A Music Language

Documentación de la Práctica de Compiladores

2do Cuatrimestre - Curso 2016/2017

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Mario Fernández Villalba

Juan Miguel de Haro Ruiz

Carlos Roldán Montaner

1. Objetivos

El objetivo principal de **A Music Language** es permitir la mezcla de dos mundos aparentemente muy distintos: la música y la informática. AML es un lenguaje de programación que permite usar tanto notación musical en un lenguaje imperativo inspirado en C++.

Gracias a nuestro lenguaje, se simplifica la creación de música compleja y se permite explotar patrones que se repiten en las composiciones musicales para así obtener programas más cortos.

Otro objetivo importante es que cualquier músico pueda usar el lenguaje sin perder mucho tiempo aprendiendo, por lo que la conversión de partitura a nuestro lenguaje es directa. Además, AML añade muchas más funcionalidades que las que permiten normalmente las partituras.

Esto también se aplica en el caso contrario, queremos que además de ser fácil para los músicos, también lo sea para los programadores, por lo que la sintaxis es muy parecida a la de C++, un lenguaje muy conocido y usado.

Por último, nuestra meta también es conseguir comprimir los archivos de música, ya que canciones largas se pueden representar con AML en menos espacio que su respectiva versión en MIDI y aún más en formato mp3 o parecidos.

2. Descripción del lenguaje y funcionalidades.

Como ya hemos mencionado anteriormente, AML permite usar tanto notación musical en un lenguaje imperativo inspirado en C++. Para conseguir esto AML dispone de diferentes tipos de datos y de dos tipos de instrucciones: imperativas y musicales.

2.1 Tipos de datos

Int

El tipo de dato Int representa un número entero. Dispone de los operadores aritméticos de suma, resta, producto, cociente y módulo. Además, puede combinar todas estas operaciones con el tipo Bool.

Bool

El tipo de dato Bool representa un booleano clásico, cuyos unos valores posibles son Cierto y Falso. Se puede operar con este tipo de dato con las operaciones clásicas de la lógica (And, Or, ...). Al ser utilizados en operaciones aritméticas estos toman valores de 1 si es cierto y 0 si es falso.

String

Una string se trata de una cadena de caracteres. Dispone del operador de concatenación (representado con el símbolo '+') que permite combinar distintas strings.

Nota musical

En AML, la nota es un tipo de dato del lenguaje. Se puede declarar como variable y ser usada en diferentes contextos o como literal en cualquier compás.

Los atributos de una nota son el nombre de la nota, la tonalidad, y su accidente. El nombre de la nota se especifica con la convención neo-latina (Do, Re, Mi, ...). La unidad básica de tonalidad es el semitono, y es tratada como un entero. El accidente puede ser *Natural, Bemol*, *Sostenido* y *Armadura.* Debido a las limitaciones de un teclado estos tres últimos son representados con *&, #* y *$* respectivamente. Sobre una nota podemos realizar operaciones que modifiquen su accidente (sostenidos, bemoles, etc.) mediante la función *setAccident().* Disponemos de una serie de métodos booleanos que nos indican si una nota tiene o no cierto accidente. Estos son *isBool*, *isSustain()*, *isArmor()* e *isNatural()*. También podemos saber si una nota es un silencio mediante *isSilence()* y podemos convertir una nota en un silencio mediante la función *makeSilence()*. Por último, el método *randomize()* modifica todos los parámetros de una nota de forma aleatoria para conseguir un sonido único.

Nota de percusión

En AML, la nota de percusión es un tipo de dato del lenguaje. Se puede declarar como variable y ser usada en diferentes contextos o como literal en cualquier compás.

La nota de percusión es un tipo de nota especial que representa un sonido de un instrumento de percusión. Esta solo contiene un atributo representado por un entero, que especifica qué sonido de un instrumento de percusión contiene la nota (para saber que sonido representa cada entero véase la especificación de General MIDI). El constructor de una nota de percusión es *DN(numero_de_sonido_de_percusión)*. Hay un caso especial que sirve para representar el silencio en una nota de percusión, que es el de una nota de percusión con el número de sonido de percusión negativo. Las notas de percusión cuentan con la función *randomize()* que genera una nota de percusión aleatoria, y *makeSilence()* que convierte una nota de percusión en silencio.

