Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ciencias y Sistemas
Organización de lenguajes y compiladores 2
Segundo Semestre 2014

Catedráticos:
Ing. Byron López
Ing. Edgar Sabán

Auxiliares:
Pablo Velásquez, Rony Granados,
Luis Morales, Ricardo Cortéz

Primer Proyecto

Objetivo General

Aplicar los conceptos generales sobre teoría de compiladores aprendidos en la clase y el laboratorio realizando una aplicación práctica para dar solución a varios problemas involucrados en proyectos de sistemas informáticos.

Objetivos Específicos

- Hacer uso de canvas para la generación de grafos dinámicos.
- Hacer uso de la librería log4Net para el reporte y control de errores encontrados.
- Implementar la Herramienta Generadora de analizadores Irony para .Net
- Implementar la comunicación entre aplicaciones de escritorio y aplicaciones móviles.

Índice

Descripción

Actualmente y debido a la presencia de internet en casi cualquier dispositivo, la música está en todos lados formando parte de nuestra vida en cada momento y se presenta de maneras distintas, siendo esta personalizada y diferente para cada ocasión o circunstancia que nos encontremos. Se ve desde que nacemos hasta el final de nuestra vida.

Al hablar de música se abre un mundo bastante amplio y es fácil entender su naturaleza, pues cuenta de pocos parámetros o características que la definen, por ejemplo la frecuencia a la que se generan las ondas, definen las notas musicales las cuales perturban el ambiente o el medio, junto con el tiempo que duran estas mismas en el medio de comunicación realizando en combinación de cada unidad de perturbación en una frecuencia, en un tiempo señalado se interpreta como musical y que utiliza como medio natural el aire para poderla transmitir.

El proyecto consiste en construir un sistema en el cual por medio de un lenguaje de alto nivel se reciba fragmentos de canciones, de notas o clases de ritmos preestablecidos haciendo a este lenguaje de definición de canciones, pistas, ritmos, etc. bastante completo y extendido pudiéndose definir largas pistas musicales sin tanto código involucrado, de una manera clara definida por métodos locales del propio lenguaje y que se puede transmitir, a un sistema consumidor que pueda hacer uso de las mismas .

El sistema constará de una arquitectura de cliente servidor, para lo que consta de dos partes: un módulo servidor/local, que es quien recibe como entrada un lenguaje de alto nivel, el que es quien describe la creación de pistas y listas de reproducción, almacenamiento de canciones, así como otras funciones, procedente a ello envía estas al módulo consumidor/cliente, éste las reproducirá y tendrá una forma de crear sus propias canciones también. La manera de comunicarse de ambos módulos es por medio de sockets.

El módulo servidor/local deberá ser capaz de crear nuevas pistas, además de ordenarlas por listas de reproducción, las cuales podrán ser reproducidas dentro del mismo, mostrando una gráfica que permita identificar la frecuencia de cada nota contenida en la pista.

El módulo cliente se encargará de solicitar cada pista desde cada lista de reproducción definida en el módulo servidor, además de ello deberá ser capaz de reproducir cada una. Cabe mencionar que la creación de canciones propias se llevará a cabo en un apartado especial de la aplicación que contendrá un teclado musical integrado.

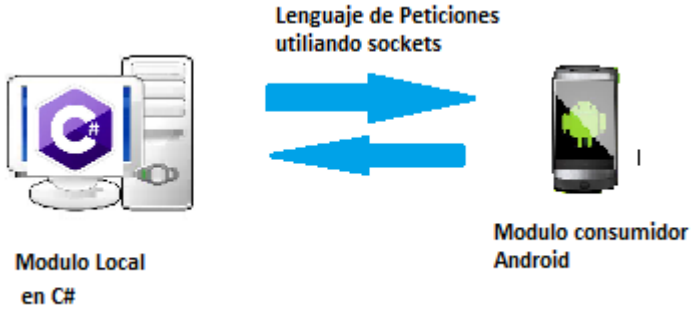
1) Comportamiento del Sistema

Interpretación:

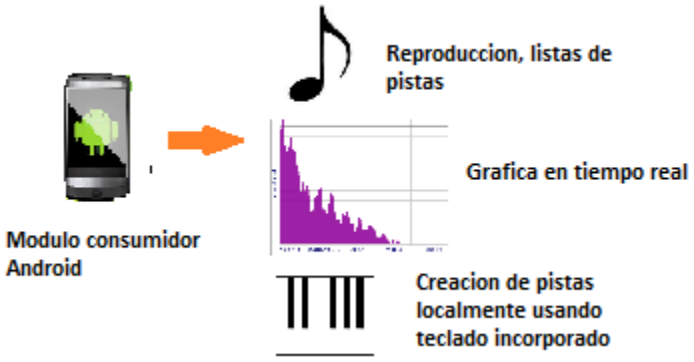
- El lenguaje de creación de pistas es un lenguaje de alto nivel (descrito posteriormente), este es introducido en el editor de texto del módulo local, este lenguaje es quien define entre su sintaxis, una lista de notas musicales con una frecuencia y duración específica, así como otros componentes que enriquecen su facilidad la definición de las mismas. Este módulo, además de reconocer el lenguaje de pistas, reconoce también un lenguaje de creación de listas de pistas (canciones), ambos lenguajes serán compilados por este módulo.

- Una vez ha sido compilada la entrada en el módulo local, se crearan ya sea una pista o una lista de reproducción. Toda pista y lista de pistas se almacenarán en un archivo binario a modo que cuando se crea una lista puedan ser incluidas pistas creadas anteriormente y poder disponer de su uso. Este módulo puede reproducir las canciones, además cuando se reproducen las canciones, este módulo mostrará una gráfica de frecuencias que muestra las notas y sus respectivas frecuencias durante el tiempo que se está ejecutando.

- Además de la parte del módulo local/servidor, se tiene un módulo de cliente/consumidor, este módulo lo que realiza son solicitudes al módulo local/servidor, para pedir pistas, listas y demás que han sido almacenadas en el módulo servidor. Estos dos módulos utilizan sockets para su comunicación, así como también utilizarán un lenguaje de peticiones (descrito posteriormente), por el que se harán las mismas del cliente al servidor.



- El módulo cliente/consumidor podrá reproducir y cargar las listas de pistas que sean enviadas desde el módulo servidor, las reproducirá y al igual que el módulo servidor, dispone la visualización de gráfica de frecuencias en tiempo real (Tanto para el módulo local y consumidor, la gráfica debe mostrar lo mismo, el ejemplo está más adelante), además el modulo consumidor tiene un teclado incorporado y con este la funcionalidad de crear pistas y listas directamente desde el propio módulo.



Grafo de Esquema para el funcionamiento del sistema.

Herramientas a Utilizar: Para la implementación de las partes del sistema se debe usar:

- Modulo local/Servidor: C# ,Irony para .NET
- Modulo Consumidor/Cliente: Java ,JLex/JFlex,CUP
- Comunicación: Sockets para Java y .NET
- Errores:Log4Net

2) Modulo Local (Servidor)

Este módulo es el encargado de dar mantenimiento a los sonidos, pistas, ritmos creados, etc. También se encarga de servirle al módulo exterior en brindarle la información necesaria para que funcione correctamente por lo que este tendrá las siguientes funciones.

2.1) Funciones del módulo:

- Crear, Modificar y Eliminar pistas.
- Servidor de pistas y listas de reproducción para el módulo externo por medio de sockets.
- Reproductor de pistas con visualizador de reproducción (Gráfica de frecuencias alcanzadas en tiempo real).
- Guardar y Recuperar Pistas.
- Crear, Modificar y Eliminar Listas de reproducción.

Crear Pista:
El usuario tendrá la opción de crear una nueva pista. Al hacer esto se tendrá que limpiar el área de edición del lenguaje de entrada y también detiene cualquier reproducción actual.

Modificar Pista:
Se deberá tener un área en donde se le mostrará al usuario alguna lista desplegable en donde se podrá visualizar las pistas existentes para que se puedan acceder fácilmente para su edición, esta lista debe actualizarse dinámicamente de acuerdo a la creación de nuevas pistas y la eliminación de estas.

Eliminar Pista:
Al seleccionar una pista en la lista visual el usuario puede eliminar dicha pista con solo oprimir el botón de eliminar pista dado a que el usuario no sabe exactamente donde se guardaran las pistas.

