas. Como se muestra en la Figura 2.53 la diferencia es muy pequeña respecto a la notación de ****kern**.

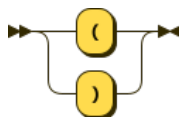


Figura 2.53: Representación del ligadura de expresión en ****mens**.

Las ligaduras de expresión abarcarán un grupo de figuras con paréntesis que podremos enumerar con el símbolo **.** Tiene el mismo formato que en la notación de ****kern**, luego ya sabremos como implementarlos para ****mens**

Ligature

La *ligature* son representadas con corchetes, éstas agrupan todas las notas que abarquen.

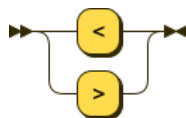


Figura 2.54: Representación de la ligadura en ****mens**.

Como observamos en la Figura 2.54, la estructura de la *ligature* es un corchete de apertura "[" y y el corchete derecho "]" para cerrar la ligadura entre ellos habrá diferentes notas que estarán ligadas.

La diferencia que reside entre la *ligature* y la ligadura de expresión, es que la primera trata de representar patrones fijos de notas cortas y largas, agrupando las notas de la misma forma que los pies métricos se utilizan para unir sílabas cortas y largas, mientras que la ligadura de expresión une las notas para que hacer un sonido fluido y sin descansos entre medio.

Una representación de una *ligature* sería:



Figura 2.55: Representación de una *ligature*.

2.9.2.4. Ejemplo de ****mens**

Dado las anteriores directrices deberíamos ser capaces de poder implementar por nosotros mismos la gramática con notación ****mens**. A continuación, podremos ver un ejemplo de la implementación de un pequeño ejemplo, con su representación gráfica y seguidamente, la visualización del árbol sintáctico que éste genera.

```
**mens
*clefG2
La
Sg
Ua
*—
```

Tabla 2.3: Código de entrada ****mens**

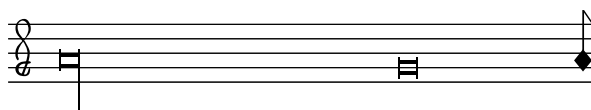


Figura 2.56: Representación musical del código ejemplo ****mens**.

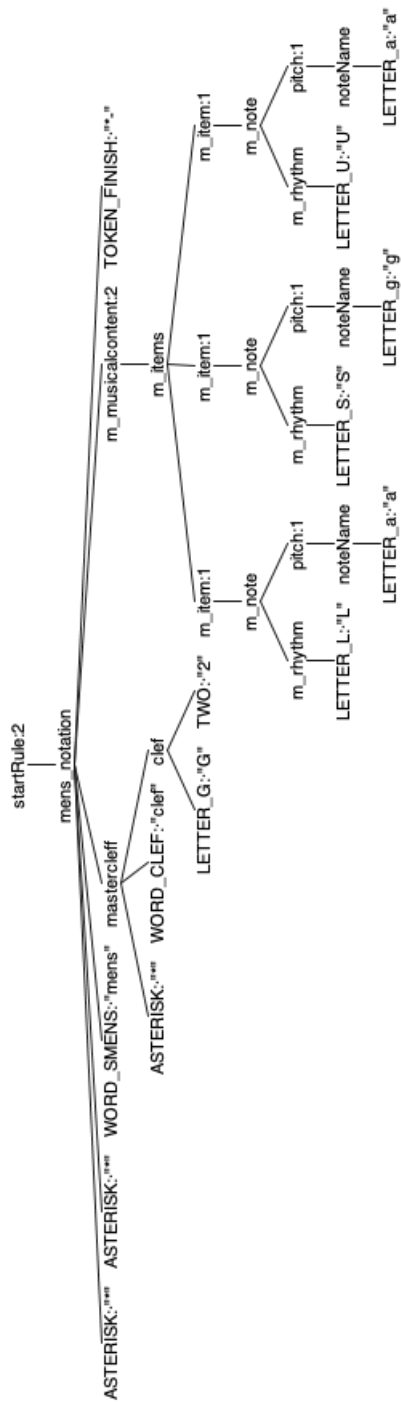


Figura 2.57: Árbol sintáctico generado de **mens.

3. Pruebas Unitarias

3.1. Pruebas de caja blanca

Las pruebas de caja blanca, son un conjunto de pruebas unitarias que se centran en el apartado del software y no en la especificación. La creación de nuestras pruebas consistirán en un análisis de nuestro código en el que se va escogiendo diferentes valores de entrada para obtener una salida valida. En el caso de ser así la prueba se habrá realizado con éxito.

