

**UNIVERSIDAD DE MÁLAGA**

**ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA**

**INGENIERA EN INFORMÁTICA**

**Sib – SI bémol, Lenguaje de programación musical**

**Realizado por  
Antonio Blanco Oliva**

**Dirigido por  
José Francisco Chicano García**

**Departamento de Lenguajes y Ciencias de la Computación**

**MÁLAGA, Diciembre 2017**



**UNIVERSIDAD DE MÁLAGA**  
**ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA**  
**INGENIERÍA INFORMÁTICA**

Reunido el tribunal examinador en el día de la fecha, constituido por:

Presidente/a Dº/Dª. \_\_\_\_\_

Secretario/a Dº/Dª. \_\_\_\_\_

Vocal Dº/Dª. \_\_\_\_\_

para juzgar el proyecto Fin de Carrera titulado: **Sib – SI bémol, Lenguaje de programación musical**

realizado por Dº. Antonio Blanco Oliva

tutorizado por Dº. José Francisco Chicano García,

y, en su caso, dirigido académicamente por

Dº/Dª. \_\_\_\_\_

ACORDÓ POR \_\_\_\_\_ OTORGAR LA CALIFICACIÓN  
DE \_\_\_\_\_

Y PARA QUE CONSTE, SE EXTIENDE FIRMADA POR LOS COMPARECIENTES  
DEL TRIBUNAL, LA PRESENTE DILIGENCIA.

Málaga a \_\_\_\_ de \_\_\_\_\_ del 201\_\_



**Agradecimientos:**

A María José, mi mujer, por estar ahí día a día y hacer posible lo imposible. Y a mi pequeña Leire, por darme el último empujón.

A mi tía Isabel, por animarme siempre a seguir, incluso en los peores momentos.

Y por supuesto a mis abuelos Teresa y Antonio, sin los cuales esta aventura nunca hubiese empezado, y por los que tiene sentido acabarla.

# Contenido

Capítulo 1. Introducción.....	8
1.1 Antecedentes.....	8
1.2 Objetivos.....	8
1.3 Organización de la memoria.....	9
Capítulo 2. Sib.....	10
2.1 Introducción.....	10
2.2 Características generales.....	11
2.3 Estructura general.....	12
2.4 Cuerpo del programa.....	14
2.5 Operadores básicos.....	21
2.5.1 Comparación.....	21
2.5.2 Operadores aritméticos.....	22
2.6 Tipo de datos.....	25
2.6.1 Tipos de datos básicos.....	26
2.6.2 Tipos de datos compuestos.....	28
2.7 Variables.....	31
2.8 Mejoras y versiones futuras.....	32
2.9 Ejemplos Sib.....	37
2.9.1 Ejemplo escala DO mayor.....	37
2.9.2 Trasportar partitura.....	39
Capítulo 3. Traductor Sib a MusicXML.....	40
3.1 Introducción.....	41
3.2 Fase de Análisis.....	42
3.2.1 Análisis Léxico.....	43
3.2.2 Análisis Sintáctico.....	45
3.2.3 Análisis semántico.....	46
3.3 Fase de Síntesis.....	47
3.3.1 Generación de código MusicXML.....	48
3.4 Sib IDE.....	49
3.5 Futuras versiones.....	50

Capítulo 4. Gramática.....	51
4.1 Introducción.....	51
4.2 Definición de la gramática Sib.....	53
Capítulo 5. MusicXML.....	61
Capítulo 6. Conclusiones y mejoras.....	65
Apéndice A: Introducción al lenguaje musical.....	66
Apéndice B: JFlex & Java CUP.....	70
B.1 JFlex.....	70
B.2 Java CUP.....	73
Apéndice C: Entorno de desarrollo.....	76
Apéndice D: Código entregado.....	78
Referencias.....	79

# Capítulo 1. Introducción

En este primer capítulo se va hacer una breve introducción al proyecto, indicando las motivaciones que llevó a proponerlo, los objetivos que se pretenden alcanzar con él, y un breve descripción de la organización de la memoria.

## *1.1 Antecedentes*

La idea surge como muchas de las ideas, ante una necesidad personal.

Hace un par de años me inicié en el mundo de la música a nivel amateur, dando clases de solfeo y empezando a tocar el saxofón tenor. En las clases era el único tenor, ya que el resto de alumnos tocaban el saxofón alto, por lo que a veces tenía problemas con las partituras, ya que sólo disponíamos de las de saxofón alto. La solución era esperar a que el profesor “transportara” la partitura dos tonos y medio más aguda, para yo poder tocar con el resto de compañeros la misma pieza.

Como informático, uno piensa: “Eso lo soluciono yo con un WHILE que lea nota a nota y las vaya transportando”

Aunque la música es un mundo creativo, existen ciertas tareas monótonas y que cumplen ciertas normas susceptibles de ser implementadas.

“Unamos ambos mundos, y pongamos todo el potencial de un lenguaje de programación al alcance creativo de los compositores, creemos Sib (SI bemol)”

Una vez detectado el problema y analizando la solución, ¿existen otros lenguajes musicales en el mercado? La respuesta es si, existen otros como Chuck, de la Universidad de Princeton, pero están más inclinados hacia la programación incluso a bajo nivel, y su sintaxis es complicada y difícil de entender. Es por ello, por lo que se decide diseñar nuestro propio lenguaje, siempre intentando hacerlo sencillo de entender y usar.

La idea que hay detrás, es que si un niño empieza a usar Sib, sea capaz de aprender conceptos básicos tanto de programación, como de música.

## *1.2 Objetivos*

El proyecto trata de alcanzar dos objetivos principales:

- Como principal objetivo, definir un lenguaje de programación orientado a la música, Sib. Que ofrezca cierto potencial para la solución de las tareas diarias de producción musical, pero a la vez sea sencillo de usar e intuitivo. Debido a la amplitud del lenguaje



musical, se trata de un punto de partida, que asiente las bases para poder extenderlo de forma sencilla y ganar más potencial en versiones futuras.

- El segundo objetivo es hacer que dicho lenguaje se pueda relacionar con el software ya existente en el mercado, lo cual facilitará su implantación. Crear un traductor de Sib a MusicXML, un estándar de código abierto usado por la mayoría de software de producción musical. Con el desarrollo del traductor, no sólo conseguimos generar código MusicXML, sino que se implementa un IDE donde poder escribir código Sib y poder validarlo.

### ***1.3 Organización de la memoria***

Según los dos objetivos vistos en el apartado anterior, la estructura de la memoria queda claramente dividida en dos partes, una primera dedicada al lenguaje Sib y una segunda parte dedicada al desarrollo de un traductor de Sib a MusicXML.

El primer capítulo donde nos encontramos es una breve introducción al escenario en el que se moverá el proyecto.

Es en el capítulo 2 donde se abordará el diseño del lenguaje Sib, y que el usuario podrá usar como guía de referencia para aprenderlo.

El traductor de Sib a MusicXML será tratado en el capítulo 3, dejando la gramática del lenguaje Sib usada por el traductor para el capítulo 4.

En el capítulo 5 se hará referencia al lenguaje MusicXML, es especial a los elementos que abarca el estado actual del lenguaje y traductor.

Debido a la amplitud del lenguaje musical, y el potencial que tiene el nuevo lenguaje Sib, en el capítulo 6 se mencionarán algunas líneas futuras de mejoras.

Por último, se presentan tres apéndices:

Apéndice A: Donde se hará una breve introducción al lenguaje musical.

Apéndice B: Donde se hablará sobre los generadores de analizadores léxicos y sintácticos usados, JFlex y Java CUP

Apéndice C: Mostrando las herramientas usadas para el desarrollo del proyecto.

En el apartado de Referencias, se encuentran tanto enlaces a páginas oficiales de los recursos usados, así como referencias a libros usados para documentación.

## Capítulo 2. Sib

En este capítulo se describirán las distintas características técnicas y funcionales del lenguaje Sib diseñado. Terminando con un par de ejemplo que servirán como referencia para entender su gramática.

### ***2.1 Introducción***

Sib es un lenguaje de programación orientado a la música. Pretende mejorar el proceso de composición, facilitando la representación, creación y edición de partituras.

Relaciona la creatividad y abstracción de la música, con el sentido lógico y funcional de la programación.

#### **Para músicos.**

Aunque al tratarse de un lenguaje de programación, son necesarios una serie de conocimientos informáticos, el uso de dicho lenguaje en herramientas orientadas al usuario final, dará a los músicos una potente herramienta para su día a día.

Pensado para músicos, son ellos los que podrán sacar el mayor aprovechamiento de su potencial.

#### **Para programadores.**

Como tal, su principal núcleo de usuarios, serán desarrolladores, que podrán tanto aprender ciertos aspectos del lenguaje musical con su uso, así como desarrollar software basado en Sib.

## 2.2 Características generales

Se trata de un lenguaje de programación de alto nivel.

**Case-sensitive:** distinguiendo entre mayúsculas y minúsculas.

**Fuertemente tipado:** por lo que las variables deberán ser definidas indicando el tipo que contendrán. Y no podrán contener valores que no pertenezcan a su tipo.

Las instrucciones terminarán todas con ';' (punto y coma)

Los comentarios serán definidos:

- Si la línea empieza por doble barra ( // ) el comentario dura hasta final de línea.

*// Esto es un comentario.*

- Si el comentario consta de varias líneas, entonces se encontrará entre '/'\* y '\*/'

*/\* Esto también es un comentario*

*pero escrito en varias líneas \*/*

Dispone de ciertas características no usuales, orientadas a la música:

Variable global \$partiture, que contendrá el estado actual de la partitura ( tempo, clave de la partitura, armadura ... )

Tipos 'clef', 'step' o 'note', representando los posibles valores de las claves, notas y objetos nota en el lenguaje.

Operadores específicos, como pueden ser \_#, \_b, \_@, \_\_ o \_ . que sólo tienen sentido en el lenguaje musical.

## 2.3 Estructura general

En la versión actual no se permite el uso de funciones, ni importación de ficheros, por lo que todo el código debe ir en un único fichero, que será procesado por el traductor.

Los programas en Sib siguen la estructura:

```
// Definición de paquete  
  
BEGIN <nombre del programa>  
  
// Zona de Instrucciones  
  
END
```

**Definición de paquetes:** Aunque actualmente sólo se permite la definición de un solo fichero, y no se importan otros ficheros o clases, se entiende como un paso natural en la evolución del lenguaje que en un futuro lo permita, por lo que se ha decidido requerir la definición del paquete al que pertenece el código.

```
PACKAGE <nombre>.<del>.<paquete>;
```

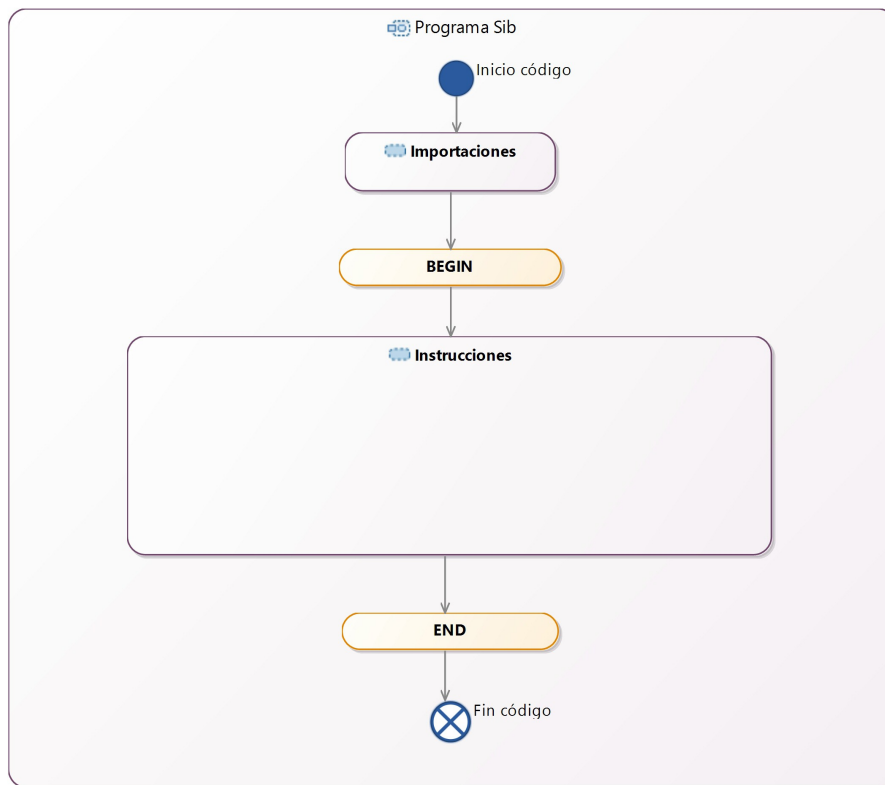
La ruta del paquete será una lista de identificadores válidos separados por punto ( . ).

**Importaciones:** No implementado en la versión 0.1, pero se contempla la zona de declaraciones para futuras versiones.

Todo programa Sib comenzará con la palabra reservada *BEGIN*, seguida del nombre del programa, que deberá ser un identificador válido.

A continuación irá el cuerpo del programa, formado por un conjunto de instrucciones.