Guardar Pista:
El almacenamiento de las pistas se llevará a cabo en un archivo binario en donde se almacenarán todas las pistas definidas identificadas por su nombre, dicho archivo irá creciendo a medida que se vayan añadiendo más pistas o también al momento de ir añadiendo más cambios a estas pistas ya creadas.

Este archivo debe de ubicarse en el directorio, nombre y extensión siguiente:

C:\FistProjectOLC2\Tracks\file.bin

El almacenamiento del archivo en forma binaria tiene como objetivo que el usuario no tenga acceso directo a la forma en la que el sistema lleva el control de esto.

Al crear una nueva pista y guardarla puede darse el caso de que este nombre ya esté utilizado por lo que se le deberá preguntar al usuario si desea sobrescribir la pista actual.

Recuperar Pista:
Esto se hará al momento en el que el usuario seleccione una pista desde la lista visual del módulo que debe de tener todas las pistas almacenadas hasta el momento.

También se puede recuperar determinada pista desde el módulo consumidor a través de sockets y del lenguaje que se definirá posteriormente.

La recuperación de la pista también debe ser desde el archivo binario ya establecido por lo que un cambio en ese archivo también debe de reflejarse en dicha recuperación.

Guardar Listas de Reproducción:
El almacenamiento de las listas se llevará a cabo en un archivo binario de la misma forma que las pistas donde serán identificadas por su nombre, dicho archivo irá creciendo a medida que se vayan añadiendo más listas o también al momento de ir añadiendo más pistas a estas listas.

Este archivo debe de ubicarse en el directorio, nombre y extensión siguiente

C:\FistProjectOLC2\Lists\file.bin

Recuperar Listas de Reproducción:
El sistema también contará con una lista desplegable de listas de reproducción (similar a la lista de pistas) en donde deberán aparecer todas las listas almacenadas hasta el momento.

También se puede recuperar una determinada lista desde el módulo consumidor a través de sockets y del lenguaje que se definirá posteriormente.

La recuperación de la lista también debe ser desde el archivo binario ya establecido por lo que un cambio en ese archivo también debe de reflejarse en dicha recuperación.

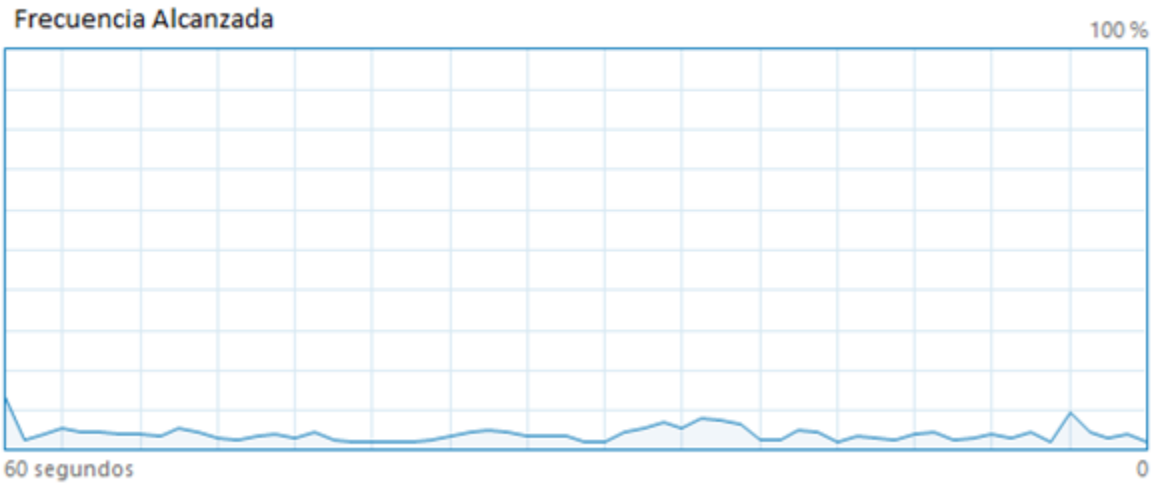
2.2) Reproductor de pistas con visualizador de reproducción (Gráfica de frecuencias alcanzadas en tiempo real):

Al momento de estar editando la pista o escribiéndola podemos reproducirla si el proceso de compilación resulta exitoso. El sistema debe de ser capaz de reproducir toda la entrada definida mostrando un menú con las opciones siguientes:

- Pausar
- Reanudar en caso de estar pausado
- Parar Reproducción

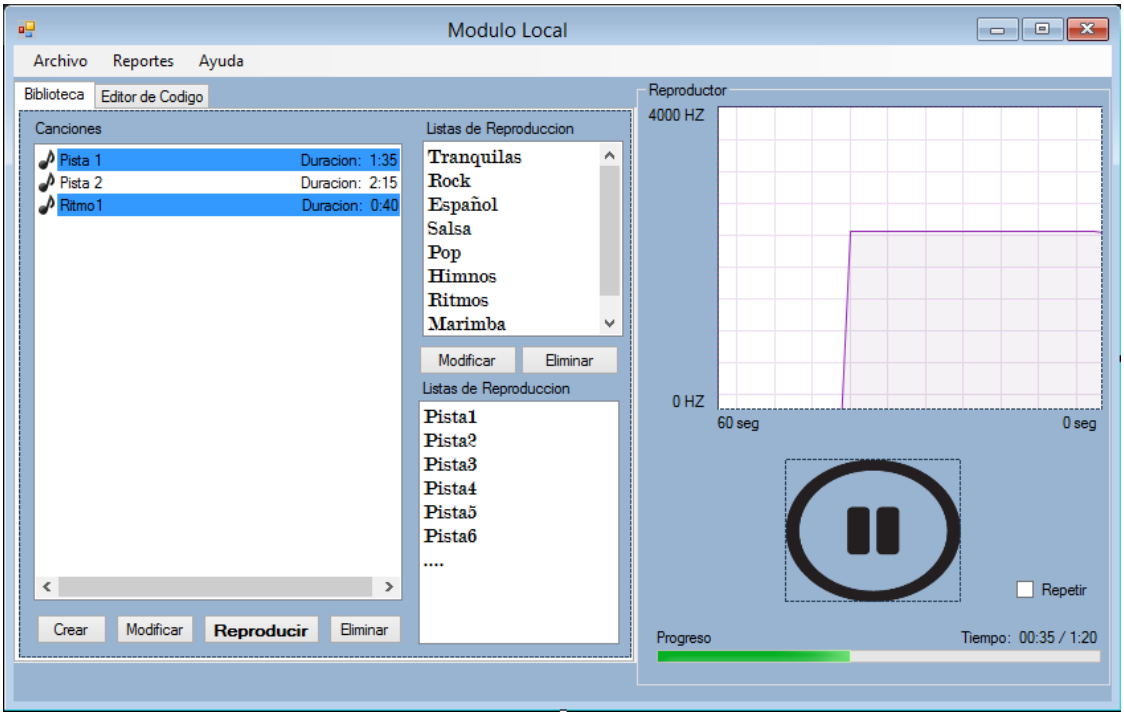
Además de estas opciones el usuario debe ver la gráfica (en tiempo real) de la frecuencia que se está alcanzando con la nota reproducida actualmente, esto debe de ir de acuerdo a la duración y la nota actual de la pista, por lo que si la pista dura 1 min 30 seg entonces debemos de observar la gráfica en un intervalo de 0:00 hasta 1:30 y los cambios deben verse reflejados cada medio segundo como máximo. Se debe tomar en cuenta que la gráfica debe reflejar las notas de cada canal de la pista siendo la gráfica de un solo canal diferente en color y distinguible de los otros canales.