Figura musical

En AML, la figura es un tipo de dato del lenguaje. Se puede declarar como variable y ser usada en diferentes contextos o como literal en cualquier compás.

Una figura es una representación de un intervalo de tiempo al que pueden ser asignadas una o varias notas musicales. Es por esto que la figura dispone de dos tipos de constructores: el primero nos sirve para crear una figura con una única nota. Esto se consigue escribiendo *nota.tipo_de_figura*. El segundo es usado declarar una figura con diversas notas ,y se escribe como (*nota1 nota2 ...).tipo_de_figura.* Los atributos de una figura son el tipo de figura, su duración, y sus modificadores. El tipo de figura puede ser semifusa (sf), fusa (f), semicorchea (sc), corchea (c), negra (n), blanca (b) y redonda (r). La duración de una nota viene representada por un entero y puede ser modificada directamente. Los modificadores de una figura son el punto y la ligadura. El punto viene representado con el carácter ***** y suma la mitad del tiempo de una figura a esta. La ligadura viene representada por el carácter **~** y permite sumar el tiempo de una figura con el de la siguiente siempre que esta tenga la misma tonalidad. Este modificador podrá ser añadido mediante las funciones *tie()* y *untie(),* y también podremos consultar su estado mediante la función *isTie()*.

Acorde

En AML, el acorde es un tipo de dato del lenguaje. Se puede declarar como variable y ser usada en diferentes contextos o como literal en cualquier compás.

Podríamos considerar un acorde como un subtipo de figura especial, donde las notas contenidas en esta vienen predeterminadas por el tipo de acorde. Los atributos de una acorde son el nombre de su nota base, el accidente, la octava, la cualidad, el intervalo y los demás atributos de una figura. El acorde dispone del constructor *Chord(nota_base cualidad intervalo).tipo_de_figura.* Para modificar los diferentes atributos de un acorde disponemos de diferentes funciones: *alterQuality()* modifica la cualidad de un acorde. La cualidad puede ser *Major, Minor, Augmented, Diminuted*. *alterInterval()* modifica el intervalo de un acorde.

El intervalo puede ser *7th, maj7* y *NoInterval. setRoot()* nos permitirá cambiar la nota base, que puede ser cualquier nota de la escala musical. *setAccident()* nos permite cambiar el accidente del acorde, cuyo valor puede ser *Natural, & (Bemol)*, # (*Sostenido)* y $ (*Armadura). setOctave()* nos permite establecer la octava del acorde, y debe ser especificada como un entero.

Otros elementos musicales

Existen otros elementos que forman parte de AML a pesar de no ser tipos de datos, es decir, de no poderse almacenar en variables. Sin embargo, son esenciales para la correcta organización y generación de piezas musicales mediante AML. Se trata de las canciones, las pistas y los compases.

Canciones

Las canciones representan composiciones musicales formadas por pistas. Los atributos de una canción son su tempo, su métrica y su tonalidad. El tempo viene dado en *Beats Per Minute* (BPM). La métrica es representada con el formato *n:m*, es decir, en un compás debe haber **n** figuras de duración **m/4** negras. La tonalidad viene dada por el número de sostenidos o bemoles de ésta. Estos atributos pueden ser modificados con instrucciones musicales, que explicaremos más adelante.

Pistas

Una pista está formada por compases y es interpretada por un instrumento. Un instrumento puede ser especificado por una string al principio de la pista, que ha de contener un nombre válido de la especificación General MIDI. Existe un tipo de pista especial utilizada para reproducir notas de percusión. Esta pista ha de ser creada al final de una canción mediante el constructor *Drums ||lista_de_compases||.*

Cualquier nota, ya sea musical o de percusión, puede ser introducida en cualquier tipo de pista; AML se encargará automáticamente de reproducir sonidos musicales o percutivos dependiendo del tipo de pista.