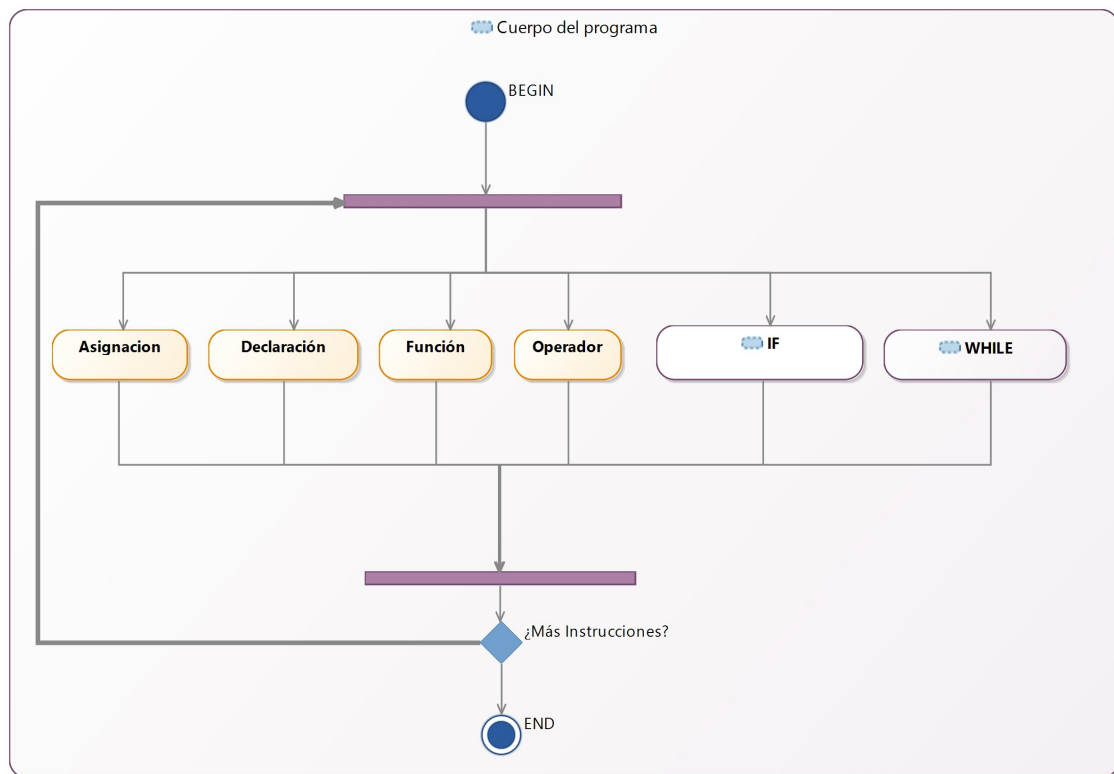
Terminando el programa con la palabra reservada *END*



## 2.4 Cuerpo del programa

El cuerpo principal de un programa Sib, se compone de una serie de Instrucciones de distinto tipo, donde cada instrucción termina siempre con el carácter punto y coma ( ; )

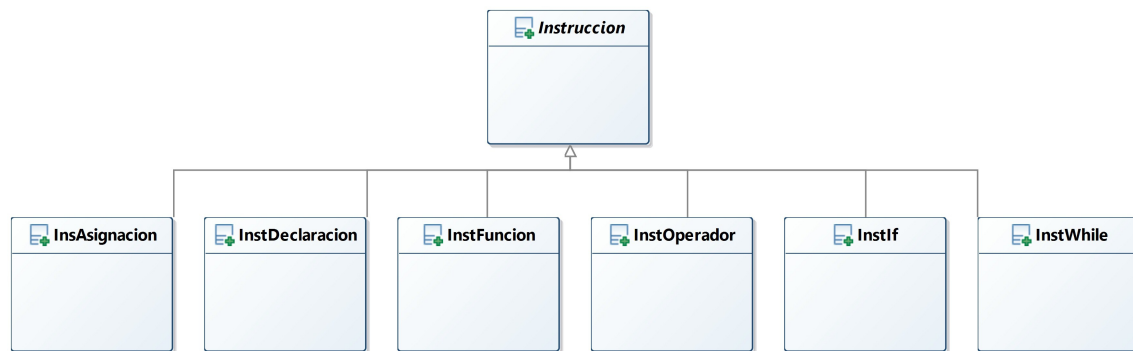
En el siguiente diagrama de actividad podemos ver el comportamiento normal del cuerpo de un programa Sib.



Las instrucciones permitirán realizar acciones sobre variables y sobre el entorno, residiendo en ellas el potencial del lenguaje.

Se ejecutarán secuencialmente tantas instrucciones como haya en el cuerpo del programa, pudiendo existir instrucciones atómicas, como asignaciones, declaraciones, funciones u operadores, e instrucciones con cierta estructura, como el condicional if o el bucle while.

Los distintos tipos de instrucciones disponibles son:



### Instrucción de Declaración.

Al tratarse de un lenguaje fuertemente tipado, las variables deben ser definidas indicando el tipo de datos que contendrá. Se permite la declaración múltiple de variables, separándolas por coma.

```
note $nota1, $nota2;
```

```
// $nota1 y $nota2 serán variables de tipo note.
```

### Instrucción de Asignación.

Permiten asignar valores a variables. Se permite tanto la asignación directa de valores finales, como que el origen de la asignación puede contener operaciones o funciones que devuelvan valores.

Tanto el antecedente como el consecuente de la asignación deberán ser del mismo tipo.

Asignación de tipos básicos simples. En este caso, el valor destino será una copia del valor origen.

```
$tono = G; // $tono valdrá G
```

```
$numero = 5; // $numero valdrá 5
```

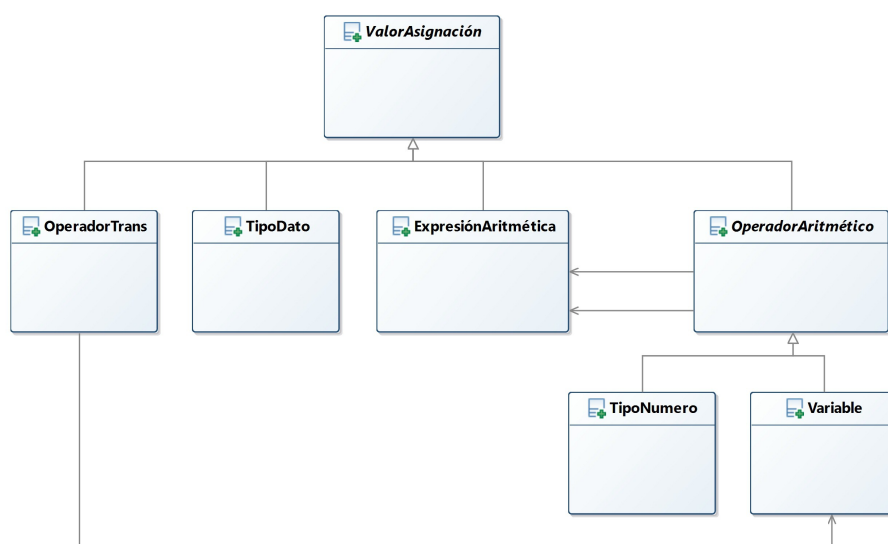
En el caso de variables cuyo valor sea de tipo compuesto, como por ejemplo tipo *note* o *partiture*, la asignación será una a una para cada una de las propiedades de los objetos

```
$nota1 = $nota2; // Asigna cada propiedad de $nota2 a $nota1
```

La asignación individual de valores a las propiedades se realizará mediante notación con punto.

*\$miObjeto.altura = 1;*

Un esquema de los posibles valores que se pueden usar en la asignación es:



### Instrucción Operador.

Sib dispone de operadores específicos relativos a la música, que permiten alterar valores como la duración y el tono de las notas musicales.

**\_<alteración>** : Operador unitario por la derecha, que aplica sobre variables de tipo note.

Modifica la alteración de la nota de acuerdo con el valor dado, siendo acumulativo, es decir,

**\_##** equivale a aplicar dos veces el operador **\_#**.

Con **\_#**, **\_##** se aplica la alteración de *sostenido* a la nota, por lo que sube su tonalidad tantos medios tonos como **#** se tenga.

Con **\_b**, **\_bb** se aplica la alteración de *bemol* a la nota, por lo que baja su tonalidad tantos medios tonos como **b** se tenga.

Con **\_@** se anulan las posibles alteraciones que la nota pudiese tener, aplicando becuadro a la nota.



Con `__` se anula las alteraciones de la nota, dejándola sin ninguna.

Son ejemplos de uso:

```
$miNota_##;
```

```
$miNota_b;
```

```
$miNota_;;
```

**\_`<puntillos>`** : Aplica 'puntillos' a una nota. Su valor es acumulativo, por lo que aplicar `_.` es equivalente a aplicar `..`.

Están definidos hasta tres puntillo como operador propio, ya que raramente se ven partituras con más de tres puntillos, pero en caso de ser necesario, al ser acumulativos, se pueden concatenar.

```
$varNota_;
```

```
$varNota_..;
```

```
$varNota_...;
```

**trans( note, frac )**: Devuelve una copia de la variable note, transportada cierto valor de tonos dado por el segundo parámetro. Actualiza tanto el valor como la octava de la nota en caso de ser necesario.

```
// Si tenemos
```

```
$nota.step = G;
```

```
$nota.octave = 4;
```

```
$nota = trans( $nota, 2 1/2 );
```

```
// Ahora sus valores son: value = C y octave = 5
```

### **Instrucción tipo función.**

Sib cuenta con funciones de entrada y salida de datos:

**play( )**: Función de salida/representación de datos. 'Imprime' la nota que se le pasa como parámetro. Internamente hará uso de la variable global \$partiture, que contiene información genérica sobre la partitura, como puede ser tempo, clave, etc ... Acepta como parámetro una

variable de tipo note.

```
note $nota;  
play( $nota );
```

**read():** Función para la lectura de notas. Devolverá la nota leída, desde la fuente de entrada. En nuestro caso, leerá la nota desde el fichero MusicXML que se le proporcionó al usar readPartiture().

```
note $nota;  
$nota = read();
```

**readPartiture():** Función para la lectura de las variables de entorno ( parámetros de la variable global \$partiture ) e inicialización del proceso de lectura. En nuestro caso, tomando como origen ficheros MusicXML, permite elegir el fichero origen, sobre el que posteriormente se trabajará al llamar a read().

### **Bloque condicional IF.**

Sib cuenta con una estructura condicional, la estructura IF-ELSE

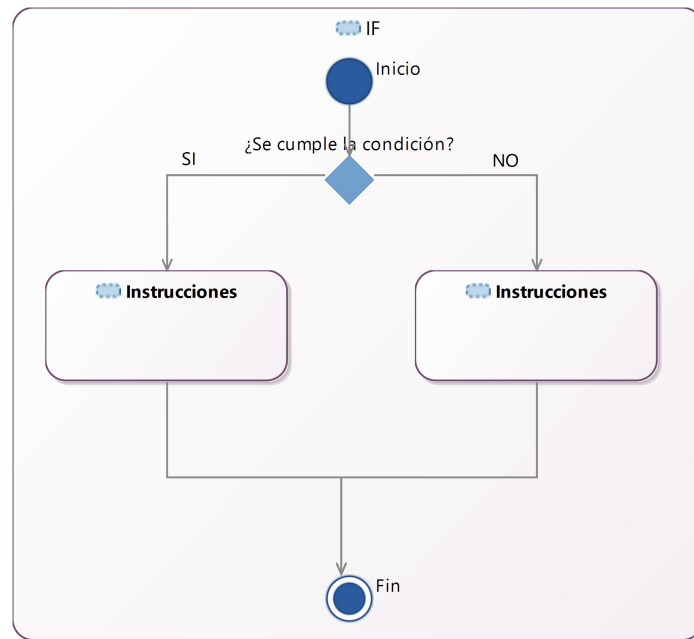
Nos permite ejecutar un conjunto de instrucciones sólo si se cumple una determianda condición, pudiendo elegir ejecutar otro bloque distinto de instrucciones, en caso de no cumplirse la condición.

```
IF ( <condición> )  
    <instrucciones_1>  
[ELSE  
    <instrucciones_2>  
] ENDIF
```

Si se cumple la *condición*, entonces se ejecutan las instrucciones del bloque <instrucciones\_1>.

En el caso de existir la parte opcional ELSE, cuando no se cumpla la condición, entonces se ejecutará el bloque de <instrucciones\_2>

Los bloques de instrucciones podrán contener otras estructuras *while* e *if*.



En el siguiente código ejemplo se puede ver su uso, en el que la variable \$nota sólo será "tocada" si se trata de un La (A)

```
IF ( $nota.step == A )
```

```
    play( $nota );
```

```
ENDIF;
```

### **Bloque iterativo WHILE.**

Como estructura iterativa, Sib incluye el bloque WHILE. Nos permite ejecutar un conjunto de instrucciones una y otra vez, hasta que deje de cumplirse la condición que sirve de guarda.

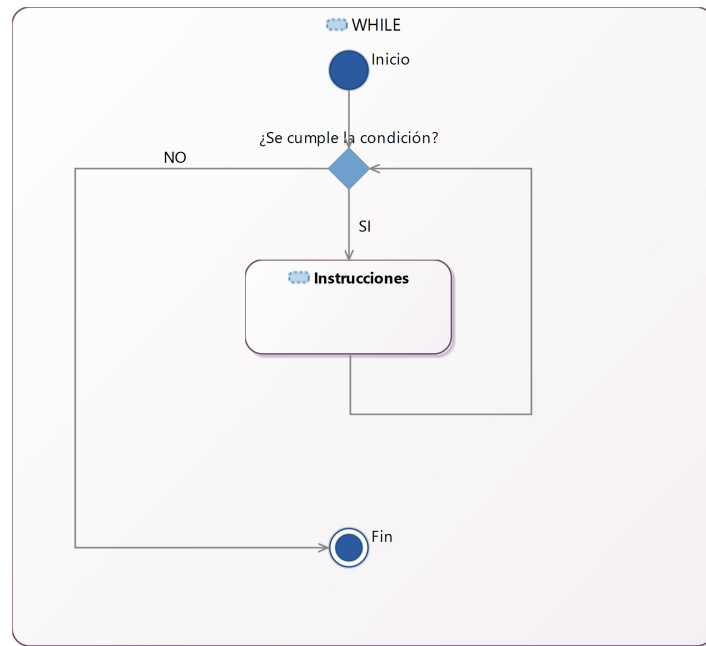
```
WHILE ( <condición> )
```

```
    <instrucciones>
```

```
ENDWHILE;
```

La estructura de control, empieza y termina con las palabras reservadas WHILE y ENDWHILE respectivamente.