Para nuestra metodología a tratar, este estilo de pruebas es el que necesitaremos ya que conforme vayamos creando las pruebas iremos modificando el código hasta lograr una gramática aplicable a `**kern` o `**mens`.

3.1.1. Creación de pruebas

El entorno de desarrollo integrado (IDE) en el que hemos realizado el trabajo ha sido *IntelliJ*, además ha sido donde hemos realizado también las pruebas. Hemos elegido este entorno por su facilidad de uso, ya que se complementa fácilmente con otras herramientas como es ANTLR y Maven. Maven es una herramienta de software, ésta es utilizada para la gestión de proyectos en Java y la hemos utilizado junto a JUnit. Por su parte, JUnit es una *framework* o librería que permite controlar ejecuciones de clases Java y es utilizada para implementar test de evaluación de diferentes métodos para verificar que se comporta de la manera correspondiente (pruebas unitarias).

Para la creación de las pruebas IntelliJ dispone de una plugin de ANTLR en la que con una gramática y un léxico, dado un código de entrada te puede decir si dicho código es válido para la gramática a la vez va generando un árbol sintáctico con el código de entrada. En caso de haber un fallo el nodo del árbol cambiará de color indicando el error. En primera instancia, ésta era la primera forma de testear si el código que dabamos de entrada correspondía a nuestra gramática con la notación que queríamos, si no era así, sencillamente cambiabamos la grámatica para que sea aceptada. Pero el coste de tiempo que iba a conllevar este método era excesivo, así que, tratamos de automatizar este apartado haciendo que, dado una secuencia de ficheros de entrada vaya ratificando si el código estaba bien escrito. Después de haber obtenido resultados satisfactorios con las pruebas, pasamos el código para generar la partitura a la herramienta de Verovio [28], que es una herramienta que genera a partir de un código de entrada de `**kern` y `**mens`.

3.1.1.1. Implementación de pruebas

Como hemos dicho anteriormente, nuestras pruebas se van realizando a partir de una secuencia de ficheros de entrada que tenemos en el archivo `SkmImporterText` que consta de un único método Java `void importSKernMens()`. Aquí introduciremos los ficheros que nosotros queramos probar.

```

1
2 @Test
3 public void importSKernMens() throws IM4Exception, FileNotFoundException {
4     String [] testFileNames = {
5         "notas.skm",
6         "silencios.skm",
7     };
8     for (String testFileName: testFileNames) {
9         SkmSyntaxDirectedTranslation skmSyntaxDirectedTranslation = new ↵
10             ↵ SkmSyntaxDirectedTranslation();
11         File file = TestFileUtils.getFile("/testdata/io/skm/" + testFileName);
12         SemanticComposition imported = skmSyntaxDirectedTranslation.importSkm(↵
13             ↵ file);
14     }
15 }

```

Como observamos en este código `testFileNames` va agrupando los archivos que queremos probar, en este caso, serían los archivos *notas.skm* y *silencios.skm*. Una vez agrupados entramos en el bucle `for` y vamos pasando uno a uno para que sean probados. Lo siguiente que realizamos es generar la ruta y verificar si la encontramos en `importSkm(file)`; y poder analizar cada código de entrada a partir del siguiente método el cual devuelve un objeto con la composición semántica de nuestra gramática.

```

1 private SemanticComposition importSkm(CharStream input, String ↵
2     ↵ inputDescription) throws IM4Exception {
3     ErrorListener errorListener = new ErrorListener();
4     try {
5         Logger.getLogger(SkmSyntaxDirectedTranslation.class.getName()).log(Level ↵
6             ↵ .INFO, "Parsing {0}", inputDescription);
7
8         lexerkernmens lexer = new lexerkernmens(input);
9
10        if (debug) {
11            new ANTLRUutils().printLexer(lexer);
12        }
13
14        lexer.addErrorListener(errorListener);
15        CommonTokenStream tokens = new CommonTokenStream(lexer);
16        kernmensParser parser = new kernmensParser(tokens);
17        parser.addErrorListener(errorListener);
18
19        ParseTree tree = parser.startRule();
20        ParseTreeWalker walker = new ParseTreeWalker();
21        Loader loader = new Loader(parser, debug);
22        walker.walk(loader, tree);
23        if (errorListener.getNumberErrorsFound() != 0) {
24            throw new IM4Exception(errorListener.getNumberErrorsFound() + " ↵
25                ↵ errors found in "
26                + inputDescription + "\n" + errorListener.toString());
27        }
28    } catch (Exception e) {
29        throw new IM4Exception(e.getMessage());
30    }
31 }