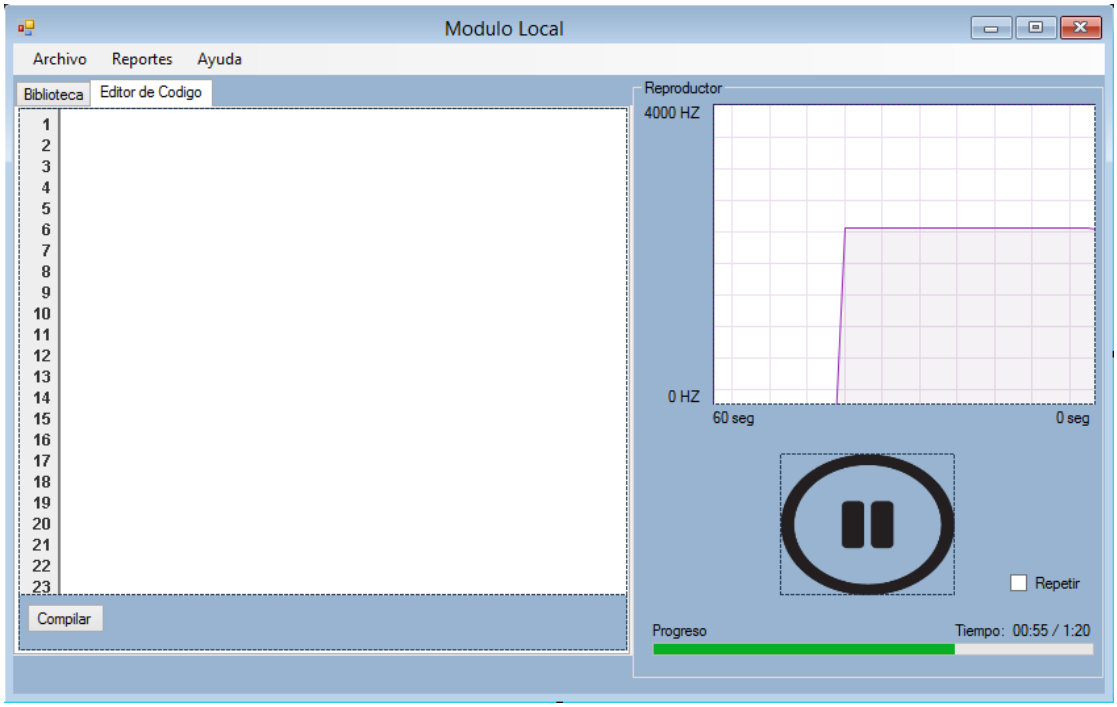
La gráfica debe de trasladarse como si fuera la gráfica de rendimiento de Windows, un ejemplo aproximado de cómo debe verse esto en pantalla sería el siguiente:



2.3) Aspecto General del módulo

El módulo servidor podría tener el siguiente aspecto





2.4) Colores y numeración de líneas en el área de edición:

El área de edición del módulo local debe estar totalmente identificada con reglas de colores según se establece en algunos IDEs actuales y se regirá por la siguiente tabla:

Color	Token
Azul	Palabras reservadas
Verde	Identificadores
Naranja	Cadenas, caracteres
Morado	Números
Gris	Comentario
Negro	Otro

Tabla 1:"Colores de Editor de Texto"

Además en el área de edición el módulo local dispondrá de numeración de líneas de edición por lo que se podrá localizar, la línea en la que se está editando actualmente y además esta parte podrá dar la ubicación para reportar errores de edición.

3) Lenguaje de Definición de Pistas (módulo local)

3.1) Case Sensitive:

El lenguaje será capaz de reconocer diferencia entre mayúsculas y minúsculas para todas las palabras que acepta el lenguaje por lo que si se declara un atributo de nombre “Contador” este será distinto a definir un atributo “contador”.

Para que no haya problema con las palabras reservadas, estas solo pueden venir en minúsculas o la primera letra en mayúsculas y todas las demás en minúsculas.

3.2) Comentarios

Comentario de una línea:

Estos deben empezar con dos signos de “mayor que” y terminar con un salto de línea:

```
>>comentario de una sola línea
```

Comentario de varias líneas:

Estos deben empezar con un signo de “menor que” y un guión y termina con un guión y un signo de “mayor que”:

```
<- comentario de varias líneas
Segunda línea
Tercera línea
....
->
```

3.3) Pista:

La sintaxis para la declaración de una pista es la siguiente, en donde debe de establecerse el nombre de la pista y también si extiende de otra pista. Extender de otra pista es opcional para cada pista que se vaya a crear.

```
Pista Nombre [Extiende1, Extiende2,.... ExtiendeN]
    //cuerpo de la pista
    .....
```

Ejemplo:

```
Pista FragmentoCoro
    //cuerpo de la pista


    .....
```

3.4) Extender una Pista:

Cada pista tendrá la posibilidad de poder extender atributos, métodos y funciones de otra pista indicando el nombre de la pista de la cual se desea obtener toda su estructura. Al momento de extender una pista, se deberá buscar la pista padre en el archivo binario de pistas cargadas y compiladas para poder extender la pista hija (función similar al “import” tradicional).

La extensión de una pista tiene las siguientes características:

1. El método principal de ejecución de pistas no será heredado de una pista a otra pista, por lo que la pista a extender puede tener su método principal de ejecución y la pista que extiende no tener este método. Al momento de reproducir dicha pista dará error por no tener método principal de ejecución dado a que este no existe o no fue definido.

2. Se puede definir la extensión de más de una pista, aunque esto puede traer el problema de que estas pistas posean métodos o funciones con el mismo nombre y que al extender no se pueda saber cuál tomar, la solución a esto es que la última función, método o variable declarada será la que permanezca al final si y solo si esta tiene la propiedad “Keep”, por lo que si la pista la cual se está definiendo tiene un método el cual fue heredado previamente de otra pista, esta sustituirá al método heredado solo si tiene la propiedad “Keep”, sino quedará el método, función o variable de la última Pista extensora.

Ejemplo:

```
Pista Coro extiende Ritmo, Fragmentos, etc
        //Sentencia 1
        //Sentencia 2
        //Ciclo 1
                //Sentencia 1
        …..
```

3.5) Indentación de sentencias:

Toda sentencia termina siempre con un salto de línea y el cuerpo de toda la pista deberá tener una jerarquía de indentación. Por lo que dentro de la pista el cuerpo de esta debe de tener un nivel más a la derecha de indentación. Al igual que en los ciclos esta jerarquía debe de mantenerse si esto no se cumple se tomará como un error sintáctico

Ejemplo:

```
//Sentencia 1
//Sentencia 2
//Ciclo 1
        //Sentencia 1
        //Sentencia 2
        //Ciclo 1_1
                //Sentencia 1
                //Sentencia 2
                …..
        //Sentencia 3
        ….
//Sentencia 3
……
```

3.6) Tipos Aceptados

Los tipos que el lenguaje deberá soportar son los siguientes:

Nombre	Descripción	Ejemplos	Observaciones
entero	Este tipo de dato acepta solamente números enteros	1, 50, 100, 255125, etc.	No tiene límite de tamaño
doble	Es un entero con decimal	1.2, 50.23, 00.34, etc.	Se manejara como regla general el manejo de 6 decimales
boolean	Admite valores de verdadero o falso, y variantes de los mismos	Verdadero, falso, true, false. 1=verdadero, 0=falso	si se asigna los enteros 1 o 0 debe aceptar como verdadero o falso según el ejemplo
caracter	Solo admite un carácter por lo que no será posible ingresar cadenas enteras viene encerrado en comilla simple entre **a-z, A-Z, -, _, :** incluyendo símbolos y caracteres de escape #t, #n	'a', 'b', 'c', 'E', 'Z', '1', '2', '#t', '#n', '#', '%', '$', '##', ')', '=', '!', '"', '&', '/', '\', '.', etc	En el caso de querer escribir comilla simple se escribirá primero # y después la comilla simple '#'', además si se quiere escribir # se pondrá 2 veces '##', los caracteres de escape se escriben siempre '#t' y' #n'
cadena	Este es un conjunto de caracteres que pueden ir separados por espacios en blanco encerrados en comilla doble, los posibles caracteres a ingresar serán los mismos que los del tipo caracter solo que agregando a estos el espacio	"cadena1" "est#$%o es una ca&/(de=na #n#n!!!"	En el caso de querer escribir comilla doble se escribirá primero # y después la comilla doble **"ho#"la mundo #""**, además si se quiere escribir # se pondrá 2 veces'**##**', por ejemplo **"ho#la mundo ##"**, los caracteres de escape se escriben siempre #t y #n

Tabla 2:"Tipos aceptados por módulo local"

Los caracteres de escape llevarán el signo # antes de su literal o signo especial por ejemplo para agregar un salto de línea en una cadena seria de la siguiente forma:

"Esto es una linea#n".

