FUNDAMENTOS DE

COMPOSICIÓN MUSICAL ASISTIDA

EN EL ENTORNO DE PROGRAMACIÓN OPENMUSIC

PABLO CETTA

UCA

Fundamentos de

Composición Musical Asistida

en el entorno de programación

OpenMusic

Instituto de Investigación Musicológica "Carlos Vega"
Facultad de Artes y Ciencias Musicales
Pontificia Universidad Católica Argentina
"Santa María de los Buenos Aires"

Pablo Cetta

Fundamentos de

Composición Musical Asistida

en el entorno de programación

OpenMusic

FACULTAD DE ARTES Y CIENCIAS MUSICALES

Cetta, Pablo
Fundamentos de composición musical asistida en el entorno de programación OpenMusic / Pablo Cetta. - 1a edición especial - Ciudad Autónoma de Buenos Aires : Educa, 2018.
230 p.; 28 x 20 cm.

Edición para Fundación Universidad Católica Argentina
ISBN 978-987-620-370-8

1. Música. 2. Composición Asistida. I. Título.
CDD 780.982

PONTIFICIA UNIVERSIDAD CATÓLICA ARGENTINA
"SANTA MARÍA DE LOS BUENOS AIRES"

Rector: Dr. Miguel Ángel Schiavone

FACULTAD DE ARTES Y CIENCIAS MUSICALES

Decano: Lic. Ezequiel Hernán Pazos

INSTITUTO DE INVESTIGACIÓN MUSICOLÓGICA "CARLOS VEGA"

Director: Dr. Pablo Cetta

Diseño: Mariela Tzeiman

OpenMusic fue desarrollado por el grupo de Representación Musical del IRCAM
© Carlos Agon, Gérard Assayag, Jean Bresson.

UCA

ISBN: 978-987-620-370-8

Queda hecho el depósito que previene la Ley 11.723
Printed in Argentina - Impreso en la Argentina

Índice

Prólogo 9

Acerca de la Composición Asistida 11

Capítulo 1 - Conceptos básicos de LISP 15

El lenguaje de programación *LISP* 17

Herramientas de programación 17

Listas 18

Tipos de datos 19

Notación 20

Normas de evaluación 21

Operaciones con listas 22

Predicados 25

Condicionales 26

Variables 28

Funciones 30

Mapeo de funciones 31

Funciones anónimas 31

Recursión 33

Equivalencia entre funciones y datos 34

Argumentos opcionales 35

Efectos colaterales de las funciones 35

Macros 36

Iteración 38

Capítulo 2 – El lenguaje *OpenMusic* 43

Generalidades 45

Instalación de los ejemplos 46

Descripción general del entorno de programación 47

Funciones y clases 48

Funciones primitivas de *LISP* en *OpenMusic* 51

Funciones aritméticas y trigonométricas 52

Predicados 54

Funciones *LISP* incluidas en el menú de OM 56

Funciones para el procesamiento de listas 57

Funciones aritméticas 64
Funciones combinatorias 70
Series numéricas 72
Conjuntos 77
Curvas 79
Curvas en dos dimensiones 79
Curvas en tres dimensiones 84
Matrices 87
Control del flujo de datos 93
Condicionales 93
Operadores lógicos 95
Iteración 98
Notación musical 112
Árboles rítmicos 116
Funciones relacionadas con la notación 119
Tratamiento de árboles rítmicos 125
Abstracciones 133
Funciones *lambda* 142
Modos de evaluación en OM 146
Instancias, *slots* y variables globales 149
Slots 150
Variables 153
Mensajes y secuencias MIDI 156
Filtros de eventos 163
Código *LISP* en OM 169

Capítulo 3 – Aplicaciones musicales 173

La práctica de la programación 175
Campos armónicos 175
Multiplicación de acordes 176
Transposiciones limitadas registradas 178
Conjuntos de grados cromáticos 179
Series con hexacordios simétricos 185
Combinatoriedad 187
Gestualidad rítmica 191
Densidad cronométrica 191
Números primos aplicados a la altura y al ritmo 193

Registración espacial de la altura 194
Ritmos a partir de las primeras reflexiones 197
Interpolación rítmica 201
Resíntesis instrumental 202
Multiplicación de espectros 206

Capítulo 4 – La librería *OMMatrix* 209

Matrices combinatorias 211
Matrices simples 212
Matrices complejas 213
OMMatrix 216
Operaciones con PCS 216
Operaciones con matrices combinatorias 220
Generación de matrices 220
Transformación de matrices 224
Impresión de matrices 227
Bibliografía 229

Prólogo

En este libro, Pablo Cetta aborda un área de conocimiento de gran importancia en la creación musical actual. La composición asistida por computadora existe desde que los músicos se interesan en la utilización y desarrollo de aplicaciones informáticas para formalizar los procedimientos y las técnicas que subyacen en la organización sonora. Este trabajo posibilita tanto la comprensión y ampliación de estos recursos, como también la renovación de enfoques de la música existente, realizando de esta manera un significativo aporte a la producción musical contemporánea y futura.

Una de las virtudes a destacar, es que el desarrollo de los temas no es únicamente teórico o abstracto, sino que se realiza a través de un poderoso entorno de programación para música: OM (*OpenMusic*, C. Agon, G. Assayag, J. Bresson, del *Equipo de Representación Musical* del IRCAM). Ya que OM se basa y amplía (mediante funciones y objetos gráficos diseñados *ad hoc* para su aplicación en música) en el lenguaje de programación *LISP*, el primer capítulo presenta una concisa pero muy substancial introducción a dicho lenguaje. En el segundo capítulo se tratan las particularidades del entorno *OM*, con ejemplos de sus funciones "nativas" que permiten comprender su potencial y son esenciales para abordar las aplicaciones específicas que el autor crea y presenta en los capítulos 3 y 4.

Las aplicaciones programadas en OM que se presentan en los capítulos 3 y 4 son originales del autor y se basan en muy diversas formas de organización de la música (algunas de ellas inspiradas en obras de compositores como Pierre Boulez y otras en música del mismo autor del libro). Se podrían clasificar, de manera general, en aquellas que atienden a la organización de la altura (campos armónicos, multiplicación de acordes, combinatoria aplicada a los conjuntos de clases de alturas, organización registral de la altura, etc.), aquellas que atienden a la organización temporal (duraciones, densidad cronométrica, gestualidad e interpolación rítmica, etc.), aquellas que tratan la proyección de los dos mencionados parámetros (altura y duración) al espacio acústico (registración espacial de la altura y derivación de estructuras de duraciones de las primeras reflexiones de un recinto virtual) y, finalmente, aquellas que atienden a la proyección a estructuras de alturas de datos derivados del análisis espectral y a su tratamiento (resíntesis instrumental y multiplicación de espectros). Ante la imposibilidad práctica de tratar en un libro todos los temas concernientes a esta enorme área con la claridad y profundidad que se requiere, el autor realiza una muy pertinente selección de estos, permitiendo así

que el lector obtenga excelente información teórico/práctica y pueda imaginar otras aplicaciones por derivación, ampliación o combinación de las que se presentan.

El autor sigue con gran habilidad en este libro una línea sutil entre programación y estructuras de datos, utilización de un entorno de programación y aplicación en la organización musical. También debe mencionarse que los ejemplos que se presentan no solo son útiles para su aplicación en los estilos o tendencias musicales de los que provienen (e.g. serialismo integral, espectralismo) sino que poseen la solidez y claridad conceptual como para ser extendidos o adaptados a muy variadas estéticas.

Finalmente, no se puede dejar de mencionar un factor clave que le otorga a este trabajo una calidad inusual. Siendo el autor un compositor de trayectoria nacional e internacional destacada, ha explorado extensamente en su obra la gran mayoría de las técnicas que presenta a través de aplicaciones y desarrollos propios, lo que produce en este libro una conjunción virtuosa de su trabajo académico y creativo, que será fascinante para compositores, analistas musicales y especialistas en informática de la música.

Oscar Pablo Di Liscia

Acerca de la Composición Asistida

Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent.

(A. A. Lovelace, 1843)

En 1955 comienzan las primeras experiencias sobre la utilización de las computadoras en música. Los aportes de Caplin y Prinz en composición algorítmica resultan pioneros en este campo, y se inician a través de la implementación en términos computacionales de la obra-tratado de W. A. Mozart, *Musikalisches Würfelspiel* K 516F, a través de la cual es posible generar un sinnúmero de valses diferentes, simplemente arrojando dos dados[1]. Previo a ello, Prinz ya había logrado construir ejemplos simples de síntesis sonora, por lo cual la computadora[2] no sólo demostró poseer la capacidad de generar una estructura melódico-rítmica[3] combinando datos mediante el azar, sino también la habilidad de interpretarla.

---

[1] Mozart escribió *Musikalisches Würfelspiel* (Juego de dados musical, para escribir valses con la ayuda de dos dados, sin ser músico ni saber composición, K 516F) en 1777. Cada una de las versiones posibles de la obra se crea arrojando 16 veces los dados y eligiendo los 8 compases del sector *A* y los 8 del sector *B*. La suma de dos dados da un número comprendido entre 2 y 12, y por ello, cada compás tiene 11 opciones distintas de ser completado. Lo que parece ser algo ingenuo sorprende por sus propiedades combinatorias, pues mediante este método es posible obtener 45.949.729.863.572.161 valses diferentes. Suponiendo que tocáramos rápido cada vals (unos 20 segundos por pieza) tardaríamos en ejecutarlos todos 29.141.127.514 años.

[2] La computadora empleada era una Ferranti Mark I*, adquirida por la empresa Shell en 1953.

[3] Si bien Mozart escribe cada compás del tratado con una textura pianística de melodía acompañada, la versión tecnológica recurre solamente al aspecto melódico, dado que en ese momento no resultaba posible reproducir sonidos simultáneos, ni realizar modificaciones dinámicas.

Otro antecedente emblemático de la programación en favor de la creación musical es el de la *Illiac Suite*, el cuarteto de cuerdas compuesto por Hiller e Isaacson en 1957, empleando distribuciones probabilísticas y cadenas de Markov. También en ese mismo año, Max Mathews concibe el lenguaje de síntesis y transformación sonora *Music*, cuyos descendientes continuaron su desarrollo hasta nuestros días.

Vemos entonces que, desde los inicios, el empleo de las computadoras con fines musicales y sonoros comienza a caracterizarse a través de dos campos de aplicación diferenciados. Uno, el de la composición asistida, cuyos desarrollos más destacados tendrán lugar principalmente en Europa; el otro, referido a la síntesis y al procesamiento de señales de audio a través de lenguajes de programación y aplicaciones, provenientes en su mayoría de Estados Unidos.

Uno de los objetivos principales de la composición asistida por computadora (CAC) es la formalización de los procesos ideados por un compositor, lo cual puede dar como resultado los materiales de una obra, o la obra misma. Se trata de poner en fórmula un proceso mental determinado, que lleva a la construcción de una estructura musical significativa para quien la realiza.

El proceso de formalización tiene lugar a través de la programación, mediante un lenguaje adecuado. Es una tarea en la que necesariamente participan diferentes áreas de conocimiento, y que se ha vinculado en muchas ocasiones con la inteligencia artificial. Seguramente esa asociación explica el porqué del uso de *LISP* en la mayoría de los desarrollos presentes, sumada a ciertas similitudes existentes entre los modos de representar la información, tanto en el contexto musical como en el ámbito propio de ese lenguaje[4].

Aprender a programar y realizar programas no es tarea fácil. Como tampoco lo es detallar uno a uno los pasos que damos cuando pretendemos explicar de qué manera componemos un fragmento de música. En ambos casos estamos obligados a pensar y repensar la música, o al menos a meditar sobre determinados procesos que nos llevan a crearla y a generar nuevas ideas a la vez.

Frente a esta dificultad, suelen surgir algunos interrogantes, tales como quiénes deberían usar estas técnicas y quiénes no, o cuándo es conveniente recurrir a la programación y cuándo no. La respuesta obvia a la primera pregunta sería: todos

---

[4] Las estructuras de datos en forma de árbol, o el empleo de números fraccionarios en *LISP*, son ejemplos de ello.

aquellos que estén interesados. Pero, aun considerando que el compositor elige sus herramientas libremente de acuerdo a sus necesidades, una elección adecuada dependería del conocimiento de todas las posibilidades a su alcance. Particularmente, en el contexto de una etapa formativa.

En primer lugar, hay tantos modos de componer como compositores existen. Siguiendo este criterio, una aplicación cerrada de composición algorítmica sería útil solamente para aquel que la desarrolla. De ahí, la necesidad de emplear lenguajes que nos permitan inventar nuevos programas.

Por otra parte, ciertas ideas resultan imposibles de llevar a la práctica sin la asistencia de un software especialmente creado, ya sea por su grado de complejidad, o por el volumen de datos a considerar. Pensemos, por ejemplo, en prácticas derivadas de la combinatoriedad, del análisis espectral, de las técnicas de síntesis del sonido, del tratamiento de conjuntos de grados cromáticos, del uso de herramientas probabilísticas, entre otros casos. Detengámonos en la posibilidad de reutilizar los métodos que configuran un estilo, o modo de hacer música, y en la forma en la que un programador optimiza el reempleo de su código mediante el uso de clases o funciones. Para muchos, y desde hace tiempo, la programación forma parte de los saberes que hacen a un compositor en la actualidad.

Desde ya, si esa complejidad no existiera, o estuviera puesta en términos distintos –y no por eso menos importantes– a los de la técnica musical, la programación resultaría innecesaria. Y con ello, respondemos al segundo interrogante planteado. No sería lógico crear un programa para llevar a cabo una tarea compositiva simple, no repetitiva, o no reutilizable, dado que hacerla mentalmente o sobre un papel sería más rápido y menos penoso.

Si bien podríamos imaginar que la composición asistida se halla en relación directa con la creación para instrumentos tradicionales, y que la síntesis y procesamiento del sonido se vincula más bien con la producción electroacústica, se observa en la práctica que ambos campos se complementan mutuamente. Hoy en día resulta difícil pensar en una obra instrumental con procesamiento en tiempo real en la cual la programación de los procesos de generación y transformación del sonido no se encuentre estrechamente vinculada con los procedimientos plasmados en la partitura asignada a los intérpretes. Por esta razón, los diferentes lenguajes utilizados en la

actualidad[5] proveen herramientas capaces de combinar los campos mencionados, y los medios para comunicarse con otros lenguajes o aplicaciones especializadas en alguna tarea, como la representación en notación musical o la transformación del dominio del tiempo al de la frecuencia.

Este libro trata la formalización de procedimientos aplicables a la composición y su codificación a través de la programación. Algunos de esos procedimientos se encuentran en la literatura musical contemporánea y han sido empleados de manera generalizada por diversos compositores. Sea o no el caso, los ejemplos presentados intentan ilustrar de manera simple alguna problemática específica, tanto de la programación como de la composición. Si bien hubiera sido posible incluir programas de mayores dimensiones que generaran aplicaciones de uso inmediato en la creación musical, considero que suelen resultar poco didácticos, son de difícil lectura y comprensión para quienes se inician en estas lides y no fomentan el desarrollo de las propias ideas.

El lenguaje escogido para abordar la Composición Asistida por Computadora ha sido *OpenMusic*[6], desarrollado por Carlos Agon, Gérard Assayag y Jean Bresson, del Equipo de Representaciones Musicales del IRCAM. Por estar basado en *Common LISP*, considero importante que comencemos por las generalidades de ese lenguaje. Pues aun dentro del entorno *OpenMusic*, la programación en *LISP* se convertirá en un complemento necesario e indispensable para extender sus posibilidades, e incluso para simplificar su uso.

---

[5] Algunos de los más empleados son *Pure Data*, *Max-MSP*, *SuperCollider*, *Csound*, *OpenMusic* y *PWGL*.

[6] http://repmus.ircam.fr/openmusic/home

CAPÍTULO 1

Conceptos básicos de *LISP*

El lenguaje de programación *LISP*

Las bases del lenguaje *LISP* fueron establecidos por John McCarthy en 1958. Es el segundo lenguaje más antiguo de los que se hallan actualmente en uso. La versión de mayor difusión en esa época fue *LISP* 1.5, publicada por el mismo McCarthy[7].

Ya en la década de 1980 existía una gran cantidad de dialectos, por lo cual se decide establecer una versión estandarizada, a la cual se denominó *Common LISP*. La forma más popular que se emplea hoy en día es *ANSI Common LISP*, diseñada originalmente por Guy Steele[8].

Herramientas de programación

Cualquiera sea la aplicación que utilicemos para escribir nuestros programas en lenguaje *LISP*, contaremos con un editor interactivo o intérprete. A través del mismo podremos escribir sentencias y obtener resultados de forma inmediata.

Un intérprete *LISP* funciona de acuerdo a un bucle continuo denominado *read-eval-print*. Cuando escribimos una expresión y presionamos la tecla *enter*, el programa lee el contenido, lo evalúa, y posteriormente devuelve un resultado.

Cada vez que ingresamos código, el programa intérprete debe traducirlo a un lenguaje comprensible por el procesador. Sin embargo, cuando escribimos programas reales, no lo hacemos a través de cortas sentencias introducidas en la ventana del intérprete, sino directamente en un archivo de texto (*.LISP*) que luego es compilado en un nuevo archivo. La compilación es una traducción eficiente a lenguaje de máquina del código *LISP* que escribimos en el archivo de texto, al cual denominamos código fuente. Podemos concluir, entonces, diciendo que *LISP* es un lenguaje cuyo código puede ser tanto interpretable como compilable.

A lo largo de este capítulo utilizaremos solamente el aspecto interpretable de *LISP*. A fin de poder comprobar los ejemplos, recomiendo instalar previamente el entorno

---

[7] McCarthy, John, Abrahams, Paul W., Edwards, Daniel J., Hart, Timothy P., and Levin, Michael I., *Lisp 1.5 Programmer's Guide*, 2nd ed., MIT Press, Cambridge, MA, 1965.

[8] Steele, Guy et al. *Common Lisp, the Language, 2nd Edition*. Digital Press, Bedford, MA. 1990.

de programación *LISPWorks*[9]. Una vez instalado el software, accederemos al intérprete a través de la ventana denominada *listener*.

Listas

El término *LISP* es acrónimo de *LISt Processing*, con lo cual podemos inferir la importancia que tuvieron las listas en los orígenes de este lenguaje. Cualquier expresión encerrada entre paréntesis conforma una lista. Una lista puede estar formada por átomos (números, conjuntos de caracteres, caracteres especiales), y a la vez puede contener a otras listas, es decir, listas dentro de listas en una estructura de árbol.

```
(1 4 7 8 0)
(celeste 32 blanco)
((oid mortales) el (grito sagrado))
```

La longitud de una lista se establece a partir de la cantidad de elementos contenidos en el primer nivel de paréntesis. La lista `(aprendo LISP)` posee dos elementos, mientras que la lista `(A (B C) D)` posee tres, pues el segundo elemento es a su vez una nueva lista, de dos átomos.

Una lista sin elementos se denomina lista vacía `()`, y se representa internamente con el símbolo `NIL`. `NIL` y `()` son equivalentes. La lista `(A NIL F)` puede representarse también como `(A () F)`.

La representación de las listas en la computadora se realiza mediante las denominadas células *cons*. Una célula *cons* consta de dos mitades, la primera se llama *CAR* y la segunda *CDR*[10]. Se trata de punteros que indican dónde se encuentra el primer elemento de la lista, y dónde continúa la lista, respectivamente. El fin de la lista apunta a `NIL`.

La lista `(A B)`, por ejemplo, posee dos células *cons*, dispuestas de la siguiente manera.

---

[9] *LispWorks* es una herramienta de desarrollo de aplicaciones en *LISP*. Puede obtenerse una versión limitada y gratuita para uso personal en http://www.lispworks.com

10 CAR (*Contents of Adress portion of Register*) y CDR (*Contents of Decrement portion of Register*) toman sus nombres de los registros de una primitiva computadora donde corría *LISP* en sus orígenes, la IBM 704.

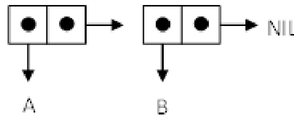


Figura 1. Representación interna de la lista (A B)

Además de tratarse de punteros, *CAR* y *CDR* son funciones propias de *LISP*, que devuelven el contenido de las direcciones donde apuntan. Por ejemplo, el resultado de aplicar car a la lista (uno dos tres) es UNO. El resultado de aplicar cdr, es el resto de la lista, (DOS TRES).

Un modo de crear células *cons*, es a través de la función cons. Si aplicamos esta función al símbolo uno y a la lista (dos tres), obtendremos nuevamente (UNO DOS TRES).

Tipos de datos

Algunos tipos de objetos de datos utilizados en *LISP* son los siguientes.

- *Números*. Los tipos de datos numéricos son *integer* (entero), *ratio* (fraccionario), *floating-point* (número con decimales) y *complex* (complejos).
- *Caracteres*. Se especifican con la forma #c\a (letra *a*).
- *Símbolos*. Son secuencias de letras, dígitos y pueden contener caracteres especiales. Los siguientes son caracteres especiales válidos: +, -, *, /, @, $, %, ^, &, \, <, >. La palabra rueda, por ejemplo, es un símbolo. Los símbolos se organizan en paquetes (denominados *packages*), en los cuales cada símbolo es único, e identificable a través de su nombre. *LISP* traduce los nombres en minúsculas a mayúsculas, pues no hace distinción entre ambos tipos. Los símbolos sirven para distintos propósitos, particularmente se usan para representar variables.
- *Listas*. Pueden contener números, símbolos, u otras listas.
- *Cadenas de caracteres* o *strings*. Son secuencias de caracteres que representan texto, y se distinguen por ir entre comillas.
- *Funciones*. Son procedimientos que invocados a través de su nombre, devuelven valores a partir de los datos aportados (argumentos de la función). En *LISP*, en contraste con otros lenguajes, una función puede ser argumento de otra, ser devuelta como resultado de otra función, ser creada

durante la ejecución del programa, o ser almacenada en una variable. De ahí que sea considerada como un tipo de dato.

Otros objetos son *arrays* (matrices de datos), *vectores* (matrices unidimensionales), *hash tables* (estructuras preparadas para la búsqueda rápida de información), *packages* (colecciones de símbolos que determinan un espacio de nombres), *pathnames* (medios para interactuar con el sistema de archivos de la computadora, de forma independiente del sistema operativo).

En resumen, las expresiones en *LISP* pueden contener listas o átomos. Los átomos, a su vez, pueden estar constituidos por símbolos o por números. Los números pueden ser de tipo entero o fraccionario (números racionales), decimales y complejos.

Notación

El código *LISP* se basa en la denominada notación polaca[11], también conocida como notación *prefix*, la cual establece que los operadores anteceden a los argumentos. Al expresar una suma, por ejemplo, solemos escribir los argumentos separados por el símbolo del operador: 2 + 3 + 4. En la notación *prefix*, en cambio, el operador se ubica al principio, y a continuación los argumentos: `+ 2 3 4`.

En *LISP* utilizamos paréntesis para delimitar las expresiones. Una expresión típica es `(* 5 6)`, en la cual observamos una lista con tres elementos. El primero es el operador de multiplicación (la función), y los otros dos representan a los números a multiplicar (los argumentos). Si evaluamos esta lista, obtendremos el número `30` como resultado.

El empleo de paréntesis nos permite combinar diferentes operaciones. En el ejemplo que sigue, el resultado se calcula de adentro hacia afuera, es decir, resolviendo las operaciones desde los paréntesis internos hacia los externos.

```
> ( / (+ 3 5) 2)
4
```

[11] La notación polaca toma su nombre de la nacionalidad de Jan Lukasiewicz, especialista en lógica que en 1924 desarrolló este tipo de escritura.

El intérprete, al evaluar una lista, espera hallar en el primer elemento de la misma un operador o el nombre de una función capaz de efectuar alguna acción válida. De no hallarlo arroja un error, como en la expresión siguiente.

```
>  (1 hola 3)
ERROR
```

Pero para comprender mejor esto, veamos cuáles son las reglas de evaluación con las cuales se maneja el intérprete.

Normas de evaluación

1. Tanto los números, como los símbolos especiales `T` (verdadero) y `NIL` (falso) son evaluados como sí mismos: `5` es evaluado como `5`, `T` es evaluado como `T`, y `NIL` es evaluado como `NIL`.
2. Cuando evaluamos una lista, el primer elemento de la lista es considerado una función a invocar, y el resto de los elementos se consideran los argumentos de esa función. Por ejemplo, `(/ 4 2)`. El primer símbolo (/) es el llamado a la función división, y el resto, los números a dividir. En el caso de `(> (* 2 5) 8)` la evaluación devuelve `T`, pues es verdadero que el producto de 2 por 5 es mayor que 8.
3. La evaluación de los símbolos devuelve el contenido de la variable a la que el símbolo se refiere. Obviamente, tendremos que declarar la variable como tal y asignarle un contenido específico, como ya veremos. De otro modo, el intérprete nos dará un mensaje de error por desconocer ese elemento.

A fin de impedir que una expresión sea evaluada se emplea un apóstrofe (*Alt* + 39), o bien la función `quote`.

```
>  '(1 2 3)
(1 2 3)

>  (quote (1 2 3))
(1 2 3)
```

En el ejemplo anterior, si quitáramos el apóstrofe, el intérprete declararía un error, dado que según las reglas de evaluación el primer elemento de una lista deber ser el nombre de una función, y el número 1 no lo es. Debemos, entonces, comenzar a

distinguir entre listas de datos (a las cuales es necesario ponerles el apóstrofe) y listas con llamadas a funciones y argumentos de esas funciones.

Veamos otro ejemplo para aclarar esto. `second` es el nombre de una función que devuelve el segundo elemento de una lista. Si escribimos `(second (uno dos tres))` y presionamos la tecla *enter*, el intérprete indicará un error, pues no conoce a la función `UNO`. Para obtener el segundo elemento de la lista dato es necesario evitar la evaluación de la misma, y que ésta sea considerada como argumento de la función `second`. Debemos colocar, entonces, el apóstrofe delante de ella, y escribir:

```
> (second '(uno dos tres))
DOS
```

Vemos, nuevamente, que los símbolos en minúscula son devueltos en mayúscula, debido a que *LISP* no hace distinción de tipos.

Operaciones con listas

Según mencionamos al presentar las listas, la función `cons` sirve a su construcción[12].

```
> (cons 'a nil)
(A)

> (cons 'a '(b c d))
(A B C D)
```

Otra función empleada habitualmente para la formación de listas es `list`.

```
> (list 'a 'b 'c 'd)
(A B C D)

> (list 'a '(b c) 'd)
(A (B C) D)
```

Y la longitud de una lista la determinamos con `length`.

```
> (length '(A (B C) D))
3
```

---

[12] En el ejemplo de aplicación de la función `cons` agregamos un elemento al principio de una lista. Puede observarse, sin embargo, que si escribimos el elemento a al final –`(cons '(b c d)'a)`– se produce una "lista punteada" –`((B C D) . A)`– que es el modo en que *LISP* indica que el último elemento de la lista no apunta a `NIL`. Si escribiéramos, en cambio `(cons '(b c d)(cons 'a nil))`, obtendríamos `((B C D) A)`.

Como también vimos, las funciones `car` y `cdr` nos permiten obtener el primer elemento y el resto de la lista, respectivamente.

```
> (car '(A (B C) D))
A

> (cdr '(A (B C) D))
((B C) D)
```

Idéntico resultado podemos lograr con las funciones `first` y `rest`, que devuelven el primer elemento, y el resto de una lista (todos los elementos menos el primero).

```
> (first '((A B) c (d e f)))
(A B)

> (rest '((A B) c (d e f)))
(C (D E F))
```

Si deseamos obtener el segundo elemento de una lista podríamos apelar a `car` y `cdr` de forma combinada, como en el ejemplo.

```
> (car (cdr '(1 2 3 4)))
2
```

O al tercero mediante

```
> (car (cdr (cdr '(1 2 3 4))))
3
```

Estas combinaciones de `car` y `cdr` se encuentran resumidas en expresiones tales como `caar`, `cadr`, `cdar`, `cddr`, `caaar`, `caadr`, etc. Los ejemplos anteriores podrían abreviarse así:

```
> (cadr '(1 2 3 4))
2

> (caddr '(1 2 3 4))
3
```

Dado que estas expresiones resultan un tanto confusas, solemos extraer los distintos elementos de una lista mediante, `first`, `second`, `third`, `fourth` hasta llegar al décimo (`tenth`). Más allá del décimo, o incluso para los anteriores, también podemos aplicar `nth`, que devuelve el enésimo término, considerando a la posición del primero elemento como 0, a la del segundo como 1, etc.

```
> (third '(1 2 3 4))
3

> (nth 0 '(1 2 3 4))
1

> (nth 3 '(1 2 3 4))
4
```

La función `last`, por otra parte, devuelve la última célula *cons*. Esto significa que retorna el último elemento apuntando a *NIL*, por lo cual el elemento es devuelto en forma de lista.

```
(last '(1 2 3 4))
(4)

> (last '(1 2 3 4 (a b)))
((A B))
```

La función `butlast` retorna la lista dato sin las últimas *n* células *cons*. Si *n* se omite, asume que vale 1, y devuelve la lista sin el último elemento.

```
> (butlast '(1 2 3 4))
(1 2 3)

> (butlast '(a b c d e) 3)
(A B)
```

Luego, podemos remover un elemento de una lista con `remove`. La función quita todas las apariciones del elemento especificado como argumento.

```
> (remove 67 '(54 67 38))
(54 38)
```

La concatenación de dos o más listas es posible a través de `append`.

```
> (append '(1 2 3) '(4 5 6))
(1 2 3 4 5 6)
```

Mientras que `revappend` retrograda la primera lista antes de concatenarla con la segunda.

```
> (revappend '(1 2 3) '(a b c))
(3 2 1 A B C)
```

`subst` efectúa la sustitución de objetos en un árbol. En el ejemplo, el número `1` es reemplazado por el símbolo `uno`.

```
> (subst 'uno 1 '(1 (3 4 1) (5 1 2 3 4) 2))
(UNO (3 4 UNO) (5 UNO 2 3 4) 2)
```

En relación con los conjuntos, la función `union` devuelve todos los elementos pertenecientes a dos listas dadas, descartando las repeticiones; la función `intersection` devuelve los elementos comunes y `set-difference` presenta los elementos no comunes a ambas listas.

```
> (union '(fa sol la do mi) '(do re sol si mi))
(SI FA SOL LA DO MI RE)

> (intersection '(fa sol la do mi) '(do re sol si mi))
(MI DO SOL)
```

```
> (set-difference '(fa sol la do mi) '(do re sol si mi))
(LA FA)
```

Para saber si un elemento pertenece o no a una lista empleamos `member`. Si el elemento no se encuentra en la lista arroja `NIL`. Si lo encuentra, devuelve el resto de la lista comenzando por el ítem hallado.

```
> (member 'bach '(mozart beethoven brahms))
NIL

> (member 'beethoven '(mozart beethoven brahms))
BEETHOVEN BRAHMS
```

Predicados

Los predicados son funciones destinadas a comprobar la veracidad o falsedad de determinadas condiciones sobre sus argumentos. En general los nombres de predicados finalizan con la letra *p*. Si deseamos saber si un argumento dado es un número, por ejemplo, podemos recurrir a `numberp`, lo cual nos permite comprobar el tipo de dato.

```
> (numberp 3.14)
T

> (numberp "hola")
NIL
```

Entre los predicados más comunes encontramos `listp` (es una lista), `symbolp` (es un símbolo), `integerp` (es un número entero), `rationalp` (es un número fraccionario), `floatp` (es una número decimal), `evenp` (es un número par), `oddp` (es impar), `zerop` (es un cero), `plusp` (es un número positivo), `minusp` (es negativo), `atom` (es un átomo), `stringp` (es una cadena de caracteres). Frente a estas afirmaciones podremos obtener `T` o `NIL`.

Otros predicados que admiten dos o más argumentos son los destinados a comparar números: =, <, >, <=, >=.

```
> (> 2 5)
NIL
```

La función = es válida para la comparación de números, pero no para otros objetos (símbolos o listas, por ejemplo). Por lo tanto, existen los predicados de igualdad `equal`, `eq`, `eql` y `equalp`. La función `equal` arroja verdadero cuando dos estructuras "lucen igual", o mejor dicho, son isomórficas.

```
> (equal '(a b c) '(a b c))
T
```

Ambas listas son iguales, sin embargo, al compararlas con `eq`,

```
> (eq '(a b c) '(a b c))
NIL
```

En este caso, `eq` determina que no son iguales porque sus argumentos son iguales sólo si se trata del mismo objeto, vale decir, si ocupan el mismo lugar en memoria. Respecto a los símbolos, *LISP* almacena una única copia de cada uno de ellos, por lo cual,

```
> (eq 'nota 'nota)
T
```

La función `eql` es similar a `eq`, pero también se usa para comparar números teniendo en cuenta el tipo, y en el caso de caracteres, si son en mayúscula o en minúscula.

```
> (eql 4 4.0)
NIL
```

`equalp`, por último, es similar a `equal`, y sus diferencias se perciben principalmente en los tipos de datos no tratados en este capítulo (como matrices y *hash-tables*). Por otra parte, no realiza distinción entre tipos numéricos, ni distingue mayúsculas de minúsculas.

```
> (equalp 4 4.0)
T
```

Condicionales

A fin de tomar decisiones, podemos establecer condiciones a través de `if`. En el ejemplo siguiente usamos el predicado `oddp` para determinar si un número es impar. Si la condición se cumple, `if` devuelve el primer símbolo (impar), caso contrario, el segundo (par).

```
> (if (oddp 1) 'impar 'par)
IMPAR
```

Cuando una condición es verdadera y se debe ejecutar el código que la acompaña, la sentencia `if` admite solamente una única expresión (`'impar`, por ejemplo, o una única lista). Si, en cambio, varias sentencias deben ser consideradas es necesario recurrir a la forma especial `progn`.

progn evalúa cada uno de sus argumentos en el orden en que fueron escritos, y luego devuelve el valor de la última evaluación. En términos más simples, es capaz de contener varias expresiones en situaciones donde se espera encontrar sólo una.

En el ejemplo que sigue, si el argumento ingresado es un número se imprimen las cadenas de caracteres y se le suma 1 a ese número. Si el dato no fuera numérico, el programa respondería simplemente con NIL.

```
> (if (numberp 5)
      (progn
          (print "Se trata de un número")
          (print "Vamos a sumarle 1")
          (+ 5 1))
      nil)

"Se trata de un número"
"Vamos a sumarle 1"
6
```

Según se observa, incluimos la sentencia print, la cual nos sirve para mostrar las cadenas de caracteres en el editor. Por otra parte, es de destacar el uso de tabulaciones (indentación) que aplicamos al código, las cuales ayudan a su lectura y comprensión.

Una variante de if es when, que consta de una condición y un cuerpo de instrucciones. Si la condición resulta verdadera se procede a evaluar el código contenido en el cuerpo de esta macro. Veamos un ejemplo similar al anterior, pero utilizando when. Aquí notamos que el operador progn ya no resulta necesario.

```
> (when (numberp 5)
        (print "Se trata de un número")
        (print "Vamos a sumarle 1")
        (+ 5 1))

"Se trata de un número"
"Vamos a sumarle 1"
6
```

Lo opuesto a when es unless. En esta *macro*[13] el código del cuerpo sólo es evaluado si la condición evaluada es falsa.

Cuando es necesario evaluar una serie de condiciones hasta encontrar una que sea verdadera podemos utilizar cond. Si alguna condición se cumple, se evalúan las sentencias que la acompañan. En el siguiente ejemplo, vemos la definición de una

[13] Una macro es una extensión del lenguaje, que permite ampliar sus posibilidades. Son series de instrucciones que se invocan a través de un nombre, y suelen emplearse en tareas repetitivas. Más adelante nos referiremos a ellas con mayor precisión.

función que devuelve el número de semicorcheas que entran en distintas figuras (más adelante nos referiremos a las funciones con mayor detalle). Según vimos, la función `equal` devuelve `T` si los símbolos son iguales. En nuestro caso, si no se conoce la figura, el programa responderá con la cadena de caracteres "Desconozco a esa figura".

```
> (defun semicorcheas (x)
       (cond  ((equal x 'redonda) 16)
              ((equal x 'blanca) 8)
              ((equal x 'negra) 4)
              ((equal x 'corchea) 2)
              ((equal x 'semicorchea) 1)
              (t "Desconozco a esa figura")))

SEMICORCHEAS

> (semicorcheas 'negra)
4

> (semicorcheas 'verde)
"desconozco a esa figura"
```

Se observa en la función que la última opción es `t` (*true*). Mediante este artilugio, si la función no encuentra en sus opciones la figura ingresada como dato, se fuerza una respuesta verdadera (en este caso "desconozco a esa figura"). Es necesario contemplar detenidamente la distribución de paréntesis de `cond`, dado que es común cometer errores al emplearlo.

Variables

En programación, una variable es un espacio de memoria caracterizado por un identificador, que puede almacenar datos de diverso tipo. Esos datos pueden cambiar durante la ejecución de un programa, por lo tanto, a una variable se le pueden asignar distintos valores.

Existen dos clases principales de variables: globales y locales. El valor de una variable global puede ser leído desde cualquier lugar dentro de un programa. Las variables locales, en cambio, son accesibles solamente en un determinado ámbito o porción del programa.

En *LISP*, las variables se establecen a partir de símbolos. El símbolo `do-mayor`, por ejemplo, no fue definido como una variable, ni tiene asignado inicialmente un valor. Si lo evaluamos obtendremos un error.

```
> do-mayor
Error: The variable DO-MAYOR is unbound
```

Para convertir un símbolo en una variable deberemos realizar una declaración. Las variables globales se declaran con la macro `defvar`, o con `defparameter`. También es posible asignarles directamente un valor, en nuestro ejemplo una lista de notas.

```
> (defvar do-mayor '(do re mi fa sol la si))
DO-MAYOR
```

Según se aprecia, la lista de notas lleva apóstrofe para evitar la evaluación de la misma, y para que sea considerada un dato. Si no escribiéramos el apóstrofe obtendríamos un error, dado que de acuerdo a las reglas de la evaluación el primer elemento de la lista debe ser una función. Una vez que la variable fue declarada es posible evaluarla sin inconvenientes, obteniendo como resultado su contenido.

```
> do-mayor
(DO RE MI FA SOL LA SI)
```

Los nombres de variables globales en *LISP* suelen escribirse entre asteriscos, por ejemplo `*re-menor*`. Esto permite reconocer sus identificadores y su condición global fácilmente.

Las variables locales, por otra parte, se definen con `let`.

```
> (let ((compas 5) (sistema 2))
       (list compas sistema) )
(5 2)
```

En este caso definimos dos variables locales (`compas` y `sistema`) y les asignamos los valores 5 y 2, respectivamente. Posteriormente, creamos una lista con sus contenidos. Si observamos detenidamente la sintaxis, vemos que el paréntesis que antecede a `let` es cerrado al final de toda la expresión. Esos paréntesis de apertura y cierre delimitan el ámbito donde las variables, definidas con `let`, son accesibles. Fuera de ese ámbito, si utilizamos los símbolos `compas` o `sistema` obtendremos un error.

Si deseamos que el valor de una variable dependa del contenido de otra previamente definida, utilizamos `let*` en lugar de `let`. En el ejemplo siguiente, la variable `frec-la5` toma su valor del contenido de la variable `frec-la4`. Luego, se crea una lista con símbolos -anticipados cada uno por un apóstrofe- y con el contenido numérico de `frec-la5`, cuyo nombre no lleva apóstrofe pues remite a una variable.

```
> (let* ((frec-la4 440) (frec-la5 (* frec-la4 2)))
       (list 'la 'frecuencia 'de 'la5 'es frec-la5 'Hz))
(LA FRECUENCIA DE LA5 ES 880 HZ)
```

Puede inferirse, entonces, que las variables locales ayudan a resolver conflictos producidos por nombres repetidos en distintos ámbitos de un mismo programa, a la vez que ayudan a optimizar el uso de memoria. Incluso, en un ámbito donde existe una variable local, una variable global del mismo nombre no tiene efecto.

Para modificar el contenido de variables, tanto locales como globales, la expresión más usada es setf.

```
> (defvar *afinacion* 440)
*AFINACION*

> (setf *afinacion* 435)
435
```

Funciones

Una función cuenta con un nombre, puede o no incluir argumentos, y posee un cuerpo donde se desarrolla su contenido. Los argumentos, si bien son variables, no precisan ser declarados. Su ámbito es local y sólo pueden ser leídos dentro del cuerpo de la función. Creamos nuestras propias funciones mediante la macro `defun`.

```
> (defun suma (a b) (+ a b))
SUMA

> (suma 4 5)
9
```

La siguiente función calcula el grado cromático[14] de un número de nota MIDI. Para extraer el grado empleamos la función `mod` (módulo) con divisor 12. Recordemos que la función `mod` devuelve el resto de una división entera. Aplicando la función `grado` a la nota MIDI 71 notamos que se trata del grado cromático 11 (nota *si*) [15].

```
> (defun grado (nmidi) (mod nmidi 12))
GRADO

> (grado 71)
11
```

---

[14] Los grados cromáticos son 12 y se numeran del 0 al 11. El 0 es *do*, el 1 es *do#*, el 2 es *re*, y así sucesivamente hasta llegar al 11, que es *si*. Aplicando módulo 12 obtenemos el grado cromático que corresponde a un número de nota MIDI. Las notas MIDI se numeran de 0 a 127, y al *do* central corresponde el número 60.

[15] 71 / 12 = 5, y el resto es 11. El resto representa al grado cromático y el resultado (5) a la octava en la que se encuentra. Respecto a esto último, es preciso tener en cuenta que en MIDI la octava central es 5 y no 4, como solemos identificarla habitualmente.

En esta función hay un único argumento (*nmidi*) y su cuerpo consiste simplemente en la llamada a la función `mod`. Para utilizar la función creada, simplemente hemos especificado su nombre y sus argumentos, dentro de en una lista.

Mapeo de funciones

Qué sucede si queremos aplicar la función `grado` que escribimos antes a todos los elementos de una lista, y no solo a un único número de nota MIDI. Si escribimos `(grado '(71 29 43 64 75))` el intérprete nos dará un error, pues nuestra función acepta solamente un único número como argumento. En ese caso, deberemos recurrir a otra función que aplique `grado` a todos los elementos de la lista, y esa función es `mapcar`.

```
> (mapcar #'grado '(71 29 43 64 75))
(11 5 7 4 3)
```

Observamos una combinación de símbolos `#'` antes del nombre de la función a aplicar, lo cual podríamos considerar como equivalente del apóstrofe (`quote`) pero para las funciones.

A continuación, vamos a crear una función que separe las notas MIDI de un grupo de sublistas, las cuales contienen tres valores: nota MIDI, *key velocity* y número de canal MIDI. La nota MIDI ocupa la primera posición de cada sublista y la extraeremos con `first`. Por otra parte, accederemos a las sublistas de forma iterativa empleando `mapcar`.

```
> (defun extraer-notas-midi (lista)(mapcar #'first lista))
EXTRAER-NOTAS-MIDI
```

El empleo de la función sobre una lista arroja el siguiente resultado.

```
> (extraer-notas-midi '((71 64 1)(60 64 1)(65 64 1)))
(71 60 65)
```

Funciones anónimas

Un tipo de funciones empleadas frecuentemente en *LISP* son las anónimas, también llamadas *lambda*. Las funciones *lambda* son invocadas por otras funciones o macros que admiten como argumento a una función, como es el caso de `mapcar`.

Si escribimos una función *lambda* que tome un número como argumento y lo eleve al cuadrado, y la evaluamos, el intérprete la reconocerá, pero no nos será posible llamarla pues precisamente no tiene nombre, es anónima.

```
(lambda (n)(* n n))
#<anonymous interpreted function 21B564CA>
```

Sin embargo, un modo de invocar funciones es a través de `funcall`, que llama a una función y le pasa los argumentos. Por ejemplo, la función de suma:

```
> (funcall #'+ 1 3 5)
9
```

Ahora, en lugar de la suma podríamos utilizar una función anónima, como la que multiplica a un número por sí mismo.

```
> (funcall #'(lambda (n)(* n n)) 4)
16
```

De este modo, utilizamos una función de modo abreviado –la declaración y la llamada se encuentran en la misma línea– sin necesidad de declararla y asignarle un nombre con `defun`. Vemos entonces cómo una función puede invocar a otra, y en este caso anónima.

También podríamos emplear una función *lambda* con `mapcar`. Para comprobarlo tomaremos el contenido de la función `extraer-notas-midi`, que creamos anteriormente, y le agregaremos la posibilidad de transportar la nota MIDI en *n* semitonos.

```
> (defun extraer-notas-midi-transp (lista n)
       (mapcar #'(lambda (lista)(+ (first lista) n)) lista))
EXTRAER-NOTAS-MIDI-TRANSP

> (extraer-notas-midi-transp '((71 64 1)(60 64 1)(65 64 1)) 7)
(78 67 72)
```

Vemos en el código, en letra negrita, que donde antes ubicábamos a la función `first` ahora colocamos una función anónima que extrae el primer elemento de una lista y le suma el valor *n*. `mapcar`, por otra parte, proyecta a la función anónima sobre cada elemento de la lista (en nuestro caso, sobre las sublistas).

En la llamada a `extraer-notas-midi-transp` los argumentos son el contenido de la lista a procesar y el número 7, por lo cual cada nota MIDI extraída es transpuesta a la quinta.

En el ejemplo que sigue, empleamos `mapcar` para realizar la suma de dos listas. Observamos, al evaluar la expresión, que el proceso se interrumpe cuando finaliza la lectura de la lista de menor cantidad de elementos.

```
> (mapcar #'+ '(1 2 3) '(10 20 30 40 50))
(11 22 33)
```

Analicemos, por último, otro ejemplo en el cual pasamos una primitiva como argumento de nuestra función (+, -, * ó /) con sus propios argumentos, para luego invocarla.

```
> (defun aritmetica (funcion a b) (funcall funcion a b))
ARITMETICA

> (aritmetica #'* 2 5)
10

> (aritmetica #'+ 2 5)
7
```

Hemos visto, entonces, que las funciones pueden operar también como argumentos de otra función, y en este caso hablamos de funciones de orden superior. Las funciones constituyen el pilar más sobresaliente de *LISP*, y por ello es considerado un lenguaje de programación funcional.

Recursión

Un tipo especial de funciones son aquellas que pueden invocarse a sí mismas. Se denominan funciones recursivas. En el siguiente ejemplo, la función `diga-un-numero` pide al usuario que ingrese un valor numérico impar. La función `read` nos permite leer lo ingresado, y `let` lo almacena en la variable `numero`. A través de la sentencia `if` establecemos la condición `oddp`: si el número ingresado es impar (verdadero) el intérprete devuelve `GRACIAS` (símbolo que no sigue las reglas de la evaluación por tener apóstrofe); si el número es par (falso), se evalúa la expresión que sigue, la cual llama nuevamente a `diga-un-numero`, para volver a empezar.

```
> (defun diga-un-numero ()
       (princ "Ingrese un número impar: ")
       (let ((numero (read)))
           (if (oddp numero)
                  'Gracias
                  (diga-un-numero))))

Ingrese un número impar: 1
GRACIAS
Ingrese un número impar: 2
Ingrese un número impar:
```

Para imprimir en la ventana del *listener* utilizamos ahora `princ`, que elimina las comillas de la cadena de caracteres. Por otra parte, no sólo empleamos una variable local (`numero`) sino también que –a diferencia de los ejemplos anteriores- incluimos dos listas en el cuerpo de la función (una correspondiente a `princ` y la otra a `let`, ésta última con sublistas en su interior).

Un empleo más elaborado de función recursiva es la que corresponde al cálculo de factorial[16] de un número. Si observamos detenidamente el código, veremos un condicional donde preguntamos si el número (*n*) es cero. De ser así, se devuelve el valor 1 y finaliza la recursión. De ser falso, se ejecuta la sentencia siguiente, en la cual se multiplica el valor actual de *n* por un nuevo llamado a la función factorial, con *n* decrementado en 1.

```
> (defun fact (n)

        (cond ((zerop n) 1)
              (t (* n (fact (- n 1))))))
FACT

> (fact 5)
120
```

Podemos verlo de este modo: que el factorial de 5 es igual a 5 multiplicado por el factorial de 4. Y a su vez, que el factorial de 4 es 4 por el factorial de 3, y así hasta llegar a 1. El factorial de 1 es 1 por factorial de 0, que por definición equivale a 1, según se lee en el condicional de la función.

Equivalencia entre funciones y datos

Las funciones son objetos, como los símbolos, las cadenas de caracteres o las listas. Esto permite que podamos asignar una función a una variable, como si se tratara de un valor cualquiera, lo cual expresa cierta equivalencia entre funciones y datos.

---

[16] La operación factorial de un número equivale a multiplicar todos los enteros positivos en orden decreciente, partiendo del número dado, hasta llegar a 1. Por ejemplo, el factorial de 5 se escribe 5! y equivale a 5.4.3.2.1 = 120. El factorial de un número *n* puede emplearse para calcular todos los ordenamientos posibles –permutaciones- de *n* elementos diferentes (como ser notas, secciones de una obra, compases, etc.).

```
> (defun suma (a b)(+ a b ))
SUMA

> (setf mi-variable #'suma)
#<interpreted function SUMA 2009DC82>

> (funcall mi-variable 6 7)
13
```

Argumentos opcionales

En ocasiones es más efectivo que una función asuma argumentos con valores por defecto, o ningún valor, y que en la llamada a la función se decida si especificar o no los parámetros opcionales.

```
> (defun acorde (i iii v &optional vii) (list i iii v vii))
ACORDE

> (acorde 'do 'mi 'sol)
(DO MI SOL NIL)

> (acorde 'do 'mi 'sol 'si)
(DO MI SOL SI)
```

En el ejemplo anterior no se establece ningún valor por defecto para la séptima del acorde. Por lo cual, si no especificamos la nota que la conforma, la función le asigna `NIL`. Veamos cómo podríamos asignar ese valor por defecto.

```
> (defun acorde (i iii v &optional (vii 'si)) (list i iii v
vii))
ACORDE

> (acorde 'do 'mi 'sol)
(DO MI SOL SI)

> (acorde 'do 'mi 'sol 'sib)
(DO MI SOL SIB)
```

Efectos colaterales de las funciones

Decimos que una función produce un efecto colateral cuando efectúa algún tipo de acción diferente a la de retornar el valor para el cual la función es creada. Algunos efectos colaterales son útiles, mientras que otros no, dado que suelen generar errores de programación.

La siguiente función eleva un número al cubo, y no produce efectos colaterales.

```
> (defun cubo (x) (* x x x))
```

La que sigue, en cambio, asigna un nuevo valor a la variable local de la función (un efecto colateral), que es posible evitar.

```
> (defun mas10 (x)
       (setf x (+ 10 x))
       (list 'el 'resultado 'es x))
MAS10

> (mas 10 9)
(EL RESULTADO ES 19)

> (defun mas10 (x) (list 'el 'resultado 'es (+ 10 x)))
MAS10
```

Macros

En *LISP*, una forma es cualquier objeto destinado a ser evaluado, y a producir uno o más valores. Las formas se dividen en tres categorías: autoevaluatorias (números, los cuales se evalúan como sí mismos), símbolos (que representan variables) y listas. Las listas, a su vez, pueden dividirse en otras tres categorías: formas especiales, llamadas a macros y llamadas a funciones.

Las formas especiales son expresiones que no siguen las normas generales de evaluación. Entre ellas encontramos a `if`, `let` y `quote`. El conjunto de formas especiales es limitado y el programador no puede crear otras nuevas. Sin embargo, sí es posible crear nuevas construcciones sintácticas a través de las macros.

Una macro es una especie de función que, al expandirse, genera código *LISP*. Al proceso de conversión de sus sentencias en código *LISP* se lo denomina *expansión* de la macro. Dado que las macros se invocan y devuelven algo a cambio, pueden ser confundidas con las funciones. La diferencia principal reside en que las funciones retornan valores, mientras que las macros devuelven expresiones.

Si bien el desarrollo de macros es de importancia en la ampliación del lenguaje y en la creación de programas de cierta extensión, aquí veremos sólo los conceptos básicos.

Para comenzar, supongamos que deseamos crear una macro que le asigne el valor 4 a una variable cualquiera. La llamaremos `vale4`, y la invocaremos como si se tratara de una función: `(vale4 x)`.

Como ya sabemos, una de las formas que corresponde a la asignación de un valor a una variable es

```
> (setf x 4)
4
```

A través de una macro podríamos crear esa misma forma, pero generando el código. Definimos nuestra macro con `defmacro`.

```
> (defmacro vale4 (x)
       (list 'setf x 4))
> VALE4
```

La macro, al expandirse, transforma `(list 'setf x 4)` en el resultado de su evaluación: `(setf x 4)`.

Cada vez que en nuestro programa aparezca `(vale4 variable)`, se convertirá en `(setf variable 4)`.

En las macros, así como empleamos el apóstrofe para evitar una evaluación, podemos utilizar la coma para forzarla. La coma interrumpe momentáneamente la acción del apóstrofe. Por lo tanto, en el ejemplo anterior, donde dice

```
(list 'setf x 4)
```

podríamos escribir directamente

```
`(setf ,x 4)
```

La coma, en este caso, habilita la evaluación de la variable `x`.

```
> (setf mivariable 9)
9

> (defmacro vale4 (x)
       `(setf ,x 4))
VALE4

> (vale4 mivariable)
4

> mivariable
4
```

Para observar y asegurarnos que la expresión que una macro devuelve es la esperada, recurrimos a la función `macroexpand-1`, que retorna el resultado que surge de la expansión de nuestra macro.

```
> (print (macroexpand-1 '(vale4 x)))
(SETF X 4)
```

Iteración

En la programación en *LISP*, la repetición de una acción determinada puede llevarse a cabo con `do`. La macro `do`, al igual que `let`, permite crear variables dentro de la lista que conforma el primer argumento de la macro. El símbolo de la variable, su valor inicial y la forma en que esa variable se actualiza, se encuentran en una sublista, pudiendo haber más de una de esas sublistas dentro de la lista que opera como primer argumento, vale decir, más de una variable con su valor inicial y modo de actualización.

El segundo argumento es otra lista de expresiones. La primera de esas expresiones establece la condición a partir de la cual la iteración debe detenerse. Las restantes, son las acciones a cumplir como parte del bucle. En el ejemplo siguiente utilizamos `do` en una función que calcula la frecuencia de los primeros *n* armónicos de una fundamental dada.

```
> (defun armonicos (fundamental n)
       (do ((x 1 (+ x 1))
              ((> x n) 'finalizado)
              (format t "Nro.: ~A  Frecuencia: ~A~%" x (* x
                   fundamental))))
ARMONICOS

> (armonicos 100 10)
Nro.: 1  Frecuencia: 100
Nro.: 2  Frecuencia: 200
Nro.: 3  Frecuencia: 300
Nro.: 4  Frecuencia: 400
Nro.: 5  Frecuencia: 500
Nro.: 6  Frecuencia: 600
Nro.: 7  Frecuencia: 700
Nro.: 8  Frecuencia: 800
Nro.: 9  Frecuencia: 900
Nro.: 10  Frecuencia: 1000
FINALIZADO
```

En el ejemplo se crea una única variable (`x`), cuyo valor inicial es 1 y su actualización ocurre para cada repetición del bucle, donde se incrementa en 1 `(+ x 1)`. Si la condición `(> x n)` es verdadera, se muestra el símbolo `finalizado`. Si en cambio es falsa, se evalúa la lista siguiente, a través de la cual se imprime el número de armónico y la frecuencia que corresponde a cada iteración del bucle `do`.

Para imprimir los resultados utilizamos la función `format`, que produce una salida formateada, en nuestro caso al *listener*. Según se observa en el texto entre comillas, `~A` es utilizado para mostrar el contenido de las variables dentro del texto, y `~%` para

generar un salto de línea. Las variables se ubican luego del cierre de comillas, y son *x* (número de armónico) y su frecuencia, que resulta de multiplicar el número de armónico por la frecuencia de la fundamental.

Una macro útil para recorrer los elementos de una lista de datos es `dolist`. Como primer argumento se debe especificar una variable donde se alojará cada elemento de la lista; el segundo argumento es la lista a procesar y el tercero (opcional) es una forma a ejecutar al finalizar el bucle. Veamos un ejemplo a través del cual se determina si la nota *do* se encuentra en una lista de notas MIDI, y de ser así se imprime la octava correspondiente.

```
> (setf secuencia '(71 89 75 60 59 63 72 78))

> (dolist (nota secuencia 'finalizado)
      (if (= (mod nota 12) 0)
         (format t "Octava: ~A~%" (truncate (/ nota 12)))))

Octava: 5
Octava: 6
FINALIZADO
```

Si el módulo 12 de la nota MIDI es igual a 0 significa que la nota es *do*. Luego, la división entera entre la nota y 12 arroja el número de octava. Mediante `dolist` pudimos recorrer la lista completa.

Vemos que existe cierta similitud entre `dolist` y `mapcar`, pues ambas nos permiten aplicar un procedimiento a cada elemento de una lista. No obstante, en algunos casos, la segunda puede llegar a ser más eficiente que la primera, ya que almacena los resultados directamente en una lista.

```
> (setf secuencia '(71 89 75 60 59 63 72 78))
> (mapcar #' (lambda (nota)
       (if (= (mod nota 12) 0) (truncate (/ nota 12))))
              secuencia)

(NIL NIL NIL 5 NIL NIL 6 NIL)
```

Con `dolist`, para obtener un resultado similar deberíamos realizar la recolección de resultados de forma explícita.

```
> (let ((resultado nil))
       (dolist (nota secuencia (reverse resultado))
              (if (= (mod nota 12) 0)
                     (push (truncate (/ nota 12)) resultado))))
(5 6)
```

Mediante `push` enviamos los números de octava a la variable `resultado`, previamente definida con `let`. En el tercer argumento de `dolist` escribimos qué queremos que la macro devuelva al finalizar el recorrido por la lista. Obviamente, lo

que se almacenó en resultado, pero retrogradado, dado que push almacena cada valor al principio de la lista, y no al final.

Si precisáramos simplemente encontrar la primera nota *do*, y no todas, deberíamos establecer una cláusula que nos permitiera salir del recorrido por la lista una vez hallada la primera coincidencia. Mediante return es posible finalizar la ejecución del bucle.

```
> (dolist (nota secuencia)
      (if (= (mod nota 12) 0)
          (progn
              (format t "Octava: ~A~%" (truncate (/ nota 12)))
              (return "Salgo del bucle"))))
Octava: 5
"Salgo del bucle"
```

Siguiendo con las macros diseñadas para realizar tareas repetitivas, cuando una iteración debe darse una cantidad determinada de veces (cinco, en el ejemplo) podemos emplear dotimes.

```
> (dotimes (x 5 'fin) (format t "El valor de x es ~A ~%" x))
El valor de x es 0
El valor de x es 1
El valor de x es 2
El valor de x es 3
El valor de x es 4
FIN
```

Por último, trataremos la macro loop, la cual posee tantas posibilidades que prácticamente constituye otro lenguaje dentro de *LISP*. Entre sus cláusulas[17] contamos con la capacidad de crear variables locales; incrementar o decrementar variables numéricas; recorrer los elementos de una lista; recolectar, contar o sumar valores, encontrar el valor máximo o mínimo de un grupo de valores; la posibilidad de decidir cuándo finalizar la iteración y la opción de incluir código a ejecutar antes de la iteración o después de la misma. Cada una de estas acciones está determinada por una palabra clave, como veremos luego.

La expansión de la macro loop puede contener tres partes: un prólogo, el cuerpo del bucle propiamente dicho y el epílogo. El prólogo contiene las formas que serán ejecutadas antes del bucle, entre las cuales se encuentra la posibilidad de crear variables locales. El cuerpo está formado por aquellas sentencias que forman parte de

---

[17] Denominamos *cláusula* a cualquier segmento sintáctico incluido en el loop. Las cláusulas pueden tener *preposiciones*, tales como from o to.

la repetición del bucle y el epílogo ejecuta aquello que es posterior a la finalización de la iteración.

A las palabras clave `do` (asociada a ejecutar determinadas tareas, sin condicionamientos), `initially` (ejecutar acciones antes de iniciar la iteración) y `finally` (ejecutar acciones después de finalizar la iteración) puede seguir un número ilimitado de formas *LISP*. Al resto, sólo una.

La palabra clave para la declaración de variables es `with`, que opera de forma similar a `let`. El avance por pasos a través de una variable numérica (incremento o decremento) se logra con `for` o `as`. Podemos ejecutar más de un avance por pasos mediante `and`, y el término `repeat`, por otra parte, nos permite iterar un proceso un número determinado de veces.

Para la recolección, concatenación, suma o cuenta de datos nos valemos de `collect`, `append`, `sum` y `count`. Los términos `minimize` y `maximize` los empleamos cuando deseamos obtener el valor mínimo y máximo, respectivamente, de una lista de datos.

Pero mejor veamos algunas aplicaciones, antes de continuar, para ir aclarando estas cuestiones. En el siguiente ejemplo incrementamos la variable `i` mediante `for`, y recolectamos los cuadrados de sus valores.

```
> (loop for i from 1 to 5
        collect (* i i))
(1 4 9 16 25)
```

Podemos modificar la forma en la que se produce el incremento o decremento a través de preposiciones, como `downto`, `upto`, `downfrom`, `upfrom` o `by`, según se aprecia en los ejemplos que siguen.

```
> (loop for i from 3 downto 1 do (print i))
3
2
1
NIL
```

O bien,

```
> (loop for i downfrom 3 to 1 do (print i))
3
2
1
NIL
```

Y en orden ascendente:

```
> (loop for i from 1 upto 3 do (print i))
3
2
1
NIL
```

O bien,

```
> (loop for i upfrom 1 to 3 do (print i))
3
2
1
NIL
```

También aumentando el paso, con `by`.

```
> (loop for i upfrom 0 to 7 by 3 do (print i))
0
3
6
NIL
```

En el siguiente caso sumamos los valores que asume la variable `i`.

```
> (loop for i from 0 to 10 sum i)
55
```

Pudiendo también contar la cantidad de repeticiones.

```
> (loop for i from 0 to 10 count i)
11
```

Las cláusulas que imponen las condiciones para finalizar la repetición de un bucle son `for` y `as`, según ya mencionamos, `repeat` (el ciclo finaliza luego de *n* repeticiones), `while`, `until`, `always` (repite siempre hasta encontrar un `nil`), `never` (lo opuesto al anterior) y `thereis` (similar a `never`, salvo que retorna el valor que no fue `nil`, y no `nil` como lo hace `never` al finalizar el bucle).

De este modo damos por finalizada esta introducción al lenguaje de programación *LISP*, que sin duda será de utilidad para alcanzar un mayor nivel de comprensión de los temas que trataremos más adelante.

CAPÍTULO 2

El lenguaje *OpenMusic*

Generalidades

OpenMusic (OM), creado originalmente por Carlos Agon, Gérard Assayag y Jean Bresson es un lenguaje destinado al desarrollo de aplicaciones de asistencia en la composición musical. Puede entenderse como una extensión de *Common LISP* y *Common LISP Object System*[18] para usos musicales, con el agregado de diversas herramientas gráficas que lo convierten en un entorno especializado de programación visual.

OM nace en los años 90 a partir de un antecesor que implementa la programación visual en *LISP*: *PatchWork*[19]. Actualmente, OM continúa su desarrollo a través del Equipo de Representación Musical[20] del IRCAM[21] y de quienes contribuyen a su expansión mediante librerías externas.

A través de OM es posible programar en *LISP*, siguiendo el modelo de objetos interconectables mediante cables virtuales, propios de los lenguajes basados en una interfaz gráfica. Sobre esta base, diversas funciones y clases extienden su empleo al tratamiento de las estructuras musicales y del sonido, lo cual convierte al entorno en una herramienta eficaz para la formalización de procedimientos compositivos y la representación de resultados en notación musical.

Por otra parte, OM se caracteriza por su capacidad de comunicarse a través de la norma MIDI o el protocolo OSC (*Open Sound Control*) y de importar o exportar datos a través de formatos de intercambio como ETF, Music XML o SDIF, entre otros. Asimismo, cuenta con maquetas de secuenciación, que permiten que las estructuras sonoras o musicales creadas puedan ser dispuestas en el tiempo.

---

[18] *Common LISP Object System* (CLOS) es una extensión de *Common LISP* destinada a la programación orientada a objetos. Como tal, se estructura en base a *clases* que determinan la forma y el comportamiento de los objetos que de ellas derivan, denominados *instancias*. Las clases actúan como modelos sobre los cuales los objetos se crean. El paradigma de la programación orientada a objetos tiene como objetivo el desarrollo de código altamente estructurado, basado en conceptos tales como el de herencia y de polimorfismo.

[19] *PatchWork* fue creado a fines de los 80 por Mikael Laurson, Jaques Duthen y Camilo Rueda.

[20] http://repmus.ircam.fr/home

[21] https://www.ircam.fr/

OpenMusic dispone de diversas librerías propias y de terceros[22], que extienden las posibilidades del lenguaje. Estas librerías aportan numerosos objetos destinados tanto a la implementación de técnicas compositivas como a la conexión con otros lenguajes especializados y programas.

Los archivos fuente de OM son de descarga libre y gratuita bajo licencia pública GNU, y cuenta con instaladores para sistemas operativos *Mac OSX*, *Windows* y *Linux*, los cuales pueden obtenerse en http://repmus.ircam.fr/openmusic/download.

La documentación oficial del lenguaje se encuentra en http://support.ircam.fr/docs/om/om6-manual/co/OM-User-Manual.html. El sitio del IRCAM provee, además, un tutorial[23] en línea y un Manual de Referencia, que pueden ser útiles para observar sus principales posibilidades y características.

Una vez que el lector se encuentre mínimamente familiarizado con el entorno de programación recomiendo leer atentamente la documentación disponible, dado que allí encontrará información necesaria para el manejo de *OpenMusic*. En este libro, he preferido no redundar en una mera traducción de lo que allí se encuentra, con el propósito de concentrar nuestra atención en cuestiones específicas de la programación.

Instalación de los ejemplos

Para acceder a los ejemplos que integran este libro, y que a continuación estudiaremos, deberá descargar el *workspace* correspondiente, del sitio

www.pablocetta.com/openmusic.php

Luego de descomprimir el archivo *WSLibroCetta* obtendrá una carpeta que podrá copiar donde lo considere conveniente, recordando el directorio utilizado ya que al

[22] Las librerías de OM se encuentran disponibles en http://repmus.ircam.fr/openmusic/libraries

[23] Los ejemplos de ese tutorial pertenecen a K. Haddad, M. Malt y J. Baboni-Schilingi. Para incluir los ejemplos del tutorial en nuestro *workspace* nos dirigimos al menú *Help/Import Tutorial Patches*. También dentro del mismo menú *Help* encontramos un *link* a la documentación en línea, al Manual de Referencia de OM y a una lista de atajos que facilitan la edición de los *patches*.

abrir OM deberá dirigirse a ese mismo lugar para proceder a la apertura del espacio de trabajo.

En la página web mencionada podrá acceder también a una librería denominada *OMMatrix* especialmente creada para nuestros ejercicios, que una vez descomprimida se copia en el subdirectorio *libraries* del directorio donde se instaló OM[24]. Una vez copiada la carpeta, se abrirá la aplicación OM y se habilitará su inclusión desde el menú *OM6.14/Preferences*. Al desplegarse el cuadro de diálogo de preferencias, elegiremos la pestaña *Libraries* y tildaremos la opción *OMMatrix*. Cada vez que iniciemos la aplicación la librería se cargará automáticamente.

Descripción general del entorno de programación

La programación en OM comienza a partir de la carga o la creación del área de trabajo, denominada *Workspace*. Al iniciar OM aparece un cuadro de diálogo cuyas opciones son: abrir el último *workspace* utilizado, abrir otro existente o crear uno nuevo. Al crear un espacio de trabajo nuevo, OM nos pedirá que seleccionemos el lugar del disco de la computadora donde será almacenado, y su nombre. Junto al área de trabajo se desplegará otra ventana denominada *listener*, que cumple una función similar a la que ya utilizamos con *LISPWorks*.

El área de trabajo no es más que una ventana capaz de contener accesos a nuestros programas –denominados *patches*– los cuales pueden hallarse sueltos u organizados mediante carpetas, y representados con íconos o listas de nombres. Todos los elementos que vemos en el *workspace* se encuentran almacenados en el disco, en la subcarpeta *elements* del directorio elegido para guardar el espacio de trabajo. En general, utilizamos un *workspace* diferente para cada proyecto.

Si partimos de un *workspace* nuevo, el cual se encontrará vacío, podremos crear un *patch* desde el menú *File/New Patch*. Caso contrario, al hacer doble *click* sobre cualquier *patch* preexistente, se desplegará la ventana que muestra su programación. Los *patches* contienen *boxes*, pequeñas cajas que representan funciones, datos, subprogramas y objetos tales como notas o archivos de sonido.

La forma más común de agregar *boxes* a un *patch* es a través del menú, tanto el principal, que se encuentra en la parte superior de la ventana, como el que aparece al

[24] En una PC, normalmente se encuentra en C:\Program Files\OM 6.14\libraries

presionar el botón derecho del mouse sobre el fondo de la ventana (menú contextual). Otra forma es haciendo doble *click* sobre el fondo de la ventana y escribiendo el nombre en el elemento de entrada de datos que aparece.

En el menú se observa que las cajas disponibles se dividen en dos grandes grupos: clases y funciones. Ambos tipos cuentan con un cuerpo principal – la caja propiamente dicha- y entradas y salidas de información, a las que llamamos *inlets* y *outlets*, respectivamente. A partir de éstos conectamos unas cajas con otras, mediante cables virtuales.

La evaluación de nuestros programas se realiza seleccionado la caja que devuelve los resultados –en general la última de una cadena de objetos interconectados- y presionando la tecla "v". De acuerdo a las funciones o clases que hayamos utilizado, el resultado podrá aparecer en la misma caja (en el caso de aquellas que sirven para representar los datos en notación musical, por ejemplo) o en el *listener*. En aquellas cajas que contienen más de un *outlet*, podremos evaluar cada una de sus salidas individuales presionando *Control* (o *Cmd*, en Mac) y haciendo *click* sobre el *outlet*. La evaluación de cada una de una serie de cajas interconectadas, incluso con bifurcaciones, se realiza siempre de abajo hacia arriba, y de izquierda a derecha. Por otra parte, cuando deseamos obtener información de referencia sobre una caja, podemos seleccionarla y presionar la tecla "d".

Funciones y clases

Por estar basado en *Common LISP* y en CLOS, *OpenMusic* no es solamente un lenguaje funcional, sino que además es un lenguaje orientado a objetos. Este tipo de lenguajes se distingue por el uso de clases, que contienen atributos (datos) y métodos (funciones). Veamos algunas definiciones de estos términos.

El concepto que tenemos de un automóvil, por ejemplo, podría ser aquello que da forma a una clase denominada "automóvil"; se trata de algo abstracto. La clase puede entenderse como un molde o modelo cuyos atributos podrían ser el color, el peso o el número de puertas, y sus métodos el acelerar, frenar o doblar. De las clases surgen los objetos, distinguibles unos de otros por sus características particulares: un auto rojo, con 5 puertas, caja automática, determinada aceleración, etc.

Este modo de pensar la programación puede ser perfectamente aplicable al ámbito de la música. El concepto de acorde puede ser entendido como una clase, de la cual

derivan los acordes particulares, denominados *instancias* de la clase, o simplemente objetos.

El gráfico siguiente muestra dos representaciones de la clase `chord`[25]. La primera es tal cual aparece al ubicar la caja dentro de la ventana de programación. La segunda muestra su contenido luego de seleccionarla y presionar la tecla "m" (*miniview*)[26]. Según se aprecia, la clase viene provista, por defecto, de una simple nota.

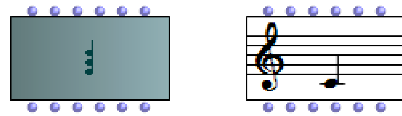


Figura 2.

Clase `chord`

Para crear una instancia de una clase simplemente basta con evaluarla. Cualquiera sea la caja que seleccionemos, al presionar "v" repetidas veces notaremos que en el *listener* aparecen números distintos en sistema hexadecimal, que distinguen a cada una de las instancias generadas. En OM, a las cajas que representan a las clases se las denomina *factory boxes*, es decir, cajas capaces de fabricar objetos.

Las cajas de fabricación de objetos, en general, disponen de un editor que permite ver su contenido y modificarlo. Para acceder al editor de `chord` simplemente hacemos doble *click* sobre la misma caja.

En la figura siguiente vemos que a la clase anterior le aportamos datos (notas MIDI[27] en *midicents*, es decir, centésimas de semitono) que definen a un objeto concreto: un acorde mayor sobre *do*. Haciendo *Shift* + *click* sobre cualquier *inlet* es posible abrir un cuadro en blanco, con su respectivo cable, para introducir datos escribiendo directamente sobre él. Luego de evaluar la caja del acorde veremos que

---

[25] La clase chord se encuentra en el menú *Classes/Score*.

[26] El contenido mostrado en la ventana *miniview* puede ser desplazado mediante *Alt* y las flechas del teclado.

[27] En una clase `chord` las notas se especifican mediante una lista conectada al segundo *inlet*. Luego trataremos en detalle las entradas y salidas de esta caja.

las notas, declaradas previamente mediante números, pasan a notación musical. De este modo, hemos generado una instancia con un acorde particular.

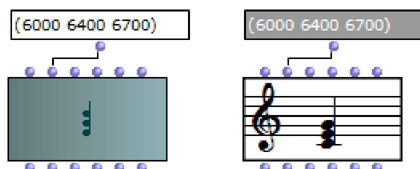


Figura 3.

Instancia con motas expresadas en *midicents*

Funciones primitivas de *LISP* en *OpenMusic*

Según mencionamos antes, *OpenMusic* es un lenguaje de programación visual construido sobre un intérprete y compilador de *Common LISP*. Gracias a ello, es posible utilizar no sólo las herramientas que OM aporta, sino también las prestaciones de *LISP*, ya sea a través de código simbólico (instrucciones en forma de texto) o gráfico (objetos virtuales interconectables).

Para acceder a las funciones de *LISP* creamos un nuevo *patch*, y sobre la ventana hacemos doble *click* sobre el fondo. En el pequeño editor que aparece podemos escribir el nombre de la función *LISP* a invocar. Comencemos con una simple suma (+).

Al crear el objeto visual de suma, que representa a la función *LISP*, observamos que no posee entradas por donde ingresar los números a sumar. Seleccionándolo y presionando *Alt* + -> (o la tecla >) podremos crear tantas entradas como deseemos. Contrariamente, reducimos el número de entradas con *Alt* + <- (o la tecla <).

Para ingresar los datos existen varios métodos. Uno de ellos, es haciendo un *click* sobre el círculo de la entrada y escribiendo el número, el cual quedará oculto al presionar el fondo de la ventana de programación. Si bien el dato no es visible, al pasar el puntero del mouse sobre el circulito el contenido aparecerá. Otro método es haciendo *Shift* + *click* sobre el *inlet*. Según ya vimos, se despliega un editor -unido por un cable a él- en el cual podremos ingresar el valor a sumar.

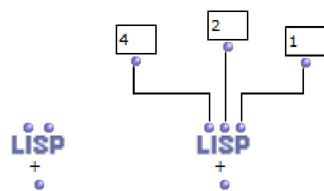


Figura 4.

Empleo de las funciones primitivas de *LISP*

Sólo resta evaluar el *patch* que hemos creado, para obtener el resultado. Para ello seleccionamos el objeto de suma y presionamos la tecla *v*, o bien hacemos *Control* +

click en la salida del objeto. En la ventana del *listener* (denominada OM *Listener*[23]) obtendremos el resultado de la adición. Nuestro programa ha sido interpretado. A las funciones propias del lenguaje *LISP* las denominaremos funciones primitivas.

Funciones aritméticas y trigonométricas

En los ejemplos del *patch* **01 – primitivas *LISP*** podremos observar distintas operaciones con números. A continuación, detallamos sus nombres y descripción:

`-`	Resta
`*`	Multiplicación
`/`	División
`1+`	Incrementa un número en 1
`1-`	Decrementa un número en 1
`sqrt`	Raíz cuadrada
`expt`	Potenciación
`log`	Logaritmo natural (en base *e*). Si se agrega otra entrada es posible especificar la base del logaritmo.
`truncate`	Elimina la parte fraccionaria (la redondea a cero)
`round`	Redondea un número decimal al entero más próximo
`ceiling`	Redondea un número decimal al entero mayor
`abs`	Devuelve el valor absoluto de un número
`min`	Valor mínimo de un grupo de números
`max`	Valor máximo de un grupo de números
`mod`	Módulo (resto de la división entera)
`numerator`	Numerador de un número fraccionario
`denominator`	Denominador de un número fraccionario
`rational`	Convierte decimal a fraccionario
`sin`	Seno de un ángulo en radianes
`cos`	Coseno de un ángulo en radianes

Allí podremos evaluar cada uno de los objetos y observar los resultados en el *listener*. Además, cambiar los argumentos con el propósito de recordar las distintas operaciones.

---

[23] La ventana del *listener* se abre automáticamente al iniciar OM. En caso de haberla cerrado, puede acceder a ella a través del menú *Windows > Lisp Listener*.

A modo de ejercitación vamos a crear un *patch* que calcule la frecuencia en Hertz de una nota MIDI, utilizando las funciones aritméticas primitivas de *LISP*. Para calcular la frecuencia de una nota MIDI nos valemos de la siguiente fórmula:

$$frecuencia = 440 * 2^{(notaMIDI-69)\,/\,12}$$

El *patch* **02 – nota midi a hz** contiene la programación que sirve de solución a nuestro problema. Seleccionando y evaluando el último objeto de la cadena (*) obtendremos la frecuencia de la nota especificada en la parte superior. El resultado de interpretar nuestro programa se observa en la ventana OM *Listener*.

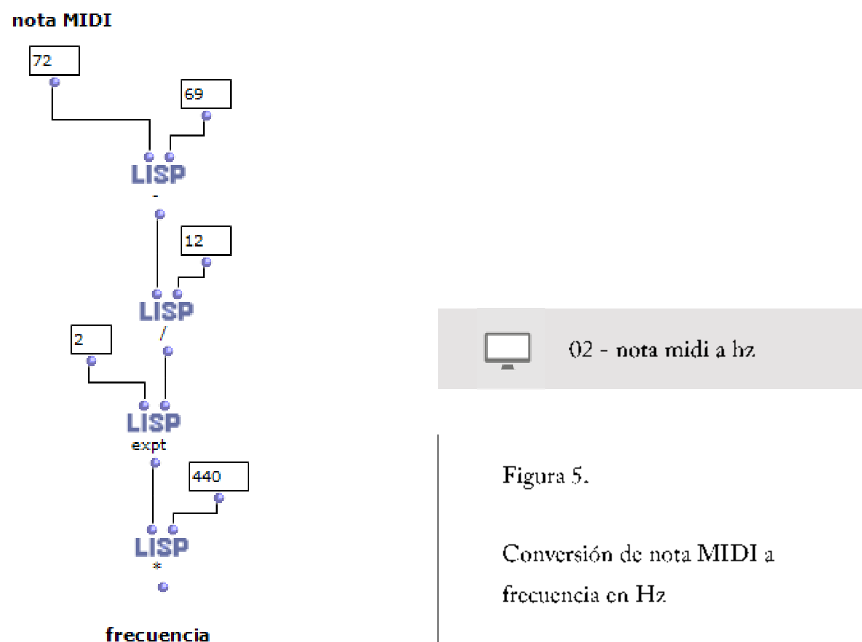


Figura 5.

Conversión de nota MIDI a frecuencia en Hz

La solución en *LISP* al mismo problema, y el resultado de la evaluación, sería la siguiente. La nota MIDI empleada como dato es la 72 (un *do* una octava arriba del *do* central).

```
> (* 440 (expt 2 (/ (- 72 69) 12)))
523.2511
```

Podemos comparar ambos ejemplos y notar que en *OpenMusic* la representación visual del flujo de datos es más clara, mientras que el código simbólico de *LISP* resulta notablemente más conciso.

Predicados

Dentro de las funciones primitivas se encuentran los predicados. Según vimos anteriormente, devuelven `T` o `nil` en respuesta a la veracidad o falsedad de un enunciado.

`evenp`	el número es par
`oddp`	el número es impar
`numberp`	es un número
`symbolp`	es un símbolo
`listp`	es una lista
`atom`	es un átomo
`plusp`	el número es positivo
`minusp`	el número es negativo
`zerop`	el número es cero
`floatp`	el número es decimal
`rationalp`	el número es fraccionario
`endp`	es una lista vacía
`tree-equal`	los árboles son iguales
`=`	es igual a
`/=`	es distinto a
`>`	es mayor que
`<`	es menor que
`>=`	es mayor o igual a
`<=`	es menor o igual a

La función `tree-equal` emplea dos argumentos, que son las listas a comparar. Los comparadores numéricos que siguen en la lista, por otra parte, utilizan dos argumentos (*a* es mayor que *b*, por ejemplo) o más. En este último caso, la comparación es de cada argumento con el siguiente; si el resultado de la comparación es verdadero en todos los pares sucesivos comprobados, la evaluación devuelve verdadero. En *LISP* lo representaríamos del siguiente modo.

```
> (< 2 4 8)
T
```

2 es menor que 4 y 4 es menor que 8, devuelve verdadero.

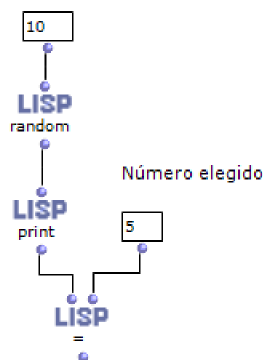
```
> (< 2 8 4)
NIL
```

2 es menor que 8 pero 8 no es menor que 4, por lo tanto devuelve falso.

En los ejemplos del *patch* 03 – **predicados** encontramos los objetos relacionados con los predicados listados. A través de la modificación de los argumentos podremos analizar el comportamiento en cada caso.

Vamos ahora a crear un *patch* en *OpenMusic* que compare un número generado aleatoriamente entre 0 y 9 con otro dado, y determine si son iguales. Utilizaremos la función primitiva `random`, que devuelve números al azar entre cero y el valor utilizado como argumento, menos uno (números aleatorios entre 0 y *N*-1).

La función `random` debe llevar como argumento el número 10 para que devuelva números al azar entre 0 y 9. Luego tenemos que comparar ese resultado con un número elegido, por ejemplo 5, para determinar si hubo o no coincidencia. Para visualizar el número obtenido al azar podemos utilizar la función primitiva `print`; el objeto imprime en el *listener* el resultado de la evaluación de aquello que le conectamos.



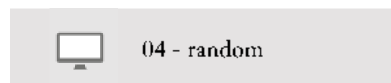


Figura 6.

Números generados al azar

La versión en código *LISP*, y un resultado posible, es la siguiente. Según se aprecia, no fuimos beneficiados por el azar.

```
> (= (print (random 10)) 5)
9
NIL
```

Funciones *LISP* incluidas en el menú de OM

Desde el menú *Fuctions/Kernel/LISP* podemos acceder a diversas primitivas de *LISP*:

`first`, `second`, `third`, `nth`, `rest`, `nthcdr`, `butlast`, `reverse`, `length`, `list`, `remove`, `cons`, `append`, `apply`, `funcall`, `mapcar` y `mapcan`.

Estas funciones han sido tratadas en el capítulo dedicado al lenguaje, a excepción de `nthcdr`, `apply` y `mapcan`, que a continuación definimos.

`nthcdr` devuelve la cola de la lista que resulta de aplicar `cdr` un número determinado de veces. El siguiente ejemplo en *LISP* ilustra lo antedicho. Al procesar con `cdr` a `(a b c)` nos queda `(b c)`, y al aplicarlo nuevamente `(c)`:

```
> (nthcdr 2 '(a b c))
(C)
```

`apply` es similar a `funcall`, ya vista. Aplica una función dada a cada uno de los elementos de una lista.

```
> (apply '+ '(1 2))
3
```

Y por último, `mapcan` es similar a `mapcar`, también estudiada. Dadas dos o más listas como dato y una función, `mapcan` aplica esa función al primer elemento de cada lista, luego al segundo, y así hasta agotar los elementos de la lista más corta. Para ello, es necesario que la función utilizada devuelva el resultado en forma de lista.

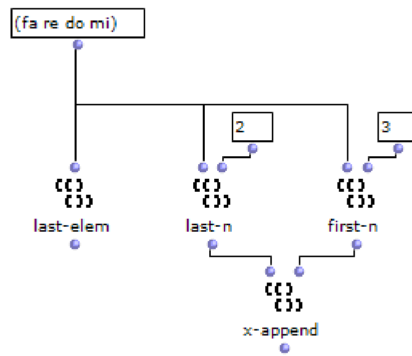
```
> (mapcan #'list '(0 2 4) '(do re mi))
(0 DO 2 RE 4 MI)
```

En el *patch* **05 – funciones kernel *LISP*** encontraremos un ejemplo sencillo de aplicación de las funciones del menú.

Funciones para el procesamiento de listas

En el menú *Functions/Basic Tools/List Processing* hallamos diversas funciones destinadas al tratamiento de listas.

La función `last-elem` devuelve el último elemento de una lista. De modo similar, `last-n`, retorna los *n* últimos elementos de la lista, y `first-n` los primeros. Por otra parte, `x-append` concatena átomos o los elementos de una lista en otra lista. En el ejemplo siguiente vemos las cajas correspondientes a estas funciones en un *patch*. Por un lado, obtenemos el último elemento de una lista (*mi*), mientras que los tres objetos restantes son usados para concatenar los dos últimos elementos de la lista de datos con los tres primeros, resultando `(do mi fa re do)`.



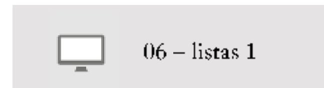


Figura 7.

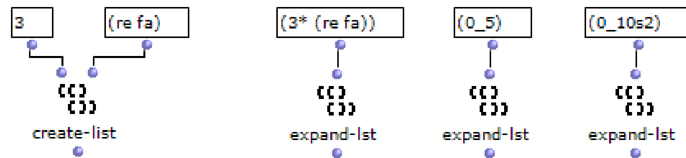
Funciones para el procesamiento de listas

`create–list` crea una lista de elementos repetidos si especificamos la cantidad de repeticiones en el primer *inlet* y el elemento a repetir en el segundo. La lista `(re fa)` repetida tres veces da como resultado `((re fa) (re fa) (re fa))`.

Otro modo de generar listas es a través de `expand-lst`. En este caso, la función expande un modelo propuesto. Las dos formas de expansión posibles son:

a) Repetición de un elemento o una lista. Si en su *inlet* especificamos `(3* (re fa))` obtenemos `(re fa re fa re fa)`. Vemos que es similar a lo que produce `create–list`, si bien esta última repite los elementos sin los paréntesis que los contienen.