```

```

24     }
25
26     return loader.semanticComposition;
27 } catch (Throwable e) {
28     e.printStackTrace();
29     Logger.getLogger(SkmSyntaxDirectedTranslation.class.getName()).log(Level
    ↪ .WARNING, "Import error {0}", e.getMessage());
30     for (ParseError pe : errorListener.getErrors()) {
31         Logger.getLogger(SkmSyntaxDirectedTranslation.class.getName()).log(
    ↪ Level.WARNING, "Parse error: {0}", pe.toString());
32     }
33
34     throw new IM4Exception(e.getMessage());
35 }
36
37 }

```

En la línea 6, observamos cómo insertamos nuestro código de entrada al léxico (**lexerkernmens**) viendo si tiene algún error (línea 12) y creando los *tokens* como observamos en la siguiente línea. En la línea 14 atribuimos los *tokens* de nuestro léxico con la lectura del código de entrada previamente mencionado a la gramática y creamos el árbol dándole la regla inicial de nuestra sintaxis `ParseTree tree = parser.startRule();`. Finalmente éste es analizado y agrupa los errores en el caso de que los haya y los muestra.

Una vez realizada cada prueba sin fallos, pasamos a generar la partitura a partir del código de entrada, para realizar esta acción utilizamos la herramienta de Verovio en la que en su Github¹ nos da las instrucciones que debemos seguir para instalarla y generar las partituras. Como sabemos que vamos a tener gran cantidad de pruebas hemos realizado un script para generar de cada código su representación con el nombre de `generate_figures_test.sh`

```

1#!/bin/bash
2INPUTFOLDER="../../src/test/resources/testdata/io/skm/skern"
3OUTPUTFOLDER="../../latex/figures_tests"
4PDFs="${OUTPUTFOLDER}/pdf/skern"
5SVGs="${OUTPUTFOLDER}/svg/skern"
6VEROVIO="verovio"
7BATIK="batik-1.12"
8
9for skm in $(find $INPUTFOLDER -name *.skm -print); do
10echo "Processing ${skm}"
11
12filename=$(basename $skm .skm)
13svg="${OUTPUTFOLDER}/svg/skern/${filename}.svg"
14
15# change **skm for **kern so that Verovio can process it
16cat ${skm} | sed 's/skern/kern/g' > /tmp/kern
17
18${VEROVIO} --adjust-page-width --adjust-page-height -o ${svg} /tmp/kern
19done

```

¹<https://github.com/rism-ch/verovio/wiki>

```

20
21 java -cp ${BATIK} -jar "${BATIK}/batik-rasterizer-1.12.jar" -m application/pdf ↵
    ↵ -d ${PDFs} ${SVGs}

```

Este código trata de coger todos los ficheros con extensión `.skm` que son los códigos de entrada de nuestras pruebas e insertamos el comando de verovio que convierte el código a partitura (línea 18) para cada uno de nuestros ficheros, generando así la representación en formato `.sgv` y la última línea crea de ese formato una copia pero con el formato `.pdf`.

3.1.1.2. Ejemplo de prueba

```

**kern
*clefG2
*met(c)
4a
4b
4c
4d
=
(8eL
8fJ
4g 4b 4dd)
=
*_

```



Tabla (3.1) Código de entrada.

(a) Representación código de entrada.

Figura 3.1: Ejemplo de prueba realizada.

Como podemos observar, en la Figura 3.1 tenemos una prueba unitaria realizada en la que dado un código de entrada se genera su figura correspondiente. En caso de que nuestra prueba no fuera satisfactoria nos saldría un error diciendo el *token* de entrada es diferente al *token* que esperaba. Por ejemplo, en el código anterior en vez de tener un `*clefG2` nos hubieramos equivocado y hubieramos puesto un `*clefN2` al compilar las pruebas nos saltaría el siguiente error en terminal:

```

(2, 5):token recognition error at: 'N'
(2, 6):mismatched input '2' expecting {'C', 'F', 'G'}
(3, 1):mismatched input 'M' expecting '*'
(4, 1):mismatched input 'met' expecting '*'

```

Figura 3.2: Mensaje error al pasar una prueba

En la figura anterior nos muestra el que no esperaba el *token* N y seguido nos dice los *tokens* que esperaba.

4. Conclusiones

4.1. Posibles proyectos futuros

La realización de esta gramática supone un inicio para posibles proyectos futuros en los que se puede ver involucrada ésta.

Nuestra gramática representa gran parte de la notación de ****kern** y ****mens** pero hay algunos puntos que ésta no comprende, como puede ser el uso de más de una voz. Una de las cosas que podríamos hacer es la conversión de código a horizontal para que la regla pueda ser representada.