3.7) Operadores Relacionales

Son los símbolos utilizados en expresiones, su finalidad es comparar expresiones entre sí dando como resultado booleanos. En el lenguaje serán soportados los siguientes:

Nombre	Símbolo	Ejemplos	Descripción
Igualación	==	1==1, “hola”==”hola”, 25.2555==80.0051	Compara ambos valores y verifica si son iguales iguales=true no iguales=false
Diferenciación	!=	1!=2, martes!=vari1	Compara ambos lados y verifica si son distintos distintos=true iguales=false
Mayor que	>	(5+5.5)>8.98	Compara ambos lados y verifica si el izquierdo es mayor que el derecho izquierdo mayor=true izquierdo no mayor =false
Menor que	<	(5/(5+5))<(8*8)	Compara ambos lados y verifica si el derecho es mayor que el izquierdo derecho mayor=true derecho no mayor =false
Mayor o igual que	>=	5-6>=4+6	Compara ambos lados y verifica si el izquierdo es mayor o igual que el derecho izquierdo mayor o igual=true izquierdo no mayor o igual =false
Menor o igual que	<=	55+66<=44	Compara ambos lados y verifica si el derecho es mayor o igual que el izquierdo derecho mayor o igual=true derecho no mayor o igual =false
Es nulo	!¡	(!¡cad1)	Verifica si la variable fue declarada pero no le fue asignado algún valor inicial o una asignación posterior

Tabla 3:”Operadores Relacionales aceptados módulo local”

Nota:

El nivel de precedencia para todos los operadores es igual, a excepción del operador es nulo que su importancia es mayor, se debe considerar al momento de operar la expresión, así como la asociatividad por la izquierda, que también es la excepción el operador es nulo.

3.8) Operadores Lógicos:

Son símbolos para poder realizar comparaciones a nivel lógico de tipo falso y verdadero, sobre expresiones

Nombre	Símbolo	Evento	Ejemplo	Observaciones
AND	&&	Compara expresiones lógicas y si son ambas verdaderas entonces devuelve =verdadero en otro caso retorna =falso	(flag1) && (“hola” == ”hola”) resultado=falso	flag1=falso
NAND	!&&	Compara expresiones lógicas y si son ambas verdaderas entonces devuelve =falso en otro caso retorna =verdadero	(flag2) !&& (“hola”==”hola”) resultado=falso	flag2=verdadero
OR	||	Compara expresiones lógicas y si al menos una es verdadera entonces devuelve=verdadero en otro caso retorna=falso	(55.5<4) || (!¡bandera) resultado =verdadero	bandera=nulo
NOR	!||	Compara expresiones lógicas y si al menos una es verdadera entonces devuelve=falso en otro caso retorna=verdadero	(band1) !|| (!¡bandera) resultado =verdadero	band1=falso bandera= no nulo
XOR	&|	Compara expresiones lógicas si al menos una es verdadera, pero no ambas entonces devuelve=verdadero en otro caso retorna=falso	(44+6!=4) &| (1+2>60) resultado =verdadero	
NOT	!	Devuelve el valor inverso de una expresión lógica si esta es verdadera entonces devolverá = falso, de lo contrario retorna =verdadero	!(!¡ais) resultado=falso	ais=nulo

Tabla 4:”Operadores lógicos aceptados por módulo local”

Para la precedencia de análisis y operación de las expresiones lógicas tenemos:

Nivel	Operador	Asociatividad
0	NOT	Derecha
1	AND, NAND	Izquierda
2	OR,NOR,XOR	Izquierda

Tabla 5:"Procedencia de operadores lógicos aceptados por módulo local"

En donde 0 es el nivel de mayor importancia.

3.9) Operadores Aritméticos:

Son símbolos para poder realizar operaciones de tipo aritmética sobre las expresiones en las que se incluya estos mismos.

Nombre	Símbolo	Evento	Ejemplo	Observaciones
Suma	+	Realiza una suma aritmética sobre las expresiones de ambos lados y retorna su resultado	“cola”+”cola” 25.555+3+alfa “hola mund” + ’o’ +’ V’	variable **alfa** es de tipo flotante
Resta	-	Realiza la resta aritmética sobre las expresiones de ambos lados y retorna su resultado	80-20 25-0.555 a-b-6	a y b son decimales
Multiplicación	*	Realiza la multiplicación aritmética sobre las expresiones de ambos lados y retorna su resultado	a*b*5*5.25 25.5*a	a y b son enteros
División	/	Realiza la división aritmética sobre las expresiones de ambos lados y retorna su resultado *dividendo / divisor*	5/5 a/b 5.25/0.22	a y b son flotantes Se pondrán restricciones de divisiones entre 0
Modulo	%	Realiza la división sobre las expresiones de ambos lados y devuelve el residuo de esta división *dividendo % divisor*	a%4 25+1%25+1	a es entero Esta solo de debe realizar entre enteros
Exponencial	^	Realiza una potenciación y devuelve el resultado de esta misma *base % exponente*	2^a 85.2^4	a en entero El exponente debe ser entero

Tabla 6:”Operadores Aritméticos aceptados por módulo local”

Para la precedencia de análisis y operación de las expresiones aritméticas tenemos:

Nivel	Operador
0	Exponencial
1	Modulo, División, Multiplicación
2	Suma, Resta

Tabla 7:"Precedencia de Operadores Aritméticos aceptados por módulo local"

En donde 0 es el nivel de mayor importancia. La asociatividad es por la izquierda para estos operadores.

3.10) Declaración y Asignación de Variables

La declaración puede hacerse desde cualquier parte del código ingresado, pudiendo declararlas desde ciclos, métodos, funciones o fuera de estos siempre dentro de una **Pista**. La forma normal en que se declaran las variables es la siguiente:

3.10.1) Declaración:

[Keep] var **Tipo nombre[,nombre2,nombre3…. , nombre" n"] [Asignación]**
Lo que está encerrado en corchetes es opcional puede no venir

Ejemplo:

```
var entero numero1
Keep var cadena cad1,cad2,cad3,titulo,artista
```

En este caso las variables no tienen valor alguno pues se declaró pero no se le ha asignado valor, además cad1, cad2, cad3, título y artista son tipo cadena.

```
Keep var doble contador = 50.5
var cadena palabra2,palabra3 = "esto es un ejemplo :D #n"+'v'
var boolean bandera2,bandera3=true
Keep var boolean flag2,flag3=!¡contador
var boolean flag4=flag2
Keep var caracter letra='A'
```

Para las variables declaradas en un mismo lugar separadas por coma y que se les asigna un valor, todas asignaran el mismo valor.

3.10.2) Asignación:

Se deberá validar que la variable exista y que el valor sea del tipo correcto
Nombre Asignación

Ejemplo:

```
numero1 = 25                //entero entero
numero1=true                //entero boolean
contador = 50.5*20+115*b
```

Con la asignación el casteo correspondiente para que la integridad del tipo de dato de la variable se mantenga, se debe realizar de acuerdo con la siguiente tabla:

Tipo Variable	Tipo Valor	Resultado de Casteo
entero	entero	entero
entero	cadena	error
entero	boolean	entero
entero	doble	entero
entero	caracter	entero acii
cadena	<cualquier tipo>	cadena
boolean	<cualquier tipo>	error
doble	entero	doble
doble	doble	doble
doble	cadena	error
doble	boolean	error
caracter	entero	ASCII
caracter	caracter	caracter
caracter	<cualquier otro tipo>	error

Tabla 8:"Casteos permitidos en módulo local"

3.11) Asignación de símbolos de operaciones simplificadas:

Con la finalidad de poder realizar la simplificación de asignaciones de valores y el trabajo sobre ellas , el lenguaje podrá soportar operadores de operaciones simplificadas

Nombre	Símbolo	Descripción	Ejemplo
Suma simplificada	+=	Realiza una suma con la variable que está al lado izquierdo y la expresión que está del lado derecho. También se utilizará para concatenar cadenas.	var1+=25 En este caso le suma lo que tiene la variable “var1” más 25 y se lo asigna a la misma variable “var1”
Suma en 1 simplificada	++	Realiza una suma en 1 la variable que está al lado izquierdo	var2++ En este caso le suma lo que tiene la variable “var2” más 1 y se lo asigna a la misma variable “var2”
Resta en 1 simplificada	--	Realiza una resta de 1 a la variable que está al lado izquierdo	var3 -- En este caso le resta 1a lo que tiene la variable “var3” y se lo asigna a la misma variable “var3”