El bloque de instrucciones podrá contener otras estructuras *while* e *if*.



En el siguiente código ejemplo, se "tocará" la variable \$nota 4 veces:

```
int $num;  
$num = 0;  
WHILE ( $num <= 3 )  
    play( $nota );  
    $num = $num + 1;  
ENDWHILE;
```

## 2.5 Operadores básicos

Sib ofrece además de un conjunto de operadores básicos comunes en la mayoría de los lenguajes de programación, un conjunto de operadores específicos orientado al lenguaje musical.

### 2.5.1 Comparación

#### **Menor que ( < )**

Indica si el operador de la izquierda es menor que el operador de la derecha.

#### **Mayor que ( > )**

Indica si el operador de la izquierda es mayor que el operador de la derecha.

#### **Menor igual ( <= )**

Indica si el operador de la izquierda es menor o igual que el operador de la derecha.

#### **Mayor igual ( >= )**

Indica si el operador de la izquierda es mayor o igual que el operador de la derecha.

#### **Igual ( == )**

Indica si ambos operadores son iguales en valor.

#### **Distinto ( != )**

Indica si ambos operadores son distintos en valor.

Aplicación sobre los tipos numéricos:

Es la correspondiente con sus valores matemáticos. En el caso de `frac` y `nfrac`, su valor `float` será usado para la comparación.

Aplicación sobre tipo `string`:

Realiza la comparación de cadenas de caracteres, priorizando su longitud, y en caso de igualdad de longitud, su valor ASCII carácter a carácter de izquierda a derecha.

*“hola” < “adios” // Por longitud*

*“ABCD” > “XYZ” // Por 'B' > 'X'*

Aplicación sobre tipo *step*:

Realiza la comparación, asociando valores numéricos a sus posibles valores, y comparando dichos números. La asociación a números es: A es 1, B es 2, C es 3, D es 4, E es 5, F es 6 y G es 7

Aplicación sobre tipo *clef*:

Realiza las comparaciones basándose en una función hash propia que devuelve un valor numérico a su combinación de *step* y número de línea. El número de línea prioriza a al valor de la nota.

La función hash es:

$$\text{valor numérico} = (\text{línea} * 7) + \text{Step.toInt()}$$

Aplicación sobre tipo *note*:

Las comparaciones sobre el tipo compuesto *note*, tiene en cuenta todos los atributos que la componen, aplicando de mayor a menor peso:

$$\text{octave} - (\text{value} + \text{accidental}) - (\text{duration} + \text{dots}) - \text{articulation}$$

octave compara como números

value + accidental compara como números, donde value se pasa a valor numérico y accidental suma 0.5 por cada sostenido, y resta 0.5 por cada bemol

duration + dots: compara como números, teniendo en cuenta que los puntillos suman la mitad del valor de la nota o del puntillo que lo precede.

Articulation compara como cadena de caracteres.

Aplicación sobre tipo *partiture*:

La comparación entre dos elementos de tipo *partiture*, aunque poco habitual su uso, aplica comparando los distintos atributos según orden de prioridad:

$$\text{clef} - \text{tempo} - \text{keysign} - \text{time} - \text{volume} - \text{wedge}$$

## 2.5.2 Operadores aritméticos

Sib soporta los operadores aritméticos básicos sobre números, incluido el operador módulo.

Los operadores aritméticos son aplicables sobre operandos de Tipo Numero (int, float, frac y

nfrac), existiendo una sobrecarga de dichos operadores, para permitir su aplicación sobre operandos de tipo note.

### **Suma ( + )**

Función matemática que suma dos valores numéricos. Aplica sobre el valor float que representa a cada uno de los Tipo Numero.

### **Resta ( - )**

Función matemática que resta dos valores numéricos. Aplica sobre el valor float que representa a cada uno de los Tipo Numero.

### **Multipliación ( \* )**

Función matemática que multiplica dos valores numéricos. Aplica sobre el valor float que representa a cada uno de los Tipo Numero.

### **División ( / )**

Función matemática que divide dos valores numéricos. Aplica sobre el valor float que representa a cada uno de los Tipo Numero.

### **Módulo ( % )**

Función matemática que calcula el resto de dividir el operando de la izquierda entre el operando de la derecha. Aplica sobre el valor float que representa a cada uno de los Tipo Numero.

### **Operadores +,-,\* y / sobrecargados**

Los operadores +, -, \* y / serán sobrecargados, tomando no sólo su comportamiento normal sobre números, sino que podrán ser aplicados sobre una nota (note)

<operando1> + <operando2>: <operando1> puede ser un elemento tipo note o tipo número, y <operando2> puede ser un elemento tipo número. 'Sumar' un valor numérico a una nota es transportar dicha nota ese número de tonos hacia tonalidades más agudas. Por lo que actualiza los atributos value y octave.

$$\text{\textit{\textcolor{blue}{\$varNota}}} = \text{\textit{\textcolor{blue}{\$varNota}}} + 2;$$

<operando1> - <operando2>: <operando1> puede ser un elemento tipo note o número, y <operando2> puede ser un elemento tipo número. 'Restar' un valor numérico a una nota es transportar dicha nota ese número de tonos hacia tonalidades más graves. Por lo que actualiza los atributos value y octave.

$$\text{\textit{\$varNota}} = \text{\textit{\$varNota}} - 2;$$

<operando1> \* <operando2>: <operando1> puede ser un elemento tipo note o número, y <operando2> puede ser un elemento tipo número. 'Multiplica' la duración de una nota, aumentando su valor.

$$\text{\textit{\$varNota}} = \text{\textit{\$varNota}} * 2;$$

*// Si \text{\textit{\\$varNota}} era una negra (dur. 1/4), ahora es una blanca (dur. 1/2)*

<operando1> / <operando2>: <operando1> puede ser un elemento tipo note o número, y <operando2> puede ser un elemento tipo número. 'Divide' la duración de una nota, disminuyendo el valor de su campo *duration*.

$$\text{\textit{\$varNota}} = \text{\textit{\$varNota}} / 2;$$

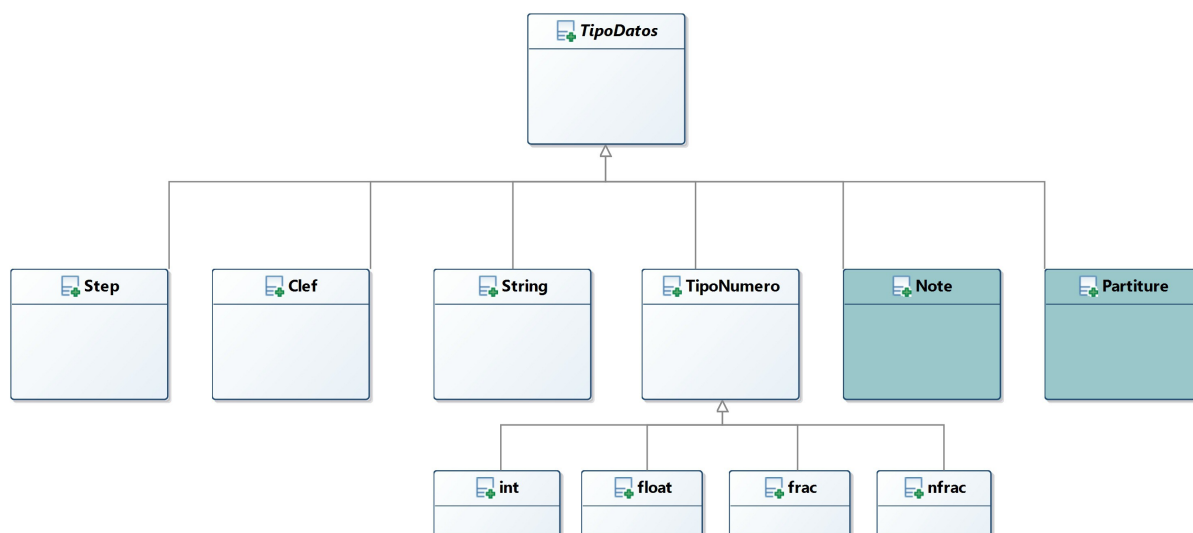
*//Si \text{\textit{\\$varNota}} era una negra (dur. 1/4), ahora es una corchea (dur. 1/8)*



## 2.6 Tipo de datos

En Sib se distinguen dos grupos de tipos de datos:

- **Tipos de datos básicos:** Entre los que se encuentran ciertos tipos de datos comunes a la mayoría de lenguajes de programación, y algunos específicos para la representación del lenguaje musical. En este grupo se encuentran: step, clef, string y tipoNumero, que engloba a int, float, frac y nfrac.
- **Tipos de datos compuestos:** Son tipos de datos compuestos por un conjunto de propiedades. Se entiende como propiedad a pares nombre-valor, siendo nombre un identificador válido y valor un tipo de dato. Son note y partiture.



### 2.6.1 Tipos de datos básicos

Se definen tipos de datos básicos aquellos que no extienden de ningún otro tipo de dato.

Se definen los siguientes tipos de datos básicos:

*TipoNumero* no es un tipo de dato como tal, y por ello no podrá ser usado como tipo de dato, sino que engloba varios tipos de datos de carácter numérico. Se hace mención en este apartado, simplemente para que conceptualmente se asocie a los tipos de datos que representa (int, float, frac y nfrac).

**int:** Define el tipo entero en 32 bits. Con valores entre  $-2^{31}$  y  $2^{31}-1$ . El uso del guión antes del número, indica valores negativos, y el uso del signo más (+) o la ausencia de signos, indica valores positivos.

```
int $varInt;
```

```
$varInt = 2;
```

```
$varInt = -10;
```

**float:** Define un número en punto flotante y simple precisión en 32 bits, siguiendo el estándar IEEE 754. El uso de signos precediendo a los número toma el mismo sentido que con int.

```
float $varFloat;
```

```
$varFloat = 1.25;
```

```
$varFloat = -5.556;
```

**frac:** Define al tipo fracción. Se trata de una representación de las fracciones de números. Se compone de dos partes: un número entero (int), que representa la parte entera del valor, seguido de una fracción, que representa la parte decimal.

Su valor como número real (float) es la suma de la parte entera, más el resultado de dividir la fracción que le sigue.

Tanto numerador como denominador deben ser números enteros (int) positivos.

*frac \$varFrac;*

*\$varFrac = -1 1/2;*

**nfrac:** (Natural frac) Define las fracciones positivas de números. Por lo que tanto numerador como denominador deben ser valores enteros (int) positivos.

*nfrac \$varNFrac;*

*\$varNFrac = 3/4;*

**string:** Define una secuencia de caracteres entrecomillados.

Si se requiere el uso de las dobles comillas como parte del valor del string, debe usarse la barra invertida ( \ ) como carácter de escape.

*string \$varString;*

*\$varString = "Y él dijo \"Hola mundo\"";*

**clef:** Define posibles valores de claves musicales.

Está compuesto por 2 caracteres:

- El primero indica el tono de la nota.
- El segundo indica la línea sobre el pentagrama.

Sus posibles combinaciones de valores son: G2, F4, F3, C1, C2, C3, C4

*clef \$varClef;*

*\$varClef = F4;*

**step:** Define los valores de las notas musicales desde LA a SOL en anglosajón, es decir, A...G.

El silencio será representado por la letra S.

*step \$varStep;*

*\$varStep = A;*

## 2.6.2 Tipos de datos compuestos

Se definen como aquellos compuestos por una serie de propiedades.

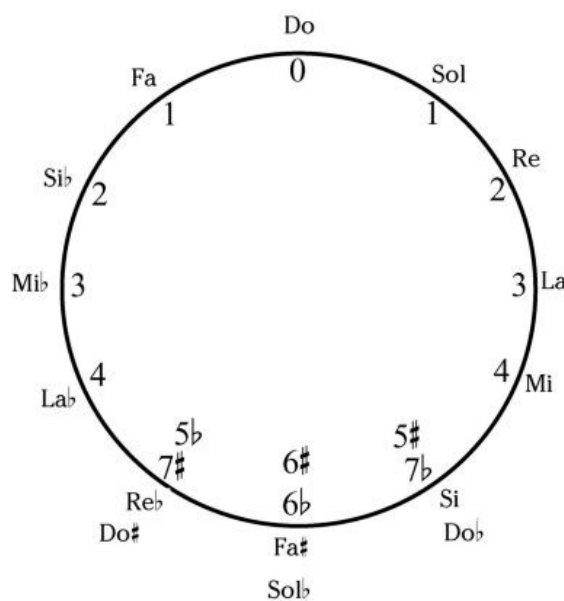
Las propiedades con pares nombre-valor, donde el nombre será un identificador válido, y el valor será de un tipo de dato simple, por ejemplo: `time - 60`

Sib cuenta con dos tipos de datos compuestos predefinidos, *partiture* y *note* con las siguientes características.