Además, esta gramática podría generalizar la notación de ****kern** y ****mens** en un único lenguaje en el que dado un código de entrada independiente de la notación que vayamos a usar se pueda identificar si es ****kern** o ****mens** y si éste está bien implementado. Habría que añadir un analizador semántico (*parser*) que corroborara las partes de código que no puede satisfacer en su totalidad nuestra sintaxis, como las ligaduras o agrupación de cocheas, por ejemplo.

Con esto podríamos implementar un editor con detección de errores, en el que cuando algún campo esté mal escrito se notificara.

Otro aspecto a destacar es el que este proyecto podría evolucionar en un futuro es el de ejercer de *piedra rosetta* o piedra angular, en el que en un principio era la intención de éste, para otras notaciones, ya que podría ejercer el papel de traductor interno para otras notaciones. Dado un código de entrada en una notación musical se produciría una traducción a nuestra gramática y ésta la traduciría a un código destino que eligiéramos.

Un último punto es el que este proyecto podría desarrollarse como apoyo a la inteligencia artificial OMR¹ la cual, como su nombre indica reconoce partituras musicales a través de imágenes. Sistematizando nuestro lenguaje podría conseguir un mejor reconocimiento.

4.2. Conclusiones

En definitiva, este trabajo ha tratado sobre la realización de una gramática para la notación de ****kern** y ****mens** con la posibilidad de que en un futuro pueda ser unificada en un código único, sabiendo separar por sí mismo que tipo de código entra y poder representarlo como es debido.

Además nuestra gramática ha sido verificada por pruebas basadas en el software. Hemos realizado las pruebas necesarias para que nuestra gramática acepte los símbolos dados por ****kern** y ****mens**. Ha habido reglas que no se pueden corroborar únicamente basadas en la gramática. Para la verificación total de ésta necesitaríamos un analizador semántico.

¹Optical Music Recognition

4.2.1. Código fuente del trabajo.

El trabajo está accesible en el repositorio de GitHub, <https://github.com/jorco91/Gramatica-kern-mens>. En el podemos encontrar la parte del léxico y de la sintaxis bajo los nombres `lexerkernmens.g4` y `kernmensParser.g4` en la ruta `src/main/antlr4/jorge/`.

En el directorio `resources` de la parte de `test` se encuentran los códigos de entrada con los que hemos realizado las diferentes pruebas. Mientras que la implementación de las pruebas las encontramos en la carpeta `test` dentro del paquete `Java`.

Finalmente, en la carpeta `memoria` tendremos la documentación que estamos leyendo ahora, con los ficheros para generar las representaciones musicales o diagramas.

Bibliografía

- [1] Herramienta generadora de analizadores léxicos desarrollada por Robert P. Corbett alrededor de 1987. <https://github.com/westes/flex>
- [2] Herramienta generadora de analizadores sintácticos creada por Robert P. Corbett en junio de 1985. <http://www.gnu.org/software/bison/>
- [3] Terrence Par (2012), *The Definitive ANTLR 4 Reference*. The Pragmatic Programmers, LLC.
- [4] Notación musical creada por David Huron en la década de 1980. <https://www.humdrum.org/guide/ch02/>
- [5] Rizo, D., Pascual-León, N., Sapp, C.S. (2018). White Mensural Manual Encoding: from Humdrum to MEI, *Cuadernos de Investigación Musical*, 6, 373-393.
- [6] Digital Alternate Representation of Musical Scores creado en 1966 por Stefan Bauer-Mengelberg.
- [7] Sistema de representación musical creado en 1971 por Leland Smith.
- [8] Sistema de representación desarrollado para crear música monofónica a finales de los 80s por Helmut Schaffrath.
- [9] Página de Humdrum. <https://www.humdrum.org>
- [10] Documentación de la representación de ****kern**. <https://www.humdrum.org/rep/kern/>
- [11] Documentación de las representaciones de la clave. https://doc.verovio.humdrum.org/humdrum/getting_started/#clefs
- [12] Documentación de las representaciones de la armadura de compás. https://doc.verovio.humdrum.org/humdrum/getting_started/#key-signatures
- [13] Documentación de las representaciones del tiempo del compás. https://doc.verovio.humdrum.org/humdrum/getting_started/#time-signatures
- [14] Documentación de las estructuras de compás. https://doc.verovio.humdrum.org/humdrum/getting_started/#meter-symbols
- [15] Documentación general de la notación. https://doc.verovio.humdrum.org/humdrum/getting_started