Tabla 9:”Tabla operaciones simplificadas aceptadas módulo local”

Aparte del uso mostrado para los operadores ++ y -- se podrán usar sobre las variables para modificar su valor en expresiones según la tabla:

Símbolo	**Operación**	**Descripción**
++ var	pre incremento	Se incrementa en 1 la variable y posteriormente se usa
-- var	pre decremento	Se decrementa en 1 la variable y posteriormente se usa
var++	post incremento	Se usa y posteriormente se incrementa en 1 la variable
var--	post decremento	Se usa y posteriormente se decrementa en 1 la variable

Tabla 10:”Operadores simplificadas en 1 aceptadas por módulo local”

Ejemplo:

```
var1=1
var2=2
var2= var1++  // valores actuales  var2= 1, var1=2
var1= --var2;   // valores actuales  var1= 0, var2=0
```

3.12) Manejo de Arreglos

Se pueden realizar declaraciones de arreglos dentro del lenguaje de la aplicación, por lo que la manera de declaración y asignación es parecida a la de variables y es la siguiente:

3.12.1) Declaración de arreglos:

[Keep]var Tipo arreglo <nombre> [,nombre2,nombre3…. , nombre" n"] <dimensiones> [= Asignación]

Lo que está encerrado entre corchetes es opcional, y lo que está dentro de los tags es arbitrario

Dimensiones: Las dimensiones será de manera "N" dimensionales, por lo que no se limita al arreglo a un número específico de dimensiones el mínimo serán de 1 dimensión.

[<expresion>][<expresion>].....[<expresion>]

dimension 1 dimension 2 … dimension "n"

Ejemplo:

```
var entero arreglo arr1 [a+b+(a+(5+1)%5)]                          //con 1 dimensión
Keep var entero arreglo arr2,arr3,arr4  [5][5*5+5-4]               //con 2 dimensiones
var entero arreglo arr5[arr1[1-1]][(arr1[5+arr2[0][4]]/5)+1][1][8][15]  //con 5 dimensiones
```

También permitirá asignar valores al arreglo cuando se declare, si no se asigna un valor todas las casillas comienzan con valor nulo o que no tienen valor asignado. Para poder asignar al momento de inicializar se utilizara las llaves "{""}" para indicar los arreglos para cada dimensión y se separa con coma por cada posición o casilla del arreglo.

1 dimension {<expresion>,<expresion>, ……,<expresion>}

2 dimensiones {{<expresion>,...,<expresion>},{<expresion>,…,<expresion>}}

n dimensiones
{{<expresion>,...,<expresion>},{<expresion>,…,<expresion>},...,{<expresion>,...,<expresion>}}

Ejemplo:

var entero arreglo arr0 [3]={5,10,15} //En este caso arr0 tiene en la primera posición 5,10 en la segunda y 15 en la tercera.

var entero arreglo arr10,arr20,arr30 [2][3]={{5,10,15},{20,25,30}} // en este caso arr10, arr20 y arr30 poseen una matriz de 2*3 por lo que en cada posición del arreglo de 2 tiene 3 elementos.

Nota: lo que puede ir en cada casilla para definir su valor es una expresión.

3.12.2) Asignación de arreglos

Una vez definido el arreglo se puede asignar valor a cada posición del arreglo por lo que solamente se debe especificar la posición:

Estructura:

<Nombre> <Dimensiones> = <expresión>

Ejemplo:

arr0[2] = 5+5 // al arreglo 0 en la posición 2 se le asigna el valor 10
arr1 [(a+(5+1)%5)] = arr0[2]+5*8+b
arr4 [5+arr1[5]] [5*arr1[4]+5*arr1[5]-4] = arr1[(a+5)%2]

Nota: Si el resultado de la expresión excede el tamaño de la dimensión, deberá mostrarse como un error semántico.

3.13) Ciclos y Bifurcaciones

3.13.1) Si:

Esta sentencia comprobará la condición y ejecutará su contenido en caso de que esta sea verdadera.

La estructura de una sentencia “si” es la siguiente:

```
si(condición)
        // Sentencia 1
        // Sentencia 2
        // Sentencia 3
        …
        // Sentencia n
```

También puede utilizarse la palabra **“sino”** para ejecutar otro contenido en caso de que la condición sea falsa, de la siguiente manera:

```
si(condición)
        // Sentencia 1
        // Sentencia 2
        // Sentencia 3
        …
        // Sentencia n
sino
        // Sentencia 1
        // Sentencia 2
        // Sentencia 3
        …
        // Sentencia n
```

3.13.2) Switch:

Esta sentencia de control evaluará una variable y ejecutará un contenido dependiendo del valor de ésta, así como también podrá establecerse un contenido **“default”** por si la variable no contiene ningún valor de los establecidos previamente.

La estructura de una sentencia de control “switch” es la siguiente:

```
switch(variable)
        caso valor1
                // Sentencias caso 1
                [salir]
        caso valor 2
                // Sentencias caso 2
                [salir]
        caso valor 3
                // Sentencias caso 3
                [salir]
        …
        default
                // Sentencias default
                [salir]
```

Aquí se introduce una nueva palabra reservada, **“salir”**, por medio de la cual se podrá salir de cualquier ciclo o sentencia de control sin ejecutar el código que se encuentre por debajo de esta palabra.

3.13.3) Para:

Este ciclo debe tener una asignación, seguido de una comparación o una expresión booleana y por último una sentencia de incrementación o disminución en unidades. El ciclo ejecutará su contenido hasta que la condición o expresión booleana sea verdadera por lo que si la condición inicialmente es verdadera el ciclo no se ejecuta. Los signos de incrementos o disminución son “++” y “--” respectivamente.

La estructura de un ciclo “para” es la siguiente:

```
para(asignación; condición; incremento|disminución/acción posterior)
        // Sentencia 1
        // Sentencia 2
        // Sentencia 3
        …
        // Sentencia n
```

Nota: La variable de asignación puede ser declarada antes o inicializada directamente en la misma asignación, así como la acción posterior debe de haber cualquier acción posterior desde una asignación con incremento o disminución u otra acción.

3.13.4) Mientras:

Este ciclo ejecutará su contenido siempre que se cumpla la condición que se le dé por lo que podría no ejecutarse si la condición es falsa desde el inicio.

La estructura de un ciclo “mientras” es la siguiente:

```
mientras(condición)
        // Sentencia 1
        // Sentencia 2
        // Sentencia 3
        …
        // Sentencia n
```

3.13.5) Hacer-Mientras:

Este ciclo ejecutará al menos 1 vez su contenido, luego comprobará la condición para determinar si debe o no ejecutarse nuevamente.

La estructura de un ciclo “Hacer-Mientras” es la siguiente:

```
hacer
        // Sentencia 1
        // Sentencia 2
        // Sentencia 3
        …
        // Sentencia n
mientras(condición)
```

3.13.6) Continuar

Para los ciclos puede venir la palabra reservada “Continuar” que lo que hace es que obvia las demás instrucciones siguientes a partir de donde fue escrita de la iteración en curso y pasa a la siguiente iteración.

Ejemplo:

```
hacer
        // Sentencia 1
        // Sentencia 2
        Continuar
        // Sentencia 3
mientras(condición)
```

Por lo que en este se ejecutará las sentencias 1 y 2, pero siempre se obviaba la sentencia 3 por lo que nunca se ejecutaría, en todas las iteraciones del ciclo while.

NOTA: Para el uso de las sentencias salir o continuar se debe verificar el ambiente, si el ambiente no es el correcto deberá marcar error.

3.14) Funciones y Procedimientos

Las funciones y procedimientos solo pueden ser llamados desde dentro de algún procedimiento o función definida dentro de la pista, no directamente dentro de la Pista. La declaración de funciones y procedimientos se realizará de la siguiente manera:

```
[keep] [Tipo] nombre([tipo parametro1, tipo parametro2, . . .tipo parametron])
        // Sentencia 1
        // Sentencia 2
        …
        // Sentencia n
        [retorna valor]
```

En esta parte se introduce la palabra reservada **“retorna”** para indicar el valor (o variable) a retornar en el método. El tipo es opcional, esto quiere decir que en caso de no ser escrito, el método se tomará como un **“void”** por lo que no será necesario establecer un valor de retorno. El prefijo keep establece su obligatoriedad al declararse/sobrescribirse

Ejemplo:

```
retorna numero2
```

3.15) Funciones y Procedimientos Nativos del Lenguaje

Las funciones y procedimientos solo pueden ser llamados desde dentro de algún procedimiento o función definida dentro de la pista, no directamente dentro de la Pista.