Compases

Un compás es un conjunto de notas que puede contener tanto instrucciones imperativas como musicales. Dentro de un compás no puede haber otros compases. No obstante, es posible llamar desde dentro de un compás a una función que contenga compases. También es posible introducir dentro de un compás fragmentos de compases. Un fragmento es una función especial que solo puede contener todo lo que pueda contener un compás. La métrica de un compás será comprobada automáticamente, y se tendrán en cuenta los fragmentos para esta comprobación. AML da la posibilidad de repetir compases utilizando la notación natural ||: :|| y de finales alternativos usando la variable *Time*, que contiene la iteración en la que se encuentra el compás en la repetición más interna, ya que éstas pueden estar anidadas. Un compás no ha de estar situado necesariamente dentro de una canción.

AML interpreta el hilo principal de ejecución de un programa como una pista. Ésta tendrá un tempo, métrica, tonalidad y instrumento propios que por defecto son 120 bpm, 4:4, Do mayor y "Grand Piano" respectivamente. Toda canción que no especifique alguno de estos atributos lo heredará del hilo principal.

2.2. Tipos de instrucciones

Instrucciones imperativas

Las instrucciones imperativas son bastante similares a las que implementan los principales lenguajes imperativos (C++, C, C#, Java, etc.). Encontramos las típicas instrucciones de control de flujo: *if*, *for* y *while*, que operan con Ints, Bools y strings. También disponemos de llamadas a funciones (que pueden devolver datos), declaraciones (con o sin asignación), declaraciones múltiples, asignaciones, operadores aritméticos (suma, resta, producto, cociente y módulo) y lógicos (AND, OR, NOT). Podemos combinar asignaciones con operadores aritméticos para conseguir asignaciones con incremento (+=), decremento (-=), etc. También disponemos de las operaciones de preincremento, postincremento, predecremento y postdecremento. Además, las instrucciones *Read* y *Write* constituyen un sistema de entrada y salida, en el que la primera permite al usuario introducir datos, y la segunda permite mostrarlos por el canal de salida estándar.

Instrucciones musicales

Las instrucciones musicales nos permiten controlar los distintos parámetros musicales. Podemos establecer el beat mediante la instrucción *Beat*, la velocidad mediante la instrucción *Speed*, el tono mediante la instrucción *Tone* o volumen mediante la instrucción *Volume*, así como determinar el instrumento con el que se reproducirá la pieza.

3. Gramática

Al ser un lenguaje estático, decidimos empezar la gramática prácticamente de cero. Lo único que comparte con ASL es la regla expr, ya que las expresiones son muy parecidas en ambos lenguajes. En cuanto al resto de la gramática, intenta simular la sintaxis de C++ con elementos musicales. Para evitar ambigüedades tuvimos que introducir elementos distintivos delante de identificadores para poder distinguir casos. Por ejemplo, los fragmentos y las funciones se llaman de la misma manera, así que para distinguirlos usamos el token ‘F:’ para llamar fragmentos. También pasa lo mismo con el acceso a notas y figuras cuando son variables, y para las notas cuando son literales.

Por ejemplo, para escribir una figura, ya sea para asignar a una variable o para ponerla en un compás, se puede escribir como una nota o como un conjunto de notas con sus respectivos modificadores (accidente, octava). Por lo tanto, para distinguir una nota como tipo nota de una figura que sólo contiene una nota, se usa el token '*N:*' para distinguirlos. Para las variables pasa un caso parecido, por eso se usa '*N*:' para acceder a variables nota y '*:*' para acceder a variables de tipo figura. Aml nos permite escribir un tipo de figura especial, las figuras de tresillo. Estas deben ser declaradas en grupos de tres rodeadas por corchetes, indicando el tipo de figura de tresillo al final. Por ejemplo, para declarar tres corcheras de tresillo hemos de escribir *[notas_figura1 notas_figura2 notas_figura3].c.*

En cuanto a las instrucciones, como no se pueden usar las mismas dentro de un compás que fuera (no puede haber compases dentro de un compás), se distinguen dos tipos de instrucciones. Por eso las instrucciones de control de flujo (*if, while, for*) se tienen que duplicar, una para cada tipo.

Las canciones se representan con listas de compases, las cuales empiezan y acaban con doble barra. También se puede empezar con una repetición, pero por un problema de ambigüedad, siempre se tienen que acabar con la doble barra. Una lista de compases es una secuencia de instrucciones musicales separadas por una barra (la cual separa compases diferentes) y de repeticiones, las cuales son también listas de compases pero dentro delimitadas por dobles barras con punto (||*:* y *:*||).