- [16] Documentación para estructura rítmica de notación mensural.
<http://doc.verovio.humdrum.org/humdrum/mens/#mensural-signs>
 - [17] Documentación para estructura rítmica de una nota en mensural.
<https://doc.verovio.humdrum.org/humdrum/mens/rhythm>
 - [18] Documentación del puntillo en notación mensural.
<https://doc.verovio.humdrum.org/humdrum/mens/dots>
 - [19] Documentación para una nota perfecta e imperfecta.
<http://doc.verovio.humdrum.org/humdrum/mens/#perfection>
 - [20] Documentación para el tono de una nota.
https://doc.verovio.humdrum.org/humdrum/getting_started/#pitch
 - [21] Documentación de la implementación de la duración de una nota.
https://doc.verovio.humdrum.org/humdrum/getting_started/#rhythm
 - [22] Documentación de la implementación del punto de aumento.
https://doc.verovio.humdrum.org/humdrum/getting_started/#augmentation-dots
 - [23] Documentación de la implementación de accidentes ****kern**.
https://doc.verovio.humdrum.org/humdrum/getting_started/#accidentals
 - [24] Documentación de la implementación de una octava.
https://doc.verovio.humdrum.org/humdrum/getting_started/#octaves
 - [25] Documentación de la implementación del ligado.
https://doc.verovio.humdrum.org/humdrum/getting_started/#slurs
 - [26] Documentación de la implementación de la ligadura.
https://doc.verovio.humdrum.org/humdrum/getting_started/#ties
 - [27] Jorge Calvo Zaragoza, David Rizo (2018). End-to-End Neural Optical Music Recognition of Monophonic Scores, Applied Sciences, 6, 606. doi:10.3390/app8040606
 - [28] Herramienta de transcripción musical.
<https://www.verovio.org/index.xhtml>
-

A. Anexo I

A.1. Introducción

Esta sección ofrece la visualización de las pruebas realizadas con éxito que han dado pie a la gramática final. El contenido de las pruebas son de la notación `**kern` y `**mens`, siguiendo a la notación podremos visualizar la partitura generada.

A.2. `**kern`

A.2.1. Alteraciones

```
**kern
*clefG2
4c--
4d-
4en
4f#
4g##
*-
```



Figura A.1: Prueba 1 de alteraciones.

```
**kern
*clefG2
4c--I
4d-I
4en
4f#P
4g##P
*-
```



Figura A.2: Prueba 2 de alteraciones.

A.2.2. Articulaciones

```
**kern
*clefG2
*M4/4
4a
4a'
*-
```



Figura A.3: Prueba 1 de articulaciones.

```
**kern
*clefG2
*M4/4
4a
4a^
*-
```



Figura A.4: Prueba 2 de alteraciones.

```
**kern
*clefG2
*M4/4
4a
4a`
*-
```



Figura A.5: Prueba 3 de articulaciones.

```
**kern
*clefG2
*M4/4
4a
4a~
*-
```



Figura A.6: Prueba 4 de articulaciones.

```

**kern
*clefG2
*M4/4
4a
4a'^
*-

```



Figura A.7: Prueba 5 de articulaciones.

```

**kern
*clefG2
*M4/4
4a
4a^^
*-

```



Figura A.8: Prueba 6 de articulaciones.

```

**kern
*clefG2
*M4/4
4a
4a;
*-

```



Figura A.9: Prueba 7 de articulaciones.

```

**kern
*clefG2
*M4/4
4a
4a'~
*-

```



Figura A.10: Prueba 8 de articulaciones.

A.2.3. Punto de aumentación

```

**kern
*clefG2
4.b
4..b
4...b
4....b
4.....b
=-
4.....b
4.....b
4.....b
4.....b
4.....b
==
*-

```



Figura A.11: Prueba de punto de aumentación.

A.2.4. Barra de compás

```

**kern
*clefG2
=1
1c
=2
1d
==
*-

```



Figura A.12: Prueba 1 de barra de compás.


```
**kern  
*clefG2  
=1  
1c  
=2-  
1d  
=3||  
1e  
=4  
1f  
=5!|:  
1g  
=6:|!|:  
1a  
=7:|!  
1b  
==  
*-
```



Figura A.13: Prueba 2 de barra de compás.

A.2.5. Corcheas

```
**kern  
*clefG2  
*M3/4  
4aL  
4aJ  
*-
```

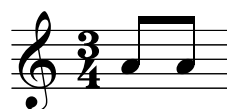


Figura A.14: Prueba 1 de corcheas.

```

**kern
*clefG2
*M3/4
4aL
8aL
8a
16aL
16aJJ
4aJ
*-

```



Figura A.15: Prueba 2 de corcheas.

```

**kern
*clefG2
*M3/4
4aL
8aL
8aJ
4aJ
*-

```



Figura A.16: Prueba 3 de corcheas.

```

**kern
*clefG2
*M2/4
16.eLL
32fk
16.e
32fJJk
8fL
16gL
16aJJ
==
*-

```



Figura A.17: Prueba de corcheas parciales.