Toda Pista tiene de forma nativa **canales** que se ejecutarán en paralelo al final del método principal de la pista esto se hace para que pueda existir un ritmo de fondo o varios. Con esto también se sabe que **la duración máxima de una Pista será la duración del canal más largo**, por lo que sí existen canales más cortos de tiempo estos terminaran antes que el canal más largo sin ningún problema. Estos canales no los debe declarar el usuario en cada Pista estos vienen implícitos en cada Pista por lo que no debe de existir ninguna variable para poder realizar esta función de canal.

La cantidad de canales que el usuario puede utilizar es infinita por lo que el usuario con tan solo utilizarlo en la función “Reproducir” o en “Esperar” se agregara a la lista de canales que tiene la Pista.

3.15.1) Reproducir

Esta función agrega a un determinado canal de la pista el tono y su respectiva octava definida con valor de tipo entero la cual solo puede estar entre 0 y 8 incluyendo estos extremos y la cantidad determinada de milisegundos que será reproducida la nota que se establecen por medio de un entero.

Reproducir (Nota, entero Octava, entero milisegundos, entero canal)

El valor que Nota puede tener es: do, do#, re, re#, mi, etc.

Más adelante se definen las notas existentes y sus respectivas escalas como también la frecuencia asociada a cada par de nota y octava. Esta frecuencia es la que el sistema reproduce.

Hay que tomar en cuenta que en el caso de tener la llamada del método nativo “reproducir” el intérprete deberá de almacenarlo en el canal especificado en la sentencia por lo que debe de apilar todos los segmentos de notas que se hayan ido haciendo a través de esta función a lo largo de todo el código de la pista

Hay combinaciones de nota y octava que no tienen frecuencia asociada, en estos casos no debe de reproducirse nada ni agregarse al canal.

El tipo de esta función es entero y lo que retornara es la cantidad de milisegundos que sonará, es el mismo valor que ingresa como su tercer parámetro

Ejemplo:

```
Reproducir(Do, (2*1)+1,50*arr0[3]+62*6*b,3)
entero tiempo = Reproducir(Re, 6,15000,1)
```

3.15.2) Esperar

Esta función tiene como finalidad dar tiempo entre reproducciones continuas y sentencias de código esto tomará como parámetro la cantidad de milisegundos definido como entero y también el canal en el cual se generara la espera, en el canal no reproducirá nada en este tiempo pero también se sumará al tiempo total del canal.

Espera(entero milisegundos, entero canal)

Ejemplo:

```
Espera(2000 , 8)
```

El ejemplo anterior hace que en el canal 8 se espere 2 segundos de la última nota almacenada en el canal y la nota siguiente que se almacene en el canal, si en caso hubiera más notas.

Suponiendo las siguientes sentencias como únicas dentro del método principal de la pista

```
Reproducir(Do,5, 1000,5)
Reproducir(Re,4, 1500,3)
Esperar(200,5)
Reproducir(Re,4, 1500,3)
Reproducir(Mi,6, 3500,5)
Reproducir(Sol,7, 1500,2)
Esperar(200,3)
Reproducir(Fa,5, 500,3)
```

Los canales para dicha lista serian: 2,3,5. la duración total por canal es

canal 2: 1500 milisegundos
canal 3: 3700 milisegundos
canal 5: 4700 milisegundos

El canal 5 es el más largo, por lo tanto la duración de la pista es 4700 milisegundos y las pistas 2 y 3 como es de esperarse se detendrán antes lo cual no tiene ninguna anomalía.

3.15.3) Método Principal de Pista

Éste método es el que el intérprete buscará al momento de reproducir la canción, a partir de este método el usuario puede llamar a otros métodos o funciones en donde define las sentencias necesarias para poder crear una Pista con sus respectivas notas anidadas en los distintos canales que en la pista se definan.

Si una pista no tiene método principal no tendrá canales definidos y por lo tanto no deberá ser reproducido, aunque es permitido definir una pista sin método principal dado a que esta pista puede que solo sirva para extender otras pistas.

La forma en que este método se declara es la siguiente:

```
Principal()
        // Sentencia 1
        // Sentencia 2
        ……
```

Como se ha mencionado anteriormente el lenguaje no soporta la extensión/herencia de métodos principales entre Pistas.

El proceso de reproducción se llevará a cabo cuando el método principal termine de ejecutar todas sus sentencias, hasta que llegue al fin del método.

Reproduciendo los canales en paralelo, si en dado caso uno de canales no tiene contenido no perjudica la reproducción de los otros canales.

3.15.4) Ordenar:

El método ordenar recibe como parámetro un arreglo y un parámetro para indicar de qué manera se quiere ordenar, si el ordenamiento tuvo éxito retorna un valor de 1, si la forma de ordenar no es aceptada o no se realizó el ordenamiento retorna 0.

Estructura:

Retorna: entero Ordenar (arreglo <arreglo a ordenar>,tipo <forma de ordenar>)

Dónde:

arreglo a ordenar: es una variable tipo arreglo que puede ser de tipo entero, cadena, caracter o doble

forma de ordenar: es la manera de como ordenara el arreglo. Se guiará como dice a continuación:

formadeordenar	Acción	Aplica
ascendente	ordena de menor a mayor	todos
descendente	ordena de mayor a menor	todos
pares	ordena los pares primero y después todos los demás	entero, doble(tomando solo la parte entera), carácter (según orden ascii)
impares	ordenar los impar primero y después todos los demás	entero, doble(tomando solo la parte entera), carácter (según orden ascii)
Primos	ordena los primos primero y después todos los demás	entero, doble(tomando solo la parte entera),carácter (según orden ascii)

Tabla 11:"Formas de ordenar procedimiento Ordenar"

3.15.5) Sumarizar:

El método sumarizar extrae la suma de todos los elementos del arreglo para lo que ese será su salida, recibe como parámetro un arreglo, su retorno es una cadena con la suma de todos sus elementos.

Estructura:
Retorno: cadena Sumarizar (arreglo <arreglo a sumar>)

Dónde:
arreglo a sumar: es una expresión tipo arreglo que puede ser de tipo entero, cadena, caracter o doble

Ejemplo:

```
var entero arreglo arr0 [3]={5,10,15}
var caracter arreglo arr1 [4]={'a','b','F','5'}
var doble arreglo arr3 [2]={1.5,2.6}
var cadena cad

cad=Sumarizar(arr0)                    // valor de cad: "30"
cad=Sumarizar(arr1)                    // valor de cad: "abF5"
cad=Sumarizar(arr3)                    // valor de cad: "4.1"
cad=Sumarizar({"hola"," ","mundo"})    // valor de cad: "hola mundo"
```

3.15.6) Longitud:

El método longitud devuelve el tamaño del arreglo o de la cadena introducida de salida tiene un entero con esa cantidad.

Estructura:
Retorno: entero Longitud (arreglo <arreglo a medir>/cadena <cadena a medir>)

Dónde:
arreglo a medir: es una expresión tipo arreglo.
cadena a medir: es una expresión tipo cadena que se medirá sus caracteres

Ejemplo:

```
var entero arreglo arr0 [3]={5,10,15}
var caracter arreglo arr1 [4]={‘a’,’b’,’F’,’5’}
var cadena cad1=”juan”
var entero num1

num1=Longitud(arr0)          // valor de num1: 3
num1=Longitud(arr1)          // valor de num1: 4
num1=Longitud(cad1)          // valor de num1: 4
num1=Longitud(“pedrito”)          // valor de num1: 7
```

3.15.7) Mensaje:

El método mensaje desplegará una ventana de alerta con algún mensaje dentro del servidor, esto para advertir sobre excepciones en el código.

Estructura:

```
Mensaje (cadena <texto>)
```

Dónde:
texto: puede ser una expresión compuesta de cadenas, caracteres o cualquier dato primitivo, por medio del cual se mostrarán mensajes. También se podrá mostrar el valor de una variable como parte de un mensaje.

Ejemplo:

```
var entero num1 = 22
var cadena cad1=”Compiladores 2”

Mensaje(cad1)                    //Mostrará: Compiladores 2
Mensaje(“Proyecto ”+1)           //Mostrará: Proyecto 1
Mensaje(cad1+” ”+num1)           //Mostrará: Compiladores 2 22
```

4) Lenguaje de definición de listas de reproducción (módulo local)

Las listas de reproducción serán definidas a partir de un lenguaje específico para ello, por lo tanto, el módulo local deberá contar con un editor y un analizador para éste lenguaje.

Una lista de reproducción se definirá de la siguiente manera:

```
{lista:
        {nombre: “<nombre de la lista de reproducción>”,
        random: true|false, // opcional
        circular: true|false, // opcional
        pistas: [Pista1,Pista2,Pista3,...,Pista_n ] //opcional
        }
}
```

Ejemplo de una lista de reproducción:

```
{lista:
        {nombre: “fiesta”,
        pistas: [Bailable1, Salsa3, Cumbia4]
        }
}
```

Se debe tomar en cuenta que las listas pueden contener más de una sola pista o no contener ninguna, por lo que el atributo pistas es opcional. Los atributos random y circular también son opcionales, por defecto las pistas no serán ni aleatorias ni circulares. También se debe considerar que las pistas no pueden encontrarse más de una sola vez en una lista, por lo que la repetición de una pista dentro de una lista de reproducción deberá ser marcada como error.

Otro punto a tomar en consideración, es que una pista puede encontrarse en varias listas de reproducción y no solamente en una.

4.1) Grupo de listas

Si se desea se pueden definir varias pistas en un solo archivo, agrupándolas dentro de corchetes ([ ]) y utilizando la etiqueta grupo_listas, por lo que el analizador deberá cargar las mismas al sistema. Por ejemplo:

```
{grupo_listas: [
        {lista:
                {nombre: “fiesta”,
                random: true,
                pistas: [Bailable1, Salsa3, Cumbia4]
                }
        },
        {lista:
                {nombre: “rockPesado”,
                random: true,
                circular: false,
                pistas: [one,hell_bells, la_cumbia_metalera]
                }
        }
] }
```

5) Frecuencias de notas musicales con sus respectivas octavas

Cada pareja de nota música por ejemplo { Do, octava 1 } tiene su respectiva frecuencia la cual puede ser reproducida por cualquier dispositivo que genere vibraciones en el ambiente, por ejemplo una bocina, una cuerda, etc.

Una representación más completa de la pareja de notas musicales y frecuencias que dependen de la octava que la posee de un piano en particular se muestra a continuación:

Frecuencias (en hertzios) de las notas musicales

	Oc. 0	Oc. 1	Oc. 2	Oc. 3	Oc. 4	Oc. 5	Oc. 6	Oc. 7	Oc. 8
Do		32,70	65,41	130,81	261,63	523,25	1046,50	2093,00	4186,01
Do#		34,65	69,30	138,59	277,18	554,37	1108,73	2217,46	
Re		36,71	73,42	146,83	293,66	587,33	1174,66	2349,32	
Re#		38,89	77,78	155,56	311,13	622,25	1244,51	2489,02	
Mi		41,20	82,41	164,81	329,63	659,26	1318,51	2637,02	
Fa		43,65	87,31	174,61	349,23	698,46	1396,91	2793,83	
Fa#		46,25	92,50	185,00	369,99	739,99	1479,98	2959,96	
Sol		49,00	98,00	196,00	392,00	783,99	1567,98	3135,96	
Sol#		51,91	103,83	207,65	415,30	830,61	1661,22	3322,44	
La	27,50	55,00	110,00	220,00	440,00	880,00	1760,00	3520,00	
La#	29,14	58,27	116,54	233,08	466,16	932,33	1864,66	3729,31	
Si	30,87	61,74	123,47	246,94	493,88	987,77	1975,53	3951,07	

6) Módulo Exterior (Consumidor)

El módulo exterior o consumidor, consistirá en una aplicación móvil para el sistema operativo Android, la cual podrá realizar peticiones al módulo servidor de las cuales dependerá su funcionamiento. Ésta aplicación podrá reproducir las pistas almacenadas por el servidor, así como obtener las diversas listas de reproducción.

La comunicación entre el módulo exterior y el módulo servidor debe ser únicamente por sockets utilizando el lenguaje que posteriormente será definido.

Las funciones de la aplicación móvil se definen a continuación:

Solicitud de listas de reproducción

La aplicación móvil deberá ser capaz de solicitar las listas de reproducción al servidor. Por medio de sockets enviará un mensaje utilizando un lenguaje específico de comunicación, especificado más adelante. El servidor deberá responder de manera síncrona con el conjunto de listas de reproducción existentes utilizando el lenguaje antes mencionado.

Selección de lista de reproducción

Una vez se obtengan las listas de reproducción existentes, el usuario de la aplicación móvil podrá seleccionar una de las listas de reproducción devueltas por el servidor. De nueva cuenta la aplicación móvil deberá enviar por medio de sockets la solicitud al servidor, utilizando el lenguaje de comunicación de ambos. El servidor responderá con la lista de pistas que se encuentran en la lista de reproducción.

Selección y reproducción de pista

Una vez se hayan obtenido las listas y/o pistas por parte del servidor, deberán mostrarse en la aplicación de forma que puedan ser seleccionadas y reproducidas desde el dispositivo móvil. Durante la reproducción, se deberá mostrar el gráfico de frecuencias asociado a cada canal que la pista contenga a lo largo del tiempo que la misma dura.

Creación de pistas nuevas

La aplicación deberá tener la opción de crear una nueva pista, para ello la aplicación poseerá un teclado musical integrado, el cual tocará las notas que conformarán la pista nueva. Posterior a ello, la pista es enviada al servidor, el cual la almacenará y la reproducirá.

7) Lenguaje de comunicación entre aplicaciones (Ambos módulos)

La comunicación a través de sockets entre ambos módulos se debe de realizar utilizando el lenguaje que se define a continuación, solamente se puede utilizar este lenguaje.

7.1) Solicitar lista(s) de reproducción

```
<solicitud>
    <tipo>Lista</tipo>
    <nombre>”nombre_lista”</nombre>
</solicitud>
```

El tag <nombre> puede venir o no venir, lo cual nos dará las siguientes respuestas por parte del servidor.

7.2) Respuesta del servidor para lista(s) de reproducción (sin tag <nombre>)

```
< listas >
    < lista nombre = ”tranquilas” pistas = 5>
    < lista nombre = ”fiesta” pistas = 10>
    …...
</ listas >
```

7.3) Respuesta del servidor para lista(s) de reproducción (con tag <nombre>)

```
< lista nombre = ”tranquilas” aleatoria =”SI/NO”>
    < pista nombre = ”pista1” duracion = 14500 >
    < pista nombre = ”pista2” duracion = 60000 >
    …...
</ lista >
```

7.4) Solicitar pista(s) de reproducción

```
<solicitud>
    <tipo>Pista</tipo>
    <nombre>“nombre_pista”</nombre>
</solicitud>
```

Del mismo modo, el tag <nombre> puede venir o no venir, lo cual nos dará las siguientes respuestas por parte del servidor.

7.5) Respuesta del servidor para lista de pistas (Sin tag <nombre>)

```
< pistas >
        < pista nombre = ”pista1” duracion = 14500 >
        < pista nombre = ”one minute” duracion = 60000 >
        ……
</ pistas >
```

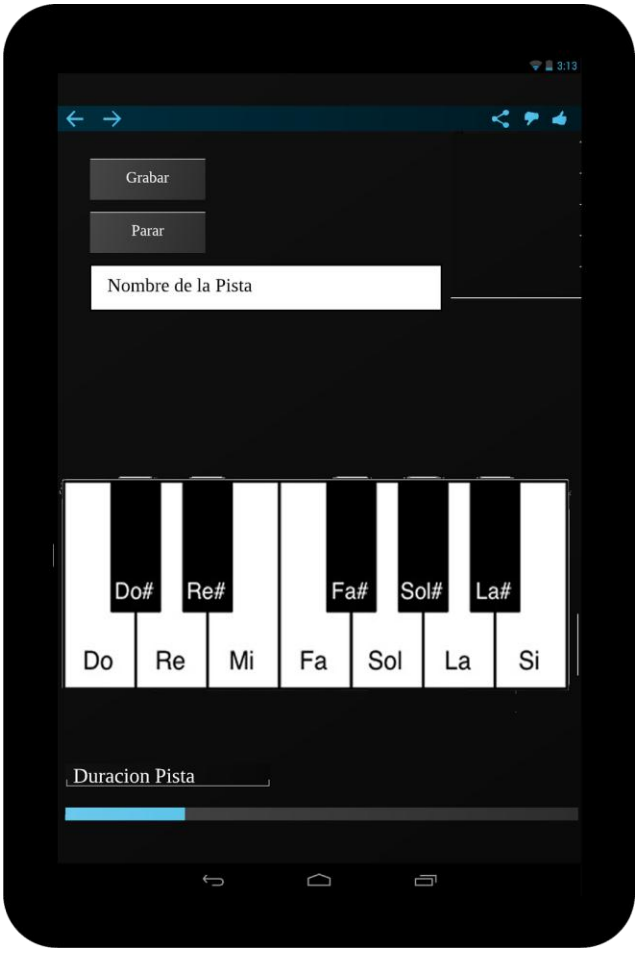
7.6) Respuesta del servidor para una pistas (Con tag <nombre>)

```
< pista nombre = ”pista1” >
    < canal numero = 1 >
        < nota duracion = 4000 frecuencia = 32.7 > //do octava 1 4 seg
        < nota duracion = 3500 frecuencia = 73.4 > //re octava 2 3.5 seg
        < nota duracion = 7000 frecuencia = 0 >    //esperar 7 seg
        …..
    </ canal >
    < canal numero = 8 >
        < nota duracion = 3500 frecuencia = 73.4 >
        < nota duracion = 7000 frecuencia = 0 >
        < nota duracion = 7000 frecuencia = 0 >
        < nota duracion = 4000 frecuencia = 32.7 >
        …..
    </ canal >
    ……..
</ pista>
```

7.7) Creación de pistas nuevas

Esta función se llevará a cabo mediante el uso del teclado musical el cual capturará la frecuencia de la nota tocada y la reproducirá mientras la nota se sostenga durante un periodo x de tiempo.

Para llevar a cabo esto, se deberá especificar el nombre de la pista e iniciar la grabación, lo cual realiza una captura de datos en base a la nota que se toque y a la duración que se mantenga sostenida, si en dado caso no se presiona ninguna tecla, se interpretará como una espera. Un vez se detenga la grabación de la pista, se creará una solicitud de la siguiente forma.



Ejemplo:

```
<solicitud>
        <tipo>pistanueva</tipo>
        <datos>
                <canal >1</canal>
                <nota>Do#</nota>
                <octava>2</octava>
                <duracion>1000</duracion>
        </datos>
        <datos>
                <canal>2</canal>
                ….
        </datos>
        …..
</solicitud>
```

8) Manejo de errores

Se debe mostrar un reporte de los errores que se encuentren durante las fases de compilación del proyecto, todos los errores deben tener una descripción concisa del porqué se generó el error, así como su posición. Por ejemplo:

Línea	Columna	Tipo	Descripción
3	30	Sintáctico	Se esperaba (
85	4	Léxico	Símbolo ° no esperado
125	18	Semántico	La variable A no ha sido inicializada

Tabla 12:"Errores mostrados en módulo local"

Consideraciones

Cuando el compilador encuentre un error (ya sea léxico, sintáctico o semántico) este deberá continuar analizando, pero no deberá ejecutar, la finalidad es mostrar al usuario la mayor cantidad de errores posible que se encuentren en su archivo.

Para el manejo de errores de forma interna es necesario que se utilice la librería log4Net y que estos además de su reporte respectivo de errores se genere un archivo log añadiéndoles estos errores captados con la librería log4Net.

9) Ejemplo de Entrada de Lenguaje de Pistas:

```
Pista RitmoFondo
    keep var entero contador
    var boolean bandera = verdadero
    var cadena texto="hasta aca todo va bien" + " " + "!!" + "#n"
    keep var doble uno, dos, tres, cuatro=50.4*6
    var entero arreglo arr1 [(a+b+(c+(d+1)))^3] // arreglo con 1 dimensión
    var cadena cad1, cad2, cad3 = texto + "aca empieza otra linea :D"
    >>puede asignar fuera de metodos utilizando otras variables
    entero primer_ritmo(entero veces)
        <- este metodo pretende
        crear un ritmo de fondo
        indicando la velocidad ->
        entero tiempo >>esta variable almacena la duracion de la pista
        mientras(permitido==verdadero)
            si(bandera==verdadero)
                esperar(200,1)
                tiempo+=200
            sino
                Reproducir(Do, 5,(20*10)*2,1)
                tiempo+=(20*10)*2
            bandera=!(bandera)
            contador++
            si(contador>=veces)
                permitido=falso
                contador--
        retorna (++tiempo)
```

```
27      keep boolean asignaciones(boolean parametro1)
28          boolean var0
29          boolean var1=(5/(5+5))<(8*8)%3
30          Etiqueta empieza
31          var0=var1!=(var1==verdadero)
32          bandera= var0 || var1
33          bandera= !bandera
34          si(!¡contador)
35              mensaje("la variable contador es nula")
36              ira empieza
37          sino
38              mensaje(texto)
39          retorna bandera
40
41      keep principal()
42          <- puede agregarse el prefijo keep en el metodo principal
43          pero este no cumple ninguna funcion
44          pues nunca se heredara este metodo ->
45
46          entero lleva=primeritmo(15)
47          asignaciones(true && (lleva>=200000))
48          matrices(3,1,2,3,4)
49          ciclos()
50
51
```

```
51
52      keep entero matrices(entero multiplicador,entero a, entero b, entero c, entero d)
53          Keep var entero arreglo arr2,arr3,arr4 [d^multiplicador][a*b+c]
54          >>arreglos con 2 dimensiones y keep esta de mas dado a que no esta en un nivel externo
55          arr1={5,10}
56          var entero reg=arr3[1][2]+arr1[1]
57          retorna reg
58      ciclos()
59          para(entero a=10;a>0;a--)
60              mensaje("el valor de a es#n")
61              mensaje(a)
62              si(a==5)
63                  continuar
64              mientras(primerritmo(a)<1000)
65                  a++
66                  Reproducir(Re, 3,1000,9)
67                  esperar(500,9)
68                  Reproducir(Re, 3,1000,9)
69                  esperar(500,9)
70      keep entero Ordenar()
71          var entero rev=0
72          Ordenar(arr1,tipo descendente)
73          rev=Tamano(arr1)
74          retorna rev
```

Restricciones y Observaciones

- El proyecto se realizará de manera individual.
- La modificación de código no es permitida al momento de la calificación.
- El lenguaje de programación a utilizar será Java para el módulo externo con las herramientas de parser CUP para análisis sintáctico y JLex/JFlex para análisis léxico, no se puede utilizar alguna otra como XMLPullParser, etc.
- Copias de proyectos tendrán una nota de cero puntos y serán reportados al catedrático titular de su sección y a la Escuela de Ciencias y Sistemas para su respectiva sanción.
- Para el módulo local/servidor se debe de utilizar para la generación de Analizador léxico y sintáctico Irony.
- La única manera de comunicarse entre el módulo local y el módulo externo es por medio de sockets.
- Requerimientos mínimos funcionales:
    - Modulo local
        - Recuperar, crear pistas y listas de reproducción
        - Reproducción de pistas
        - Manejo de Procedimientos y Funciones generales
        - Funciones Reproducir, Esperar, Procedimientos ordenar y sumarizar
        - Declaración y Asignación de variables
        - Operadores Aritméticos, Relacionales y Lógicos
        - Ciclos y Bifurcaciones
        - Método Principal
    - Modulo Consumidor
        - Solicitud de listas y pistas
        - Reproducción de pistas

El no cumplir con alguno de los requerimientos mínimos funcionales, involucra una penalización sobre la nota al momento de calificarse.

Fecha de Entrega

22 de septiembre del 2014 *NO SE DARÁ PRÓRROGA***.** Se entrega todo en un disco compacto y la entrega será presencial. El lugar y hora será acordado con su auxiliar.