### ***partiture:***

Tipo de dato compuesto que define el estado de una partitura. Cuenta con las siguientes propiedades:

- **clef:** Indica la clave de la partitura, siendo de tipo clef. Con valor por defecto: G2
- **tempo:** Indica la velocidad, el tempo, siendo de tipo int. Con valor por defecto: 60
- **keysign:** Indica la armadura del compás, siendo de tipo int. Está basado en el círculo de quintas, donde valores positivos indican el número de sostenidos, valores negativos el número de bemoles, y cero indica que no hay ninguna alteración en la armadura. Con valor por defecto: 0



- **time:** Indica el tiempo de compás. Es de tipo nfrac. El numerador indica el número

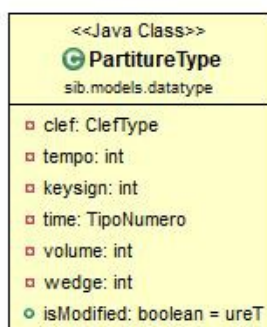
de tiempos, y el denominador, el tipo de nota por tiempo. Su valor por defecto es: 4/4 (cuatro tiempos de negras)

- **volume:** Indica el volumen actual, siendo de tipo float. Con rango entre 0 y 100, con valor por defecto: 50.
- **wedge:** Entero que indica si existe una notación de crescendo (1) o disminuyendo(-1) en el momento actual. Valor por defecto, cero (no existe dicha notación).

Por lo general, no es necesario usar este tipo de dato, ya que por defecto existe una variable global predefinida, llamada \$partiture, que contiene el “estado” actual de la partitura.

El acceso y escritura de sus propiedades se hará mediante la notación punto, por ejemplo:

*\$partiture.volume = 60;*



#### **note:**

Define a las notas musicales. Se trata de un tipo compuesto, con las propiedades: *value*, *duration*, *octave*, *dots*, *accidental* y *articulation*, con valores por defecto: C, 4, 4, 0, " y " respectivamente.

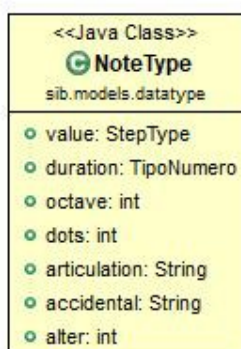
- **value:** Define el valor de la nota como tal. Contendrá valores de tipo step. Valor por defecto C.
- **duration:** Indica la duración de la nota según el sistema anglosajón. Donde 1/1 es una redonda, 1/2 una blanca, 1/4 una negra, 1/8 corchea, 1/16 semicorchea, 1/32 fusa y 1/64 semifusa. Es de tipo nfrac. Valor por defecto 1/4 (negra).
- **octave:** Indica la octava en la que se encuentra la nota. Usa el sistema

científico/internacional, con valores comprendidos entre 0 y 7 (variable tipo int).  
Valor por defecto 4.

- **dots:** Número de puntillos a aplicar a la duración de la nota. Por defecto es cero.
- **articulation:** Indica la articulación de la nota en caso de tenerla. Posibles valores: "staccato", "tenuto", "accent" y cadena vacía ( "" ), que indica que no aplica acento. Aunque existen más posibles valores musicales que podemos usar como articulación, Sib usa actualmente este conjunto de los más comunes.
- **accidental:** Indica la existencia de una alteración accidental sobre la nota.. Se trata de campo tipo String, con posibles valores internos: "natural", "flat", "flat-flat", "sharp", "double-sharp" y cadena vacía ( "" ), que indica que no aplica símbolo accidental. Su aplicación se hará con los operadores de nota: `_@`, `_b`, `_bb`, `_#`, `_##`, `__` (doble guión) respectivamente.

Al igual que con \$partiture, el acceso y escritura de sus propiedades se hará mediante notación punto, por ejemplo:

*\$nota.octave = \$nota.octave + 1;*



## 2.7 Variables

Las variables son “contenedores” que pueden albergar algún tipo de dato. Deben ser previamente declaradas para poder usarse, y para ello se usa el tipo de instrucción Declaración (vista en la sección 2.4)

La declaración de variables se realiza indicando inicialmente el tipo de la variable ( que al ser un lenguaje fuertemente tipado, es obligatorio ) y a continuación el nombre de la variable.

Los nombres de variables siguen las siguientes normas de composición:

- El nombre de variable empieza por el carácter dólar ( \$ ), seguido del identificador.
- El identificador será una cadena de caracteres alfanuméricos, siendo el primer carácter de tipo letra.

*int \$unaVariable123;*

El ámbito de las variables en la versión actual, es global, ya que aún no están definidas las posibles funcionalidades de importación y creación de funciones.

### \$partiture

Especial mención a \$partiture, una variable predefinida por el sistema, y que define el '*estado*' de la partitura sobre la que trabajará el código del programa escrito en Sib.

No es necesario declararla, se encuentra disponible en todo el cuerpo del programa Sib.

Existe un tipo de dato compuesto, fuertemente ligado a esta variable, y del cual es tipo, '*partiture*'.

Dicha variable toma inicialmente los valores por defecto del tipo, pudiendo accederse a ella desde la parte de instrucciones del programa, consultando / modificando sus valores.

En el apartado 2.6.2 de tipo de datos compuestos, se puede consultar más información sobre el tipo *partiture*.

## 2.8 Mejoras y versiones futuras

A continuación se muestran posibles mejoras en distintos aspectos del lenguaje, que le proporcionarán mayor potencia y funcionalidad.

### Estructura general de un programa Sib

Permitir la importación de código procedente de otros ficheros mediante la directiva `IMPORT`.

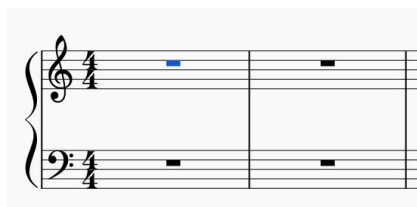
Esto nos dará más flexibilidad a la hora de escribir código, evitando tener que tenerlo todo en un mismo fichero.

El uso de llamadas a funciones y/o macros facilitará la inserción de código externo en el cuerpo principal del programa.

Probablemente se opte por implementar una versión de macros para la versión 0.2 del lenguaje. Entendiéndose como macro las declaraciones de código que serán directamente sustituidos donde el macro se llame. Aunque no sea la solución más potente, evitará que el cuerpo del programa sea demasiado largo, y poder estructurarlo mejor.

### Llave

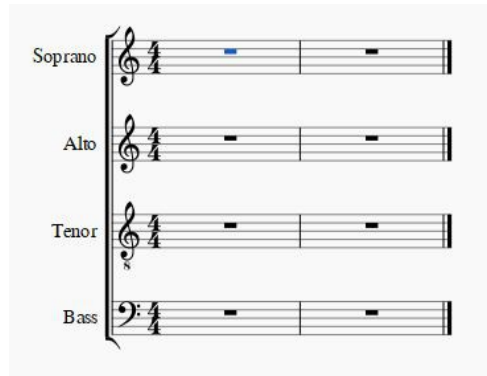
Implementación de llaves, que permitirá representar varios pentagramas que pertenecen al mismo instrumento, al mismo tiempo. Instrumentos como el piano u órgano trabajan con dos pentagramas a la vez. Esta funcionalidad se representa en MusicXML con los nodos `<staves>`





## Sistema

Implementación de sistemas, que permita representar varios pentagramas a la vez, indicando que se van a interpretar al mismo tiempo, pero por distintos instrumentos. Funcionalidad representada en MusicXML con los nodos <score-part>



## Tipo object

Sib usará el tipo *object* para definir tipos de datos compuestos, define a los *objetos*. Un objeto es una representación de una colección de propiedades.

Se entiende como propiedad, al atributo que representa una característica de dicho objeto. Cada propiedad es un par nombre-valor perteneciente a un determinado tipo.

```
object $varObjeto;
```

```
string $varObjeto.nombre;
```

```
float $varObjeto.altura;
```

En la declaración de un tipo object no se especifican las propiedades al definir a la variable, sino que se realiza posteriormente.

Las propiedades deben tener un tipo de dato, por lo que hay que indicarlo al definirlas. Tomar como referencia el ejemplo de arriba, donde al definir la propiedad 'nombre', se indica que su tipo es string.

Tanto el acceso como la lectura de propiedades, se realiza mediante notación con punto:

`<var_object>.<propiedad>.`

```
float $varFloat;
```

```
$varFloat = $varObjeto.altura;
```

```
$varObjeto.altura = 1.75;
```

Según el conjunto de datos actual, los tipos de datos compuestos (Note y Partiture) extenderán del tipo Object.

Asignación múltiple de valores mediante una lista clave : valor, siendo la clave el nombre de la propiedad, y el valor, el valor a asignar.

```
$varObjeto = { nombre : "Antonio", altura : 2 };
```

```
$varNota = { dots : 1 };
```

El usuario puede definir sus propios tipos compuestos del tipo objeto en el desarrollo de sus programas. Para ello, deberá hacerlo en ficheros .sid que podrá importar en la sección inicial de importaciones de su programa.

Si se define un tipo ya existente, las propiedades nuevas serán añadidas a las propiedades ya definidas con anterioridad, sobrescribiéndose en caso de existir una propiedad con el mismo nombre. Es por ello, por lo que el orden de importación tiene relevancia.

Estos nuevos tipos definidos tendrán visibilidad dentro del paquete actual desde el que se importó el fichero de definición.

## **Tipo Array**

Tipo que permitirá guardar un conjunto de elementos del mismo tipo. En su definición hay que indicar el tipo que contendrá y el número de elementos (longitud del array).

```
array(2) string $varArray;
```

La asignación de valores se puede realizar individualmente, haciendo uso de corchetes ( [ ] ), entre los cuales se indicará la posición del elemento del array al que asignar valor. La numeración de posiciones será mayores a uno.

```
$varArray[1] = "Antonio";
```

```
$varArray[2] = "Blanco";
```

```
// El array $varArray contendrá los valores "Antonio" y "Blanco"
```

La asignación múltiple se realiza entre llaves ( { } ). Indicando entre ellas un listado de valores separados por comas. Leyendo de izquierda a derecha, se irán asignando valores al array incrementando su posición.

```
$varArray = {Antonio, Blanco};
```

```
// El array $varArray contendrá los valores "Antonio" y "Blanco"
```

```
// Ejemplo similar al de la asignación simple de arriba.
```

### **Ficheros de definiciones de tipos**

Los ficheros de definición de tipos compuestos, tendrán extensión sid ( sib definitions ).

Constarán de un conjunto de definiciones, permitiéndose comentarios (de una o varias líneas).

Si la definición de un tipo ya existe, o bien porque se trate de un tipo global definido por el lenguaje o por que ya se definió en otro fichero anteriormente cargado, el tipo objeto resultante contendrá la unión de propiedades, sobrescribiendo las ya existentes.

Como reglas de producción tendremos:

<code>&lt;fichero&gt;</code>	→	<code>&lt;definiciones&gt;</code>
<code>&lt;definiciones&gt;</code>	→	<code>&lt;definición&gt;&lt;lista_definiciones&gt;</code>
<code>&lt;lista_definiciones&gt;</code>	→	<code>&lt;definición&gt;&lt;lista_definiciones&gt;</code>
		E
<code>&lt;definición&gt;</code>	→	<code>&lt;comentarios&gt;</code>
		<code>&lt;str_definición&gt;</code>

`<str_definición>      →      def <str_ident> { <declaraciones> }`

Tanto `<comentarios>`, `<declaraciones>` como `<str_ident>` se encuentran definidos en las reglas de producción del lenguaje (sección **Definición de la Gramática**).

Posible ejemplo de definición de un tipo *myNote*:

```
// Type: myNote
```

```
def myNote {  
  
    step v;  
  
    int d;  
  
    int o;  
  
    string a;  
  
}
```

## Traducible

Hacer las palabras reservadas “traducibles”. Se puede indicar en la zona de importaciones una instrucción que indique el idioma a usar, por ejemplo:

```
LANG es;
```

Entonces el preprocesador sabrá que tiene que cargar un fichero `.sil` (Si language), donde habrá traducciones de las palabras reservadas, un ejemplo de fichero `es.sil` puede ser:

```
BEGIN = INICIO
```

```
END = FIN
```

```
IF = SI
```

```
ELSE = SINO
```

```
WHILE = MIENTRAS
```

## 2.9 Ejemplos Sib

A continuación se muestran algunos ejemplos de código sib.

### 2.9.1 Ejemplo escala DO mayor

Escala Do Mayor de negras.