A.2.6. Estructura de tiempo de compás

```
**kern  
*clefG2  
*M4/4  
*met(c)  
=1  
2g  
4a  
4b  
==  
*-
```

**Figura A.18:** Prueba 1 de estructura de compás.

```
**kern  
*clefG2  
*M4/4  
*met(c|)  
=1  
2g  
4a  
4b  
==  
*-
```

**Figura A.19:** Prueba 2 de estructura de compás.

A.2.7. Armadura

```

**kern
*clefG2
*k[f#]
=1
4g
4f
4e
4f
=
*-

```



Figura A.20: Prueba 1 de la armadura de compás.

```

**kern
*clefG2
*k[d#e#f#g#]
=1
4g
4f
4e
4f
=
*-

```



Figura A.21: Prueba 2 de la armadura de compás.

```

**kern
*clefG2
*k[f#a-]
=1
4g
4f
4e
4f
=
*-

```



Figura A.22: Prueba 3 de la armadura de compás.

A.2.8. Notas

```

**kern
*clefG2
*M4/4
4a
4b
4c
4d
4e
4f
4g
*-

```

**Figura A.23:** Prueba 1 de notas.

```

**kern
*clefG2
*M4/4
4...AA--M'~\
*-

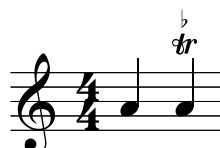
```

**Figura A.24:** Prueba 1 de notas.**A.2.9. Ornamentos**

```

**kern
*clefG2
*M4/4
4a
4at
*-

```

**Figura A.25:** Prueba 1 de ornamentos.

```

**kern
*clefG2
*M4/4
4a
4aT
*-

```

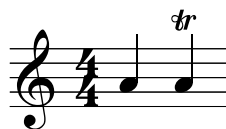


Figura A.26: Prueba 2 de barra de ornamentos.

```

**kern
*clefG2
*M4/4
4a
4am
*-

```



Figura A.27: Prueba 3 de ornamentos.

```

**kern
*clefG2
*M4/4
4a
4aM
*-

```

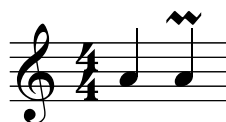


Figura A.28: Prueba 4 de barra de ornamentos.

```

**kern
*clefG2
*M4/4
4a
4am
*-

```



Figura A.29: Prueba 5 de barra de ornamentos.

```

**kern
*clefG2
*M4/4
4a
4aW
*-

```

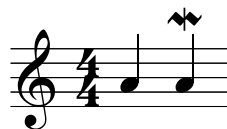


Figura A.30: Prueba 6 de ornamentos.

```

**kern
*clefG2
*M4/4
4a
4aw
*-

```

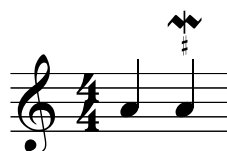


Figura A.31: Prueba 7 de ornamentos.

A.2.10. Silencios

```

**kern
*clefG2
*M5/4
=1
4r
2.r
32r
8r
16r
32r
=2
*M2/1
0r
==
*-

```

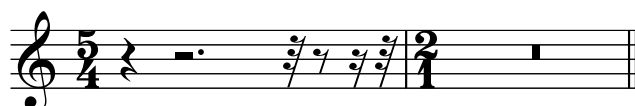


Figura A.32: Prueba de silencio.

A.2.11. Ritmo

```
**kern
*clefG2
000b
00b
0b
1b
2b
4b
8b
16b
32b
64b
128b
256b
==
*-
```

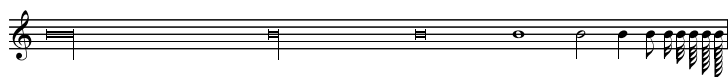


Figura A.33: Prueba 1 de tiempo de la nota.

```
**kern
*clefG2
*M4/4
00a
0a
1a
2a
4a
8a
16a
32a
64a
124a
*-
```

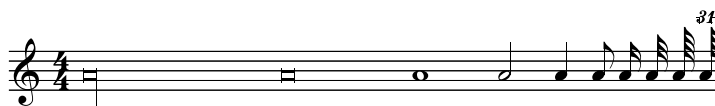


Figura A.34: Prueba 2 de tiempo de la nota.