Dejando los compases a parte, el resto de la gramática funciona de manera parecida a la del ASL (llamadas a funciones, control de flujo..).

4. Semántica

Diferencias principales respecto a ASL

La semántica de AML es muy diferente a la del ASL. EL hecho de ser dinámico, hace que la comprobación semántica se haga en tiempo de ejecución, lo que le da más trabajo al intérprete. Por eso decidimos quitarle este trabajo ya que nuestro lenguaje es estático.

De ahí sale el analizador semántico, el cual va a parte del intérprete y comprueba todo lo que se puede antes de ejecutar el código, como la comprobación de tipos, la visibilidad de variables, las llamadas a funciones...

De esta manera, el trabajo del intérprete se simplifica mucho, ya que éste supone que todo es correcto en el momento de la ejecución.

En cuanto al acceso de variables, ya no hace falta un mapa para guardarlas dinámicamente, sino que el analizador semántico ya sabe cuántas va a haber visibles en cada momento. Por eso a cada nodo del AST que pertenece a un acceso de una variable, le asigna una id que corresponde a su posición en un vector de variables, que es donde se guarda toda la memoria local. Es un concepto parecido al de la pila de funciones de cualquier programa, cada vez que se llama, se reserva el espacio necesario para todas las variables locales de esa función. Entonces, para acceder a cualquier variable durante la ejecución, el intérprete sólo tiene que usar el índice que el analizador semántico le ha asignado, y acceder al vector de variables locales.

La pila de funciones es muy parecida a la del ASL, cada vez que se llama a una función, ésta se guarda en la pila, junto a un vector de variables locales.

Los tipos de datos se implementan de manera similar al ASL. Todo tipo de dato se representa con una clase, sólo que ésta es abstracta, y cada tipo de dato es subclase de ésta. Con este método, podemos guardar/retornar cualquier variable independientemente

del tipo. Y para acceder a atributos o métodos concretos, se puede realizar downcast sin problemas porque el analizador semántico se asegura que en ese momento los tipos sean los correctos. En cuanto a los operadores, son diferentes a los del ASL ya que el lenguaje permite utilizar los mismos operadores con distintos tipos. Ya que Java está orientado a objetos y permite el polimorfismo, una manera muy simple y fácil de implementarlos es implementar los operadores en la clase que representa cada tipo. Solo queda declarar una función abstracta por operador. Por ejemplo, el operador suma se implementa en una función llamada *sumOperator* que está declarada en la clase abstracta Data, y su subclase Int la implementa con la suma de enteros habitual. De esta manera, en el intérprete lo único que hay que hacer para sumar es llamar a esa función sin importar el tipo, ya que el analizador semántico se encargará de eso.

Como hay tipos que tienen atributos y métodos en el lenguaje (Note, Figure...), existe otra clase abstracta que hereda de Data, el cual representa los tipos que tienen estas características, y tiene funciones que permiten acceder a atributos y a métodos, las cuales se implementarán en sus respectivas subclases.

Para ejecutar instrucciones, también hay que distinguirlas como en la gramática. Por eso hay una función para tratar las instrucciones imperativas y las musicales. Como ambas tienen bastante en común, para ahorrar código hay una tercera función que ejecuta estas instrucciones que se evalúan igual en ambos casos.

Generación del lenguaje MIDI

El proceso de crear un archivo MIDI es bastante largo y complejo. Pero el proceso en general sigue de la siguiente forma: El intérprete guarda en todo momento la pista que se está evaluando. Cada pista guarda los eventos MIDI que se generan durante la interpretación, y los compases de esa pista guardan las figuras para que en caso de que haya un error, se puedan mostrar las notas que contiene. A medida que se va interpretando una pista, se generan intervalos donde esa pista va a necesitar un canal para ser reproducida. Por ejemplo, cuando llamamos a una función, desde que empezó el programa hasta esa llamada se necesita un canal para que suene la pista, pero durante la llamada esa pista deja de sonar, ya que pasará a sonar otra que hereda sus características (instrumento,