El código Sib sería:

```
/* Programa que representa la escala de Do mayor con negras.*/  
BEGIN escalaDOMayor
```

```
  // Variables
```

```
  int $num;
```

```
  note $nota;
```

```
  $num = 0;
```

```
  WHILE ( $num <= 7 )
```

```
    play( $nota );
```

```
    IF ( $nota.step == B )
```

```
      $nota = trans($nota, 1/2);
```

```
    ELSE
```

```
      IF ( $nota.step == E )
```

```
        $nota = trans($nota, 1/2);
```

```
      ELSE
```

```
        $nota = trans($nota, 1);
```

```
      ENDIF;
```

```
    ENDIF;
```

```
    $num = $num + 1;
```

```
  ENDWHILE;
```

```
END
```

Y el lenguaje MusicXML generado:

```
<?xml version="1.0" encoding="UTF-8"
standalone="no"?>
<!DOCTYPE score-partwise PUBLIC
"-//Recordare//DTD MusicXML 3.0
Partwise//EN"
"http://www.musicxml.org/dtds/partwise.dtd">
<score-partwise version="3.0">
  <part-list>
    <score-part id="P1">
      <part-name>Generada por Sib</part-
name>
      </score-part>
    </part-list>
    <part id="P1">
      <measure number="1">
        <attributes>
          <divisions>16</divisions>
          <key>
            <fifths>0</fifths>
          </key>
          <time>
            <beats>4</beats>
            <beat-type>4</beat-type>
          </time>
          <clef>
            <sign>G</sign>
            <line>2</line>
          </clef>
        </attributes>
        <note>
          <pitch>
            <step>C</step>
            <octave>4</octave>
          </pitch>
          <duration>16</duration>
          <type>quarter</type>
        </note>
        <note>
          <pitch>
            <step>D</step>
            <octave>4</octave>
          </pitch>
          <duration>16</duration>
          <type>quarter</type>
        </note>
        <note>
          <pitch>
            <step>E</step>
            <octave>4</octave>
          </pitch>
          <duration>16</duration>
          <type>quarter</type>
        </note>
        <note>
          <pitch>
            <step>F</step>
            <octave>4</octave>
          </pitch>
          <duration>16</duration>
          <type>quarter</type>
        </note>
        <note>
          <pitch>
            <step>G</step>
            <octave>4</octave>
          </pitch>
          <duration>16</duration>
          <type>quarter</type>
        </note>
        <note>
          <pitch>
            <step>A</step>
            <octave>4</octave>
          </pitch>
          <duration>16</duration>
          <type>quarter</type>
        </note>
        <note>
          <pitch>
            <step>B</step>
            <octave>4</octave>
          </pitch>
          <duration>16</duration>
          <type>quarter</type>
        </note>
        <note>
          <pitch>
            <step>C</step>
            <octave>5</octave>
          </pitch>
          <duration>16</duration>
          <type>quarter</type>
        </note>
      </measure>
    </part>
  </score-partwise>
```

## 2.9.2 Trasportar partitura

Con el siguiente programa, se consigue transportar las notas de una partitura, dos tonos y medio más agudos:

/\*

*En este ejemplo se supone que la partitura de entrada tiene 8 notas.*

\*/

*BEGIN AltoATenor*

*note \$nota;*

*int \$cnt;*

*readPartiture();*

*\$cnt = 0;*

*WHILE ( \$cnt < 8 )*

*\$nota = read();*

*\$nota = trans( \$nota, 2 1/2 );*

*play( \$nota );*

*\$cnt = \$cnt + 1;*

*ENDWHILE;*

*END*

Partiendo de una partitura como la siguiente:

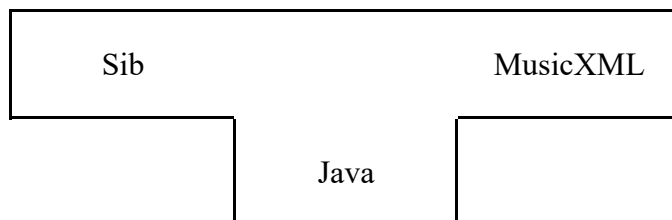


Obtendríamos como resultado:



### Capítulo 3. Traductor Sib a MusicXML

En este tercer capítulo, se abordará otro de los pilares del proyecto, el diseño e implementación de un traductor de lenguaje Sib a MusicXML, implementado en Java.





### 3.1 Introducción

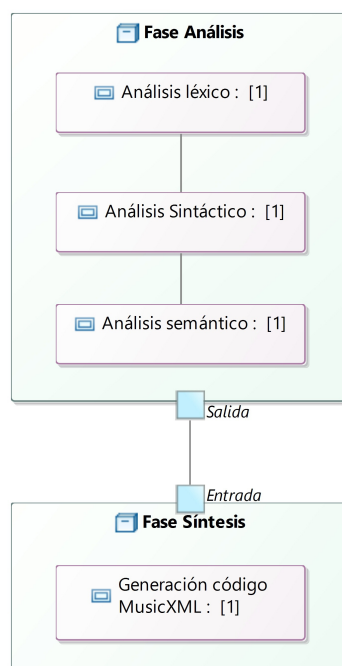
Se ha optado por traducir Sib a unos de los lenguajes basados en XML más extendido para el intercambio de música entre aplicaciones, MusicXML. Esto permitirá no sólo estudiar MusicXML, sino que las salidas que produzca nuestro traductor, puedan ser probadas en software comercial.

Dicho traductor será generado por las librerías Jflex y Java CUP, que permiten generar analizadores léxicos y sintácticos respectivamente.

Se distinguen dos fases en el funcionamiento del traductor.

Una primera fase de análisis de código Sib. Donde se analiza el código fuente Sib, para ver si se trata de un programa bien formado según su gramática y según las reglas sintácticas preestablecidas.

Y una segunda fase de síntesis, para la generación de código objeto, en este caso, MusicXML.

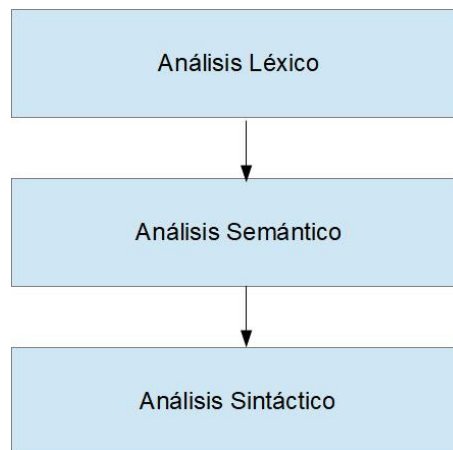


### ***3.2 Fase de Análisis***

En esta primera parte de análisis, el traductor hará tareas de compilación, analizando el código Sib suministrado y verificando si se trata de un código Sib válido.

Mediante Jflex se ha generado un analizador léxico, y mediante Java CUP, un analizador sintáctico, que basándose en la gramática definida en el apartado “Definición de la Gramática Sib”, darán validez o no al código Sib.

La fase de análisis se puede dividir en tres subfases, cada una con un cometido específico:



### 3.2.1 Análisis Léxico

Se encargará de leer el programa fuente carácter a carácter, agrupándolo en componentes léxicos o tokens. Se encarga de testear si dichos tokens pertenecen al código fuente permitido o no, en cuyo caso produce un error.

Será el analizador sintáctico el encargado de hacer peticiones al analizador léxico, e irle pidiendo los distintos tokens permitidos en cada parte del código.

Otras tareas adicionales del analizador léxico son:

- Tratamiento de los errores léxicos.
- Eliminar comentarios, saltos de líneas, espacios en blanco ...
- Inclusión de ficheros y expansión de macros (no implementado en la versión actual)
- Calcular el número de línea y columna donde aparece el token. Esto ayudará en las labores de debug.

La detección de errores puede ser de distinta índole: símbolo no permitido, identificador mal construido, cadenas no cerradas, números mal contruidos....

Se opta por un tratamiento de errores básico, indicando la posición del error en el debug, y parando la ejecución. En la sección de mejoras en versiones futuras, se indican algunos tratamientos de errores que podrían ser incluidos y ayudarán en la escritura de código Sib.

Los caracteres “vacíos” de contenido, como los comentarios (tanto de línea simple como multi líneas), los saltos de páginas (se contemplan tanto saltos de líneas en codificación Windows como Unix), y los espacios en blanco, son eliminados en esta fase de la traducción.

Aunque la versión actual no tiene desarrolladas las macros, y todo el código se escribe en un único fichero, se deja especificado un preprocesador en el apartado de futuras mejoras para la inclusión de estos en ficheros anexos, por lo que se podrán generar “ficheros librerías” con funcionalidad común que serán sustituidas en el fichero .sib general donde proceda.

El cálculo de línea y columna son simples parámetros que ayudan en la localización de errores léxicos, y hacen la tarea de programar mucho más sencilla.

El analizador léxico ha sido generado por JFlex, un generador de analizador léxicos desarrollado en Java.

Siguiendo un diagrama de transiciones, el analizador va leyendo carácter a carácter el código de entrada, que basado en las expresiones regulares que se le suministra, pasa por los distintos estados, y en caso de llegar a algún estado final, devuelve un token. Siempre siguiendo la premisa de devolver el token de mayor longitud posible.

Será el analizador sintáctico el que solicite el siguiente token a nuestro analizador léxico.

Su código se encuentra en el fichero *AnalizadorLexicoFase2.flex* del paquete *sib.flex*.

En el apéndice B se puede consultar más información sobre JFlex.

### 3.2.2 Análisis Sintáctico

El analizador sintáctico procesará los distintos tokens que le suministra el analizador léxico, para organizarlos internamente en forma de árbol y así poder comprobar si una cadena de entrada pertenece a nuestro lenguaje Sib.

Al igual que el analizador léxico es implementado basado en expresiones regulares, el analizador sintáctico, usa el lenguaje BNF para representar las reglas que debe cumplir el código de entrada.

Los analizadores sintácticos se suelen agrupar en dos conjuntos, dependiendo de cómo recorran el árbol de dependencias generado:

- Analizadores sintácticos descendentes (LL):  
Partiendo del axioma inicial, que será la raíz del árbol, recorren el árbol hacia abajo, hasta intentar construir la frase que se está analizando.
- Analizadores sintácticos ascendentes (LR):  
Recorren el árbol desde las hojas, que serán los distintos tokens de la entrada, subiendo por el árbol hasta intentar llegar a la raíz, que será el axioma inicial. En este grupo se encuentra nuestro analizador sintáctico generado. Más concretamente, se trata de un analizador LALR (look-ahead LR), que nos permite con una menor cantidad de estados, y por consiguiente memoria, analizar la mayoría de lenguajes LR(1).

Su código se encuentra en el fichero *AnalizadorSintactico.cup* del paquete *sib.cup*.

En el apéndice B se puede consultar más información sobre Java CUP.

### 3.2.3 Análisis semántico

Es la fase encargada de comprobar que “tiene sentido” el código Sib escrito. No sólo que está bien escrito y estructurado (de lo cual se encargan los analizadores léxicos y sintáctico), sino que los tipos son los correctos, las operaciones se pueden realizar sobre dichos tipos, etc ...

Puntos que se han tenido en cuenta en dicho análisis:

- Se trata de un lenguaje fuertemente tipado, por lo que las variables deben ser previamente definidas, indicando su tipo, y pudiendo sólo albergar información de dicho tipo.
- Operaciones y operandos. Aunque en el análisis sintáctico se filtra gran posibilidad de fallos en este sentido, ya que por la estructura del programa sólo se permiten ciertos tipos de elementos a los que aplicar operandos y operaciones, queda abierta la posibilidad de fallo en caso de usarse variables, que pueden ser de muy distintos tipos. Esta fase se encarga de chequear los tipos de las variables y verificar si son aplicables los operandos y operaciones.

Estos y otros puntos de chequeo semántico son realizados por las distintas clases que representan a los elementos generados durante el análisis sintáctico, como pueden ser los operadores, los tipos de datos o instrucciones.

### **3.3 Fase de Síntesis**

Al no tratarse de un traductor como tal, ya que el lenguaje destino, no es un lenguaje de programación, sino un formato de salida, la fase de síntesis se reduce a una sólo fase de generación de código MusicXML. Obviando fases de generación de código intermedio u optimización de código, muy comunes en esta fase de los traductores.

La fase de síntesis se encargará de generar el código MusicXML resultante de la ejecución del programa Sib.

### 3.3.1 Generación de código MusicXML

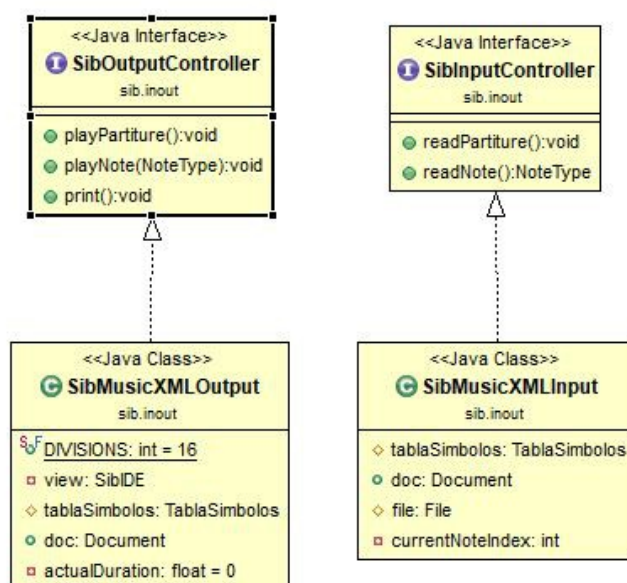
Haciendo uso de la interfaz de salida predefinida, el traductor a MusicXML generará como salida código MusicXML.

Se ha optado por una crear una interfaz que nos permita en un futuro poder reutilizar el traductor y poder generar otros tipos de salidas, como por ejemplo directamente generar sonido.

La función principal Sib orientadas para la generación de MusicXML será play(), que nos permite representar en MusicXML una determinada nota con sus propiedades.

Para poder cerrar la comunicación con MusicXML, y no sólo generar dicho código, sino también poder leerlo y poder actuar sobre él, se ha creado una interfaz de lectura MusicXML.

Las interfaces de entrada y salida con sus clases que implementan para MusicXML, quedarían:



Para poder entender mejor el código MusicXML, se recomienda leer el capítulo 5 dedicado a introducir MusicXML y su formato.



### 3.4 Sib IDE

Para la validación de código Sib y posterior traducción a MusicXML, se ha desarrollado un software que nos permita realizar dichas tareas.

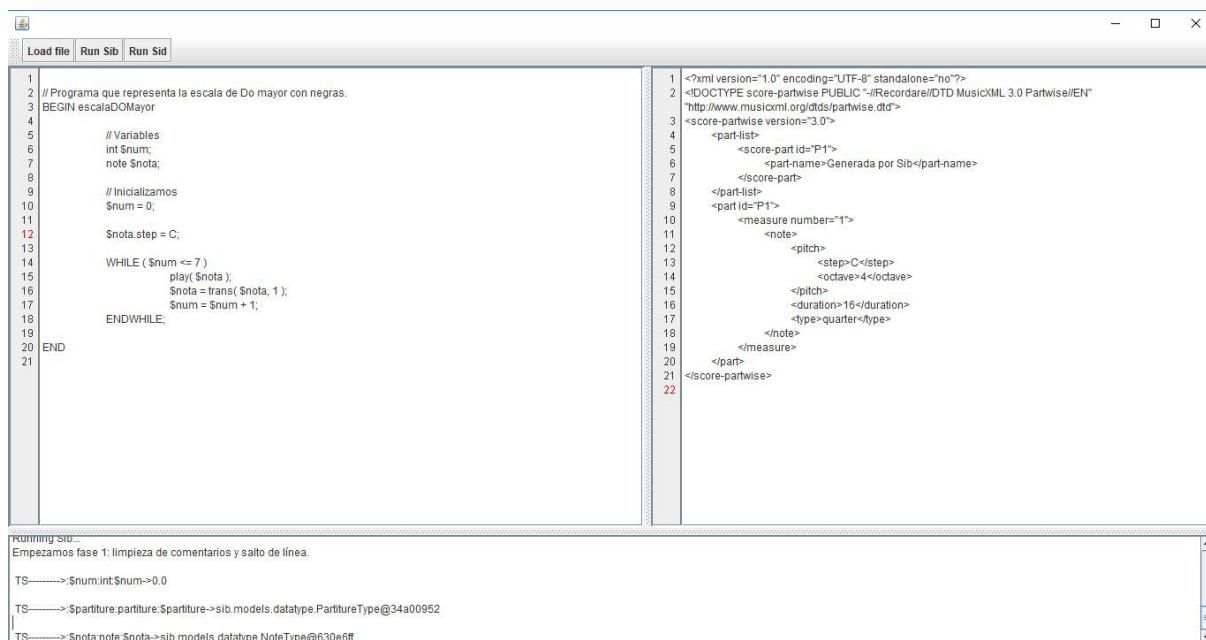
El software “Sib IDE” nos permite cargar código .sib bien escribiéndolo en el panel de entrada de código, o cargándolo desde archivo.

Hay disponible una carpeta con códigos ejemplo, que se pueden utilizar para pruebas y como referencia para el desarrollo de código propio.

Una vez tengamos el código escrito, el programa nos permite ejecutar dicho código, validándolo y generando una salida en código MusicXML.

Disponemos del panel de Debug, donde poder ver los posibles fallos, y demás información del proceso de validación y traducción.

Además en el panel de Salida se generará el código MusicXML resultante, pudiéndose guardar en archivo .xml para su posterior uso.



The screenshot displays the Sib IDE application window. It features a menu bar with 'Load file', 'Run Sib', and 'Run Sid'. The main workspace is divided into two panes. The left pane contains the original Sib code, which defines a scale and a loop to play notes. The right pane shows the resulting MusicXML code, which uses standard XML tags like <note>, <pitch>, <duration>, and <type> to represent the same musical information. At the bottom of the window, a 'Running Sib...' status bar provides real-time feedback, indicating the current phase of execution (cleaning comments and line breaks) and the specific variables being processed, such as \$num, \$partiture, and \$nota.

```
1 // Programa que representa la escala de Do mayor con negras.  
2 BEGIN escalaDOMayor  
4  
5 // Variables  
6 int $num;  
7 note $nota;  
8  
9 // Inicializamos  
10 $num = 0;  
11  
12 $nota.step = C;  
13  
14 WHILE ( $num <= 7 )  
15   play( $nota );  
16   $nota = trans( $nota, 1 );  
17   $num = $num + 1;  
18 ENDWHILE;  
19  
20 END  
21
```

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>  
2 <!DOCTYPE score-partwise PUBLIC "-//Recordare//DTD MusicXML 3.0 Partwise//EN"  
3 "http://www.musicxml.org/dtds/partwise.dtd">  
4 <score-partwise version="3.0">  
5   <part-list>  
6     <score-part id="P1">  
7       <part-name>Generada por Sib</part-name>  
8     </score-part>  
9   </part-list>  
10   <part id="P1">  
11     <measure number="1">  
12       <note>  
13         <pitch>  
14           <step>C</step>  
15           <octave>4</octave>  
16         </pitch>  
17         <duration>16</duration>  
18         <type>quarter</type>  
19       </note>  
20     </measure>  
21   </part>  
22 </score-partwise>
```

Running Sib:  
Empezamos fase 1: limpieza de comentarios y salto de línea.  
TS----->\$num.int:\$num->0.0  
TS----->\$partiture.partiture:\$partiture->sib.models.datatype.PartitureType@34a00952  
TS----->\$nota.note:\$nota->sib.models.datatype.NoteType@630e6ff

### **3.5 Futuras versiones**

Se contemplan las siguientes mejoras independientes de la versión Sib a analizar. Son mejoras propias del traductor.

#### **Gestión de Errores**

Se podría mejorar el tratamiento de errores, continuando la ejecución hasta encontrar X errores e ir informando en el debug o incluso 'solucionar' algunos sencillos, como escribir en minúsculas alguna palabra reservada o la falta de un punto y coma.

#### **Tabla de Variables**

Añadir panel para poder mostrar los valores de las variables. Ayudará a labores de debug.

## Capítulo 4. Gramática

Capítulo dedicado a la gramática que define al lenguaje Sib. Se hará una breve introducción al concepto de gramática, para luego presentar la gramática definida. Por último, se presentará un ejemplo de árbol sintáctico, que ayudará a comprender cómo se descompone una frase del lenguaje.

### 4.1 Introducción

Para tener un conocimiento más exacto sobre las gramáticas y su uso en el desarrollo y diseño de lenguajes de programación, se va a empezar esta sección explicando algunos conceptos básicos sobre ellas.

Un **gramática** es una cuádrupla:

$$G = ( VT, VN, S, P )$$

Donde:

VT = Conjunto finito de símbolos terminales

VN = Conjunto finito de símbolos no terminales

S = Axioma o símbolo inicial, perteneciente a N

P = Conjunto de reglas de producción / derivación

Las cadenas del lenguaje son todas pertenecientes al conjunto de símbolos del vocabulario de terminales. Sin embargo, los símbolos del vocabulario no terminales, son creados para ayudar en la definición de la gramática.

La intersección de ambos conjuntos es vacía:  $\{T\} \cap \{N\} = \{\emptyset\}$

Y a la unión se le llama *vocabulario*:  $\{N\} \cup \{T\} = \{V\}$

El axioma S, es un símbolo perteneciente al conjunto de símbolos no terminales, a partir del cual se empiezan a aplicar las reglas de producción de la gramática.

Las reglas de producción (P) nos permiten obtener a partir del axioma, el conjunto de cadenas del lenguaje. La estructura de una regla de producción en una gramática libre de contexto como la nuestra es:

$$A \rightarrow b$$

Donde A es un símbolo no terminal, y b es una cadena o secuencia de símbolos terminales y no terminales.

**Gramática libre de contexto** es aquella en la que tomando en atención las reglas de producción, A puede ser siempre sustituido por b independientemente del contexto (de ahí su nombre). Por lo que se exige que en el lado izquierdo de la definición de la regla sólo exista un símbolo no terminal.

Estas gramáticas normalmente se escriben usando representación BNF (Backus-Naur Form), que a modo de resumen, sigue las siguientes normas:

- Los símbolos no terminales se representan entre paréntesis angulares  $\langle \rangle$
- Los símbolos terminales se representan directamente como cadena de caracteres, sin paréntesis angulares.
- El símbolo  $::=$  sustituye a la flecha  $\rightarrow$  y puede interpretarse como “se define como”
- Si tenemos varias reglas con el mismo lado izquierdo, se pueden agrupar usando el símbolo  $|$  que hace función de OR.

Usaremos dicho lenguaje para definir la gramática de Sib más adelante.

Un **árbol de derivación** es una representación gráfica de como se puede generar o derivar cualquier cadena del lenguaje a partir del axioma.

Algunas de las características que cumple son:

- El árbol tiene una raíz, que será el axioma de la gramática.
- Los nodos intermedios serán símbolos no terminales.
- Los nodos hojas, pueden ser terminales o no terminales.

## 4.2 Definición de la gramática Sib

Basándonos en los conceptos antes mencionados de gramática, se creará una gramática libre de contexto para definir e implementar nuestro lenguaje Sib.

Para las definiciones de las reglas de producción usaremos BNF como lenguaje de representación.

### Símbolos Terminales

BEGIN, END, IF, ENDIF, ELSE, WHILE, ENDWHILE, ;, \_#, \_@, \_b, \_, +, -, \*, /, %, =, ==, !=, <=, >=, ., ', int, float, frac, nfrac, string, array, clef, step, object, note, partiture, ", \$, (, ), staccato, staccatissimo, marcato, tenuto, accent, A, B, C, D, E, F, G, S, {, }, [, ], trans, play, read, playPartiture, readPartiture

### Símbolos No Terminales

<package>, <mas\_packages>, <declaraciones>, <órdenes>, <declaración>, <sig\_declaración>, <tipo>, <lista\_variables>, <variable>, <sig\_lista\_variables>, <str\_ident>, <letra>, <palabra>, <letraN>, <resto\_palabra>, <orden>, <sig\_ordenes>, <condición>, <bucle\_while>, <asignación>, <operador>, <comparación>, <sig\_condición>, <comp\_operador>, <comp\_condición>, <tipo\_números>, <step>, <operación>, <número\_entero>, <número\_real>, <int\_frac>, <número>, <numeros>, <sig\_numeros>, <fracción>, <espacios>, <origen\_asignacion>, <expresión\_arit>, <clef\_value>, <cadena>, <mas\_palabras>, <operador\_arit>, <exp\_arit>, <mas\_espacios>, <comentarios>, <comentario\_simple>, <comentario\_compuesto>

### Axioma

Nuestro axioma será el símbolo no terminal <programa>

### Reglas de producción

// Programa

<programa> → <package><imports>BEGIN <str\_ident> <instrucciones> END

### *// Package*

<package> → PACKAGE <str\_ident><mas\_packages>;  
<mas\_packages> → .<str\_ident><mas\_packages>  
|  $\epsilon$

### *// Importación de tipos*

<imports> → <import\_type><imports>  
|  $\epsilon$   
<import\_type> → IMPORT <str\_ident><mas\_import>;  
<mas\_import> → .<str\_ident><mas\_import>  
|  $\epsilon$

### *// Instrucciones*

<instrucciones> → <instruccion><instrucciones>  
|  $\epsilon$   
<instruccion> → <comentario>  
| <declaracion>  
| <orden>

### *// Declaraciones*

<declaración> → <tipo> <lista\_variables>;  
  
<tipo> → <tipo\_básico>  
| <tipo\_extendido>  
  
<tipo\_básico> → int  
| float  
| frac

		nfrac
		string
		step
		clef
		object
		array(<numero_entero>)
<tipo_extendido>	→	note
		partiture
<lista_variables>	→	<variable><sig_lista_variables>;
<sig_lista_variables>	→	,<variable><sig_lista_variables>
		ε

#### *// Variables*

<variable>	→	\$<str_ident>
		\$<str_ident>.<str_ident>
<str_ident>	→	<letra>
		<letra><palabra>
<letra>	→	A   B   C   ...   Z   a   b   c   ...   z
<palabra>	→	<letraN><resto_palabra>
<resto_palabra>	→	<letraN><resto_palabra>
		ε
<letraN>	→	A   B   C   ...   Z   a   b   ...   z   0   1   ...   9

#### *// Órdenes*

<orden>	→	<operador>;
---------	---	-------------

```
| <asignación>;
| <condicional_if>;
| <bucle_while>;
| <funcion>;
```

*@todo Voy por aqui comprobando las sentencias con respecto a la versión actual de jflex y cup*

```
// *****
```

*// Selector if*

```
<condicional_if>    →    IF (<comparación>) <instrucciones> <sig_condición_if>
<sig_condición_if> →    ENDIF
                    | ELSE <instrucciones> ENDIF
<comparación>      →    <origen_asignacion> <comp_condición> <origen_asignacion>
/*
```

```
<comp_operador>    →    <variable>
                    | <tipo_números>
                    | <step>
                    | <clef_value>
                    | <operación>
```

```
*/
```

```
<tipo_números>     →    <número_entero> | <número_real> | <int_frac>
<número_entero>    →    [-]?<número><numeros>
<numeros>          →    <numero><sig_numeros>
<sig_numeros>      →    <numero><sig_numeros> | ε
<número>           →    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<número_real>      →    <número_entero>.<numeros>
```



<int\_frac> → [-]?<fracción>  
 | <numero\_entero><espacios><fracción>

<fracción> → <numero\_entero>/<numero\_entero>

<step> → A | B | C | D | E | F | G | S

<comp\_condición> → == | < | > | >= | <= | !=

*// Bucle while*

<bucle\_while> → WHILE ( <comparación> ) <instrucciones> ENDWHILE

*// Asignación*

<asignación> → <variable> = <origen\_asignacion>

<origen\_asignacion> → <expresión\_arit>  
 | <step>  
 | <operación>  
 | <clef\_value>  
 | <cadena>  
 | <tipo\_numeros>  
 | <accent\_value>  
 | <variable>

<cadena> → \"<espacios>?<palabra><mas\_palabras><espacios>?\"

<mas\_palabras> → <espacios><palabra><mas\_palabras>  
 | ε

<exp\_aritmetica> → <tipo\_numeros>  
 | <variable>  
 | <exp\_aritmetica><operador\_arit><exp\_aritmetica>

/\*

<expresión\_arit> → (<expresión\_arit> <operador\_arit>  
<expresión\_arit>)<exp\_arit>

| <tipo\_números><exp\_arit>

| <variable><exp\_arit>

<exp\_arit> → <operador\_arit><expresión\_arit><exp\_arit>

|  $\epsilon$

\*/

<operador\_arit> → + | - | \* | / | %

<clef\_value> → G2 | F[3-4] | C[1-4]

// Accent: [https://en.wikipedia.org/wiki/Accent\\_\(music\)](https://en.wikipedia.org/wiki/Accent_(music))

<accent\_value> → staccato | staccatissimo | marcato | tenuto | accent

// Quizás debamos tener algo como anotaciones

// Para calderón (fermata), ligaduras, etc ....

// <https://en.wikipedia.org/wiki/Fermata>

// Operaciones

<operacion> → | trans(<variable>,<tipo\_números>)

| trans(<operacion>,<tipo\_números>)

| trans(<operador>,<tipo\_números>)

/\*

*Parece que reduce también por expresión\_arit*

| <variable><operador\_arit\_nota><tipo\_numeros>

<operador\_arit\_nota> → + | - | \* | /

*\*/*

### *// Operadores*

$\langle \text{operador} \rangle \rightarrow \langle \text{variable} \rangle \langle \text{operador\_nota} \rangle$   
 $| \langle \text{operador} \rangle \langle \text{operador\_nota} \rangle$

$\langle \text{operador\_nota} \rangle \rightarrow \_ \# | \_ \# \# | \_ \text{b} | \_ \text{bb} | \_ . | \_ . . | \_ \dots$

### *// Funciones*

$\langle \text{funcion} \rangle \rightarrow \text{play}( \langle \text{variable} \rangle )$

### *// Espacios*

$\langle \text{espacios} \rangle \rightarrow \langle \text{espacio} \rangle \langle \text{mas\_espacios} \rangle$

$\langle \text{mas\_espacios} \rangle \rightarrow \langle \text{espacio} \rangle \langle \text{mas\_espacios} \rangle$   
 $| \epsilon$

$\langle \text{espacio} \rangle \rightarrow \backslash \text{s} | \backslash \text{t}$

### *// Comentarios*

$\langle \text{comentarios} \rangle \rightarrow \langle \text{comentario\_simple} \rangle \langle \text{comentarios} \rangle$   
 $| \langle \text{comentario\_compuesto} \rangle \langle \text{comentarios} \rangle$

$\langle \text{comentario\_simple} \rangle \rightarrow // \langle \text{espacios} \rangle \langle \text{palabra} \rangle \langle \text{mas\_palabras} \rangle$

$\langle \text{comentario\_compuesto} \rangle \rightarrow /* \langle \text{espacios} \rangle \langle \text{linea} \rangle \langle \text{mas\_lineas} \rangle */$   
 $| /* \langle \text{espacios} \rangle \langle \text{palabra} \rangle \langle \text{mas\_palabras} \rangle \langle \text{espacios} \rangle */$

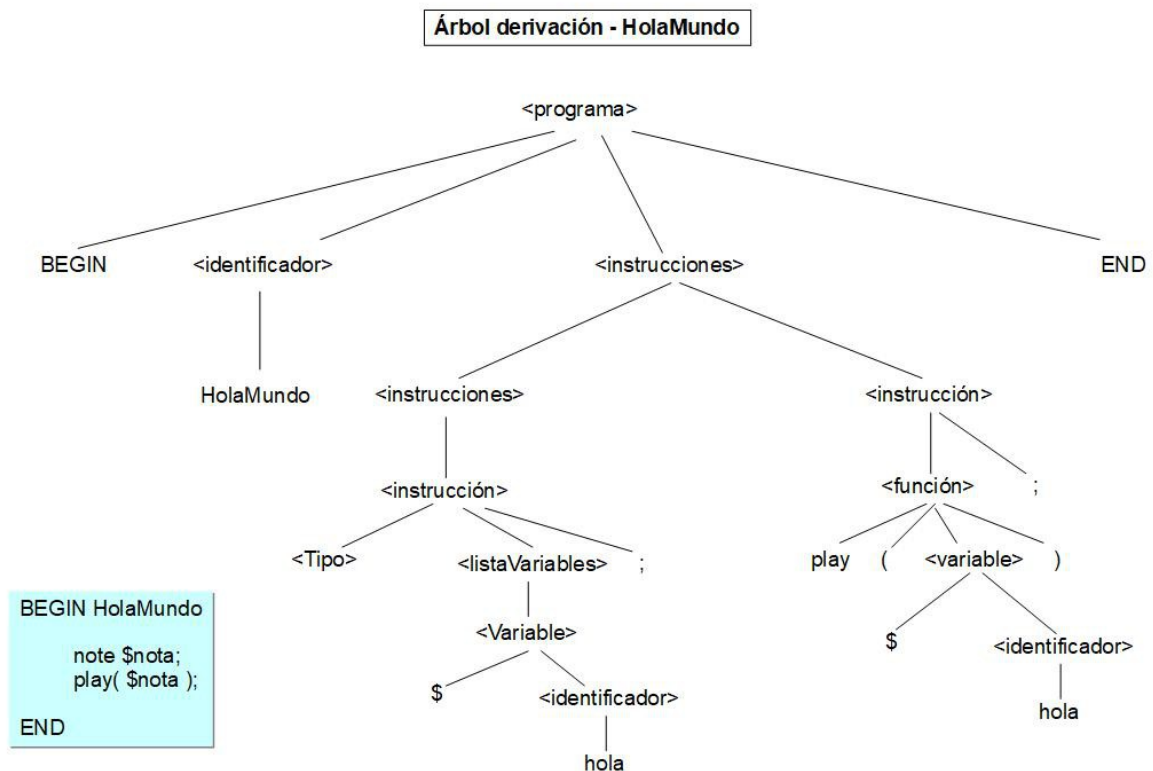
$\langle \text{linea} \rangle \rightarrow \langle \text{palabra} \rangle \langle \text{mas\_palabras} \rangle \backslash \text{n}$   
 $| \langle \text{espacios} \rangle \langle \text{palabra} \rangle \langle \text{mas\_palabras} \rangle \backslash \text{n}$

$\langle \text{mas\_lineas} \rangle \rightarrow \langle \text{linea} \rangle \langle \text{mas\_lineas} \rangle$   
 $| \epsilon$

## Ejemplo de árbol de derivación

Aunque no forme parte como tal de la definición de una gramática, se muestra un ejemplo de árbol de derivación, ya que es una forma de representación muy extendida, y ayuda a entender el funcionamiento de los analizadores.

El ejemplo se trata del programa HolaMundo.sib, un sencillo programa que muestra la nota DO en 4º octava en un pentagrama en clave de SOL y compás 4/4.



## Capítulo 5. MusicXML

MusicXML es un formato basado en XML para la representación de anotaciones musicales.

Se trata de un formato muy extendido hoy en día, tanto para la representación, como para el intercambio de anotaciones musicales entre aplicaciones.

Es por ello por lo que se decide usar este formato como lenguaje de salida de nuestro traductor Sib.

Actualmente se encuentra disponible su versión 3.1 en el repositorio público de Github

( <https://github.com/w3c/musicxml> )

A continuación se describirán la estructura y conceptos más importantes de MusicXML abarcados por el traductor, y que ayudarán a interpretar cómo trabaja y cómo son las salidas generadas.

### Cabecera

Inicialmente tenemos la cabecera XML específica de dicho formato, indicando versión, codificación, y posible DTD para su validación.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

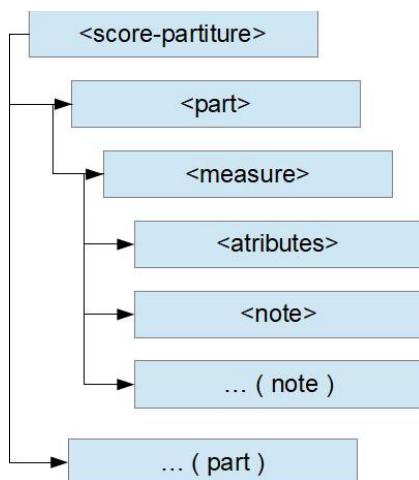
```
<!DOCTYPE score-partwise PUBLIC
```

```
"-//Recordare//DTD MusicXML 3.0 Partwise//EN"
```

```
"http://www.musicxml.org/dtds/partwise.dtd">
```

## Cuerpo

A continuación en el cuerpo, disponemos de un elemento raíz: `<score-partwise>`, que se compondrá de partes ( `<part>` ) y estas a su vez, de compases o medidas ( `<measure>` ), donde se encuentran las distintas notas. Una representación en árbol sería:



### `<score-partwise>`

Se trata de un elemento raíz, que contendrá una o más partes ( `<part>` ).

Como atributo tiene *version*, no obligatorio.

### `<part>`

Representa partes de la obra, formada por uno o más compases ( `<measure>` )

Como atributo tiene *id*, obligatorio.

### `<measure>`

Measure representa a cada uno de los compases que componen una partitura.

Está compuesto por unos atributos ( `<attributes>` ), y una o más notas musicales ( `<note>` ).

*Number* es el atributo obligatorio que tiene.

### `<attributes>`

Entre los atributos asociados a un compás, destacamos:

`<divisions>` Indica cuántas divisiones se realiza por unidad de tiempo.

Nuestro traductor trabajará con divisiones de 16, es decir, precisión de

semifusas.

<key> → <fifths> Representa la armadura del compás, basado en el círculo de quintas.

<clef> → <sign> Indica el valor de la nota de la clave del compás.

<clef> → <line> Indica la línea de la nota de la clave del compás.

<time> → <beats> Indica el numerador del tiempo del compás. Cuantos tiempos hay por compás.

<time> → <beat-type> Indica el denominador del tiempo del compás. Qué nota equivale a un tiempo de compás.

### **<note>**

Representa al elemento más común en una partitura, las notas.

Haremos uso de los siguientes nodos hijos para representar su información:

<note> → <pitch> → <step> Tono de la nota.

<note> → <pitch> → <octave> Octava en la que se encuentra.

<note> → <pitch> → <alter> Indica si la nota tiene alguna alteración.

<note> → <duration> Duración de la nota en número de divisiones.

<note> → <dot> Una etiqueta <dot> por cada puntillo que aplique a la nota.

Un ejemplo de nodo `<measure>` es:

```
<measure>
  <attributes>
    <divisions>16</divisions>
    <key>
      <fifths>-3</fifths>
      <mode>minor</mode>
    </key>
    <time>
      <beats>3</beats>
      <beat-type>4</beat-type>
    </time>
  </attributes>
  <note>
    <pitch>
      <step>C</step>
      <octave>4</octave>
    </pitch>
    <duration>4</duration>
  </note>
</measure>
```

Esta es una representación básica tomada como referencia para la creación del lenguaje Sib versión 0.1, la cual puede ser extendida en funcionalidad, abarcando más etiquetas MusicXML.



## Capítulo 6. Conclusiones y mejoras

Tras la realización del proyecto se pueden obtener las siguientes conclusiones:

- El potencial del proyecto no sólo se queda en el nivel académico, donde se puede usar como método de aprendizaje en el diseño e implementación de nuevos lenguajes de programación y sus compiladores/traductores. Si no que puede ser usado como punto de inicio de un lenguaje musical, que permita desarrollar nuevas aplicaciones musicales desde el punto de vista de los músicos. Por su estructura y sencillez, permite extender nuevas funcionalidades de forma rápida y sencilla.
- Ante la pregunta de si ha resuelto el problema inicial, la respuesta es si. Con dicho lenguaje y su traductor, podemos transportar partituras de saxofón alto a saxofón tenor. Bien es cierto que al no estar cubierto todo el lenguaje MusicXML, por su extensión y complejidad, las características principales cubiertas permiten solucionar el problema casi en su totalidad. En la carpeta de ejemplos, hay uno llamado *AltoATenor.sib*, que realiza dicha transportación.

Referente a las mejoras, como se acaba de mencionar, el lenguaje MusicXML es muy extenso, y permite representar casi todo el lenguaje musical. Se han cubierto las características principales de dicho lenguaje, para poder trabajar con él, y obtener resultados legibles por software comerciales, pero queda a disposición de futuras versiones cubrir más características de MusicXML.

En los capítulos referentes al lenguaje Sib y al traductor, se han indicado mejoras referentes a dichas secciones, y que darán más potencial tanto al lenguaje como a su traductor.

## **Apéndice A: Introducción al lenguaje musical**

Se va a realizar una breve introducción a las nociones básicas de la música, orientando el contenido a los conceptos que se tratan a lo largo del proyecto.

### **Pentagrama**

La música se escribe o representa dentro de un sistema de líneas y espacios, más concretamente 5 líneas y 4 espacios, llamado Pentagrama.

Es en el pentagrama donde se representan los distintos signos musicales, tanto dentro, como por encima o debajo, añadiendo líneas adicionales en caso de hacer falta.

Algunos de los signos básicos son:

### **Clave**

Podría decirse que la clave indica la altura de las notas que representa el pentagrama. Teniendo en cuenta que las 7 notas musicales se repiten una y otra vez de más grave a más agudo, dando lugar a las distintas octavas, la clave nos ayuda a centrar dichas notas en el pentagrama, destacando entre sus posibles valores, la clave de SOL y la de FA entre las más usadas, siendo SOL para las notas más agudas y FA para las más graves.

### **Armadura**

Son un conjunto de alteraciones que se indican al principio del pentagrama, e indican que todas las notas de dicho valor sufrirán dicha alteración de forma permanente, lo cual evita tener que estar indicando las alteraciones cada vez que nos encontremos con una de estas notas. Si fuese necesario aplicar una alteración para una nota en concreto, pero no de forma permanente, disponemos de las alteraciones accidentales, que aplican sólo a dicha nota dentro del compás.

### **Alteraciones**

Son modificaciones sobre la tonalidad en una nota, aumentando o disminuyendo semitonos al valor natural de la nota.

Las alteraciones más comunes son:

Sostenido: Sube el tono de la nota un semitono. Su representación es: #

Bemol: Baja el tono de la nota un semitono. Su representación es: b

Doble sostenido: Sube el tono de la nota un tono completo, ya que equivale a aplicar dos sostenidos. Su representación es: x

Doble bemol: Baja el tono de la nota un tono completo, ya que equivale a aplicar dos bemoles. Su representación es: bb

Becadro: Anula las posibles alteraciones que haya en la nota. Su representación es:

Las alteraciones se pueden presentar o bien al comienzo del pentagrama, como parte de una armadura, lo cual indica que se aplicarán a todas las notas de dicho valor, o bien de forma accidental, junto a alguna nota, lo cual indica que aplica a todas las notas de dicho valor que se encuentren en ese compás.

### **Compás**

Son segmentos del pentagrama, separados por líneas verticales, que contienen notas y silencios musicales. Están compuestos un por número determinado de tiempos, y según su anotación podemos saber cuántos tiempos y qué duración tiene cada uno.

El número superior de la fracción indica el número de tiempos del compás, y el número inferior la clase de nota que corresponde a un tiempo. Por ejemplo, si tenemos 2/4, indica que el compás dura dos tiempos, donde cada tiempo es de duración de una negra.

Las duraciones de las notas son: redondas 1, blancas 1/2, negras 1/4, corcheas 1/8, semicorcheas 1/16, fusas 1/32 y semifusas 1/64.

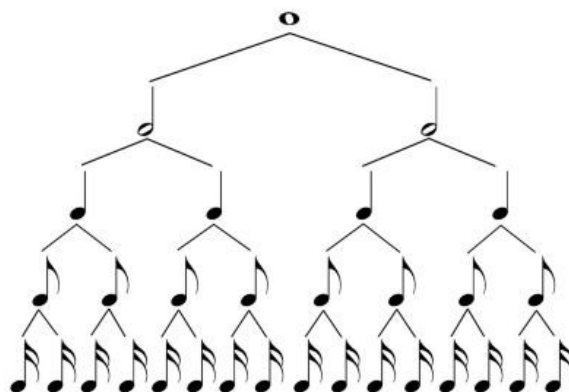
### **Tempo**

Indica la velocidad a la que la obra debe ser interpretada. Se suele indicar en pulsaciones por minuto (ppm)

### **Notas**

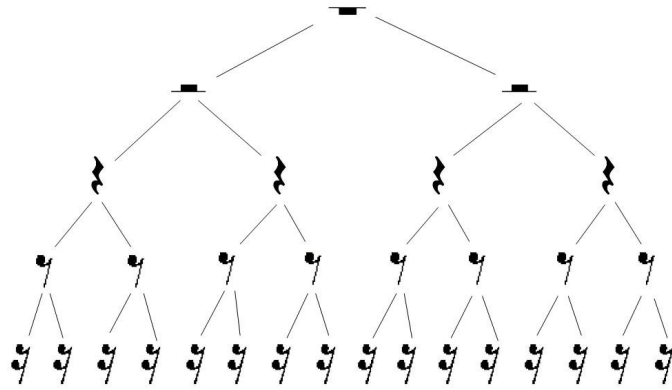
Empezamos por la notas, que se tratan de símbolos que indican la duración y la frecuencia con la que debe tocarse un determinado sonido. Es el elemento más común a la hora de escribir música.

Atendiendo a su duración, las notas se pueden clasificar en redondas, blancas, negras, corcheas y semicorcheas, existiendo otros valores de mayor y menor duración, pero menos frecuentes. En el esquema siguiente se pueden ver gráficamente cómo cada una dura la mitad de tiempo que su predecesora:



## Silencios

Indican duraciones de silencios o pausas en una obra. Al igual que las notas, existen distintas representaciones de los silencios en función de su duración. La correspondencia entre las duraciones de los silencios se muestra en el siguiente árbol:



## Apéndice B: JFlex & Java CUP

Para el desarrollo del traductor hemos usado JFlex y Java CUP para generar el analizador léxico y sintáctico respectivamente.

Veamos algunos conceptos básicos sobre JFlex y Java CUP, que nos ayudarán a entender mejor al funcionamiento del traductor Sib.

### ***B.1 JFlex***

Se trata de un generador de analizadores léxicos, basado en java.

Tras su ejecución, nos generará una clase .java que será el analizador léxico a llamar por nuestro traductor.

Para explicar la estructura de un fichero jflex, nos basaremos en el fichero que usamos en el traductor, el fichero *AnalizadorLexico.flex* del paquete *sib.flex*.

La estructura general de un programa jflex es:

*Código del usuario (java)*

%%

*Opciones JFlex y macros*

%%

*Reglas y acciones*

### **Código del usuario.**

En esta sección se realizan las importaciones, definiciones de código java necesarias, inicialización del código y demás tareas propias de cualquier programa java.

Este código será directamente copiado a la clase .java generada como analizador léxico.

En nuestro caso lo usamos para definir el paquete y las importaciones necesarias:

*package sib.flex;*

*import sib.cup.\*;*

## Opciones JFlex y macros

JFlex dispone de un gran número de directivas que definirán opciones y modos de actuar del generador. A continuación se muestran algunas de las más comunes y de las que hemos hecho uso.

*%public* – Define el ámbito de la clase resultante.

*%class* – Define el nombre de la clase resultante.

*%unicode* – Formato de caracteres aceptados.

*%column*, *%line* – Activa la contabilización de la columna y fila por la que va leyendo el analizador. Se activa para poder dar más información en caso de error.

*%cup* – Activa la compatibilidad con Java CUP.

*%yylexthrow* - Se lanzarán excepciones en caso de error.

*%type* - Indica el tipo de objeto que se devuelve como token.

*%init* - Código que se ejecutará en el constructor de la clase.

*%eof* - Código que se ejecutará al llegar al final del archivo de lectura.

*%{ ... %}* - Copia el código java en la clase resultante. Útil para definir atributos o métodos extras.

*%state* – Define los distintos estados que podemos tener.

A continuación se definen los macros, que nos permitirán tener el código más ordenado, y no tener que definir continuamente las mismas expresiones regulares en la sección de reglas.

Algunos ejemplo de macros definidas son:

$$\textit{Letra} = [A\text{-}Za\text{-}z]$$
$$\textit{LetraNumero} = [A\text{-}Za\text{-}z0\text{-}9]$$
$$\textit{Str\_ident} = \{\textit{Letra}\}\{\textit{LetraNumero}\}^*$$

## Reglas y acciones

La sección de reglas léxicas contiene un conjunto de expresiones regulares y acciones a ejecutar asociadas cuando el escáner del analizador se encuentra dicha expresión regular.

Como norma general para la detección de reglas, hay que tener en cuenta que el escáner siempre intenta coincidir con la regla más larga posible (la que consume más caracteres), y en caso de igual longitud, la ganará la regla que se encuentre más arriba en las declaraciones.

Por defecto, existe un estado inicial en el que se encuentra el escáner cuando empieza el proceso, y que no es necesario indicar si no existen más estados adicionales. Se trata del estado <YYINITIAL>. En nuestro caso, como definimos un estado adicional <CADENA> con la directiva %state, tenemos que indicar ambos estados, ya que según se encuentre en uno u otro con una determinada expresión regular, se actuará de una manera u otra.

Una regla de ejemplo de código (puede verse el fichero completo en el código aportado a la memoria, en el fichero AnalizadorLexico.flex):

```
{Str_ident} {  
    Token t = new Token( sym.IDENTIFICADOR, yycolumn, yyline+1, 0, yytext(),  
Token.STR_IDENT );  
    this._existenTokens = true;  
    return t;  
}
```

Como puede verse en el código, se devuelve al analizador sintáctico, que será el encargado de llamar al léxico para pedirle tokens, un objeto de tipo Token, que fue el tipo definido con la directiva %type.



## ***B.2 Java CUP***

Java CUP nos permite generar analizadores sintácticos LR(0), más en concreto LALR. El cual nos permitirá dar validez a nuestro código Sib basado en la gramática libre de contexto definida.

Para explicar la estructura de un fichero cup, nos basaremos en el fichero que usamos en el traductor, el fichero *AnalizadorSintactico.cup* del paquete *sib.cup*.

La estructura general de un programa cup es:

< Código java >

< Código del usuario para el parser >

< Declaración de terminales y no terminales para la gramática >

< Gramática >

### **Código java.**

Código java para la definición de paquetes e importaciones necesarias.

### **Código del usuario para el parser.**

Código java que será copia en la nueva clase del parser generada. Muy útil para definir atributos y métodos propios. Como ejemplo, parte del código usado por nuestro fichero CUP:

*parser code* {:

```
public TablaSimbolos tablaSimbolos;
```

```
public void setTablaSimbolos( TablaSimbolos ts ) {
```

```
    this.tablaSimbolos = ts;
```

```
}
```

```
:};
```

## **Declaración de terminales y no terminales para la gramática.**

A continuación se declara la lista de símbolos terminales y no terminales que serán usados en la declaración de la gramática.

Los terminales representan símbolos terminales de la gramática, y los no terminales, variables que representan reglas de producción.

En la declaración, se puede indicar la clase que devuelve al símbolo. En caso de no indicarse, devolverá de la clase `Symbol` propia de Java CUP (`java_cup.runtime.Symbol`).

En nuestro código haremos uso de clases propias para los símbolos no terminales, no siendo necesarias para los terminales. Así tenemos como ejemplo de definiciones:

*terminal IF, ENDIF, ELSE, COMPARADOR, WHILE, ENDWHILE;*

*non terminal Programa programa;*

## **Gramática.**

En esta sección se describe las reglas de la gramática, siguiendo la forma:

$$\begin{aligned} <no-terminal> ::= <terminal \mid no-terminal> \{ : /* Acciones */ : \} \\ & \mid <terminal \mid no-terminal> \{ : /* Acciones */ : \}; \end{aligned}$$

Usándose la primera regla como axioma de partida, salvo que se indique lo contrario con la instrucción: `start with non-terminal;`

Ejemplo de reglas usadas en el proyecto son:

```
bucle_while ::= WHILE:wh LPAREN:lp condicion:con RPAREN:rp instrucciones:ins  
ENDWHILE:endw { : RESULT = new InstWhile( con, ins ); : };  
  
condicion ::= valor_asignacion:oper1 COMPARADOR:cc valor_asignacion:oper2 { :  
RESULT = new Condicion( oper1, cc.toString(), oper2 );  
: };
```

En el ejemplo presentado, se puede ver que en ocasiones, tras los símbolos, hay dos puntos ( : ) seguidos de un identificador. Se trata de la definición de alias para los símbolos, y así poder usarlo en la zona de acciones.

En el apartado de Referencias, se encuentran enlaces con las páginas oficiales de JFlex y Java CUP, desde donde poder descargarlos, además de referencias a documentación y otros recursos.

## **Apéndice C: Entorno de desarrollo**

Se va a hacer un breve recorrido sobre las herramientas usadas para el desarrollo del proyecto:

### **Eclipse**

Como herramienta principal sobre la que desarrollar el código java del traductor de Sib a MusicXML, se ha usado Eclipse como entorno de desarrollo (IDE).

Entre las múltiples versiones, se ha usado la Neon 4.6.3, que aunque no es la última versión disponible, es muy estable ( <https://www.eclipse.org/neon/> )

### **JFlex y Java CUP**

Como se ha mencionado anteriormente, se han usado JFlex ( <http://jflex.de> ) y Java CUP ( <http://www2.cs.tum.edu/projects/cup/> ) como librerías para la generación de los analizadores léxicos y sintáctico.

Se ha optado por integrar la llamada a ambas librerías directamente desde eclipse, para evitar tener que estar alternando entre la consola del sistema y el IDE.

### **Control de versiones Git**

Se ha usado el sistema de control de versiones git, para poder trazar los cambios y poder recuperar código en caso de tener que volver a versiones anteriores.

Usando github como repositorio ( <https://github.com/eggemplo/sib> ), ha permitido que el CVS nos sirva tanto como para el control de versiones, como copia de seguridad en la nube, y así poder trabajar en varios equipos, siempre con las versiones actualizadas.

### **UML designer**

Para la generación de los diagramas UML, se ha usado UML Designer ( <http://www.umldesigner.org> ) un entorno de libre acceso basado en Eclipse, que permite el diseño de diagramas UML.

## **Musescore 2**

Software gratuito para la escritura de partituras ampliamente extendido en el mercado ( <https://musescore.org/es> ). Se ha usado dicho software tanto para probar los resultados que generaba el traductor a MusicXML, como para generar ficheros MusicXML que podemos importar y sobre los estudiar la estructura del formato MusicXML.

## **OpenOffice**

Para la realización de la memoria, se ha usado Openoffice ( <https://www.openoffice.org/es/> )

## Apéndice D: Código entregado

Junto a la memoria se entrega un CD con una copia electrónica de ella, y código fuente del traductor de Sib a MusicXML, además de otros ficheros de apoyo.

La estructura de ficheros y carpetas es:

**/memoria:** Incluye la memoria en formato PDF

**/src:** Código fuente del traductor de Sib a MusicXML. Los ficheros más reseñables de dicho código son:

*/sib/flex/AnalizadorLexico.flex:* Con el código jflex que dará lugar al analizador léxico.

*/sib/cup/AnalizadorSintactico.cup:* Con el código Java CUP, que dará lugar al analizador sintáctico.

**/ejecutable:** En dicha carpeta se encuentra el ejecutable .jar que lanzará el traductor Sib a MusicXML

**/ejemplos:** Carpeta con algunos ejemplos .sib que se pueden usar para probar el traductor.

## Referencias

- <http://www.creandopartituras.com/numeracion-de-octavas-y-notas-indices-acusticos/> : Información sobre octavas.
- <http://www.musicxml.com/> - Página oficial del lenguaje MusicXML
- <http://jflex.de> - Página oficial de JFlex
- <http://www2.cs.tum.edu/projects/cup/> - Página oficial de Java CUP
- Lenguajes de Programación, principios y paradigmas. A. Tucker y R. Noonan. Editorial McGraw Hill
- Teoría, diseño e implementación de Compiladores de Lenguajes. Francisco J. Martínez López y Alejandro Ramallo Martínez. Editorial Ra-Ma
- Java a tope: Traductores y compiladores con Lex/Yacc, Jflex/CUP y JavaCC. Sergio Gálvez Rojas y Miguel Ángel Mora Mata.
- Teoría musical para Dummies. Michael Pilhofer y Holly Day. Editorial Grupo Planeta