A.2.12. Ligado

```

**kern
*clefG2
*M4/4
=1
(4c
4d
4e
4f
=2
4g
4f
4e
4c)
==
*-

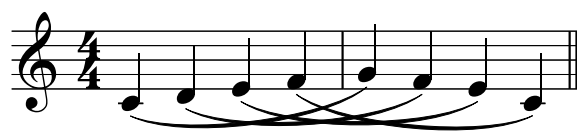
```

**Figura A.35:** Prueba 1 del ligado.

```

**kern
*clefG2
*M4/4
=1
(4c
&(4d
&&(4e
&&&(4f
=2
4g)
4f&)
4e&&)
4c&&&)
==
*-

```

**Figura A.36:** Prueba 2 del ligado.

A.2.13. Dirección de la plica

```

**kern
*clefG2
*M4/4
4a
4a/
*-

```



Figura A.37: Prueba 1 de la dirección de la plica.

```

**kern
*clefG2
*M4/4
4a
4a\
*-

```



Figura A.38: Prueba 2 de la dirección de la plica.

A.2.14. Ligadura

```

**kern
*clefG2
*M2/4
[4c
=1
4.c]
==
*-

```



Figura A.39: Prueba 1 de ligadura.

```

**kern
*clefG2
*M2/4
[4c
=1
[4.c]
4c]
==
*-

```

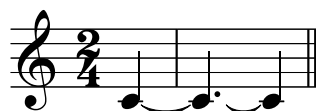


Figura A.40: Prueba 2 de ligadura.

```

**kern
*clefG2
*M2/4
[4c
=1
4d] [4a
4a[
=2
2a
==
*-

```



Figura A.41: Prueba 3 de ligadura.

A.2.15. Octavas

```

**kern
*clefF4
2ccc
2cc
2c
2C
2CC
2CCC

```

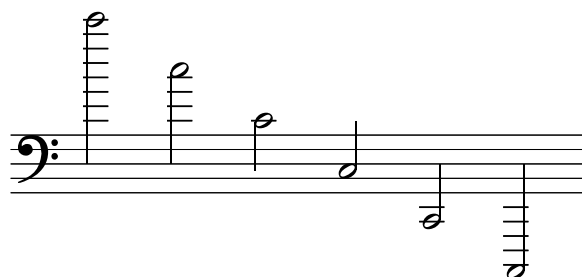


Figura A.42: Prueba 1 de octavas.

A.3.4. Ligature

**mens
*clefG2
<Ma
Mg
Ma
Mb
Mb>
*-

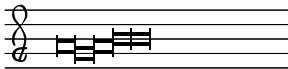


Figura A.48: Prueba de ligature.

A.3.5. Estructura de ritmo

A.3.5.1. Común

**mens
*clefG2
*met(C)
La
*-

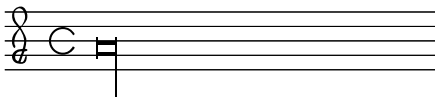


Figura A.49: Prueba 1 de estructura de ritmo común.

**mens
*clefG2
*met(C)
La
*-

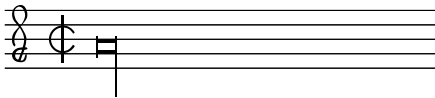


Figura A.50: Prueba 2 de estructura de ritmo común.

```
**mens
*clefG2
*met(C|3)
La
Sg
Ua
*-
```

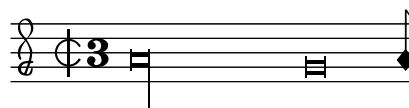


Figura A.51: Prueba 3 de estructura de ritmo común.

```
**mens
*clefG2
*met(C|2)
La
Sg
Ua
*-
```

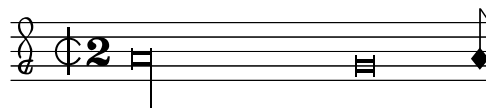


Figura A.52: Prueba 4 de estructura de ritmo común.

```
**mens
*clefG2
*met(C|/2)
La
Sg
Ua
*-
```

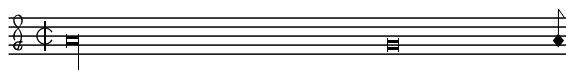


Figura A.53: Prueba 5 de estructura de ritmo común.

```
**mens
*clefG2
*met(C|/3)
La
Sg
Ua
*-
```

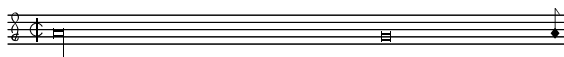


Figura A.54: Prueba 6 de estructura de ritmo común.

```

**mens
*clefG2
*met(C|r)
La
Sg
Ua
*-

```

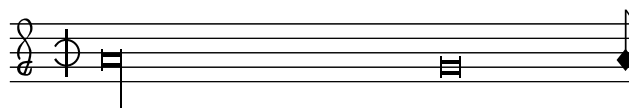


Figura A.55: Prueba 7 de estructura de ritmo común.

```

**mens
*clefG2
*met(C2)
La
Sg
Ua
*-