volumen, tono...). Al final de la ejecución, el gestor de canales asigna a cada intervalo generado un canal. El algoritmo más óptimo requeriría partir estos intervalos en otros más cortos, pero debido a la complejidad y a la falta de tiempo, usamos una aproximación que aprovecha los canales siempre que puede, pero sin partirlos nunca. Reutilizar canales puede generar problemas, ya que una pista puede estar sonando con un instrumento, y más tarde el mismo canal se usa para otra pista con un instrumento diferente (también pasa con el volumen). Por eso a la hora de poner el canal, la pista ha de tener en cuenta cuál fue el último instrumento de la pista y del canal, y si son diferentes hay que añadir un nuevo mensaje al principio del intervalo. Esta gestión de canales es necesaria debido a que MIDI como máximo puede reproducir 16 canales simultáneamente.

5. Posibles extensiones no implementadas

A pesar de las numerosas posibilidades que ofrece **A Music Language**, no cabe duda de que estas podrían extenderse de muchas maneras. A continuación mencionamos algunas de las características que sería interesante implementar:

Números en coma flotante

AML dispone de representación de enteros con sus correspondientes operaciones aritméticas, pero la adición de floats añadiría nuevas opciones y posiblemente aumentaría la precisión en ciertos aspectos.

Arrays y otras estructuras de datos

Ahora mismo, es posible almacenar datos para usarlos cuando sean necesarios mediante variables. No obstante, sería muy útil disponer de estructuras de datos que permitieran el almacenamiento de cantidades mayores de datos de una manera más simple y menos engorrosa que el uso de numerosas variables. Además, la combinación de estas estructuras de datos con los métodos de control de flujo de los que ya dispone AML agilizarían muchísimo ciertas tareas.

Interpretación del lenguaje

Como hemos mencionado anteriormente, AML es actualmente un lenguaje compilado. En un futuro sería interesante convertirlo en lenguaje que pudiera ser tanto compilado como interpretado, es decir, reproducir el contenido MIDI en vez de generar un fichero que contenga este.

Instrucción Play

Relacionado con el punto anterior cabe destacar que sería interesante la introducción de una instrucción que nos permita reproducir alguna nota, fragmento, canción, ... Cuando desearamos. De esta manera podriamos depurar nuestro codigo más fácilmente o sencillamente jugar con las capacidades del lenguaje.

Adición de nuevos tipos de datos y funciones

A Music Language dispone de ciertos tipos de datos típicos, que aparecen en la mayoría de lenguajes, pero existen otros (como el Char) que no están presentes. Añadir estos tipos de datos, aunque a priori no presenta ninguna ventaja específica, daría más juego y otorgaría más libertad a los programadores, que dispondrán de más opciones para lograr sus objetivos. Por otro lado, sería interesante la adición de más funciones predefinidas. Esto otorgaría a los programadores mayor libertad a la hora de manipular los tipos de dato a su gusto de manera más simple.

Optimización del uso de los canales

Dado que MIDI solo dispone de 16 canales, AML debe gestionar de manera astuta la distribución de pistas en los canales para permitir la reproducción de estas de la mejor manera posible, permitiendo que se ejecuten tantas como sea posible. El algoritmo que utiliza AML gestiona los canales de manera inteligente, y consigue una repartición bastante buena de las pistas sobre los canales. No obstante, es posible que puedan hacerse optimizaciones que aumenten la eficacia de esta repartición y maximicen la ocupación de los canales.

Optimización de los mensajes MIDI

Dependiendo de como se escriba una canción, los mensajes resultantes pueden repetirse o ser redundantes, lo que provoca que el archivo final ocupe más espacio del necesario. Se podría optimizar haciendo un análisis de los mensajes una vez se ha interpretado toda la canción, para poder minimizar la cantidad de mensajes usados.

Eliminar la restricción de 16 canales

A pesar de los algoritmos implementados para distribuir las pistas en los canales, existe una limitación física de MIDI que ningún algoritmo puede resolver, y es el hecho de que dispone de tan solo 16 canales, lo que limita la ejecución a 16 pistas como máximo sonando a la vez en un momento determinado. Idealmente, sería un avance importante incrementar este número, aunque esto requeriría dejar de utilizar MIDI.