```

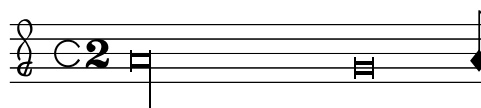


Figura A.56: Prueba 8 de estructura de ritmo común.

```

**mens
*clefG2
*met(C3)
La
Sg
Ua
*-

```



Figura A.57: Prueba 9 de estructura de ritmo común.

```

**mens
*clefG2
*met(C.)
La
Sg
Ua
*-

```

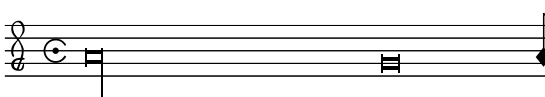


Figura A.58: Prueba 10 de estructura de ritmo común.


```

**mens
*clefG2
*met(C.|)
La
Sg
Ua
*-

```

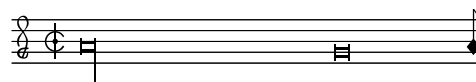


Figura A.59: Prueba 11 de estructura de ritmo común

A.3.5.2. Perfecta

```

**mens
*clefG2
*met(0)
La
*-

```

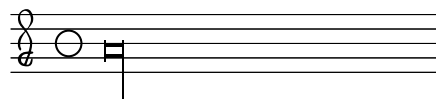


Figura A.60: Prueba 1 de estructura de ritmo perfecto.

```

**mens
*clefG2
*met(02)
La
*-

```



Figura A.61: Prueba 2 de estructura de ritmo perfecto.

```

**mens
*clefG2
*met(03)
La
*-

```



Figura A.62: Prueba 3 de estructura de ritmo perfecto.

```

**mens
*clefG2
*met(0|)
La
*-

```

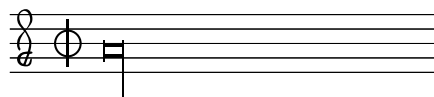


Figura A.63: Prueba 4 de estructura de ritmo perfecto.

```

**mens
*clefG2
*met(0|3)
La
*-

```



Figura A.64: Prueba 5 de estructura de ritmo perfecto.

```

**mens
*clefG2
*met(0/3)
La
*-

```

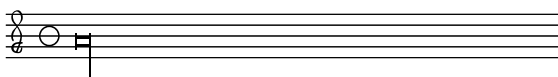


Figura A.65: Prueba 6 de estructura de ritmo perfecto.

```

**mens
*clefG2
*met(0.)
La
*-

```

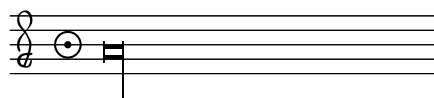


Figura A.66: Prueba 6 de estructura de ritmo perfecto.

```

**mens
*clefG2
*met(03/2)
La
*-

```

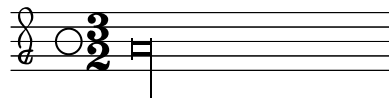


Figura A.67: Prueba 8 de estructura de ritmo perfecto.

```

**mens
*clefG2
*met(03)
La
*-

```



Figura A.68: Prueba 9 de estructura de ritmo perfecta.

```

**mens
*clefG2
*met(02)
La
*-

```



Figura A.69: Prueba 10 de estructura de ritmo perfecta.

A.3.6. Notas

```

**mens
*clefG2
Up:i~a- -
Up:pb
U~c#
Ud
Uie
*-

```



Figura A.70: Prueba de notas.

A.3.7. Alteración

**mens
*clefG2
La-
Lg- -
La##
Lb#
Lbn
*-

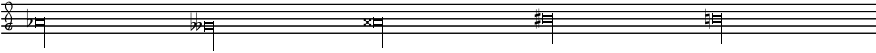


Figura A.71: Prueba de alteración de la nota.

A.3.8. Silencios

**mens
*clefG2
Xr
Lr
Sr
Mr
Ur
sr
mr
ur
*-

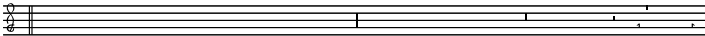



Figura A.72: Prueba de silencio.

A.3.9. Tiempo de la nota

**mens
*clefG2
Xa
La
Sa
Ma
Ua
sa
ma
ua
*-



The musical staff shows a sequence of notes corresponding to the lyrics in the table. The notes are: Xa (half note), La (half note), Sa (half note), Ma (half note), Ua (half note), sa (half note), ma (half note), ua (half note), and *- (half note). The staff is in G2 clef.

Figura A.73: Prueba de tiempo de las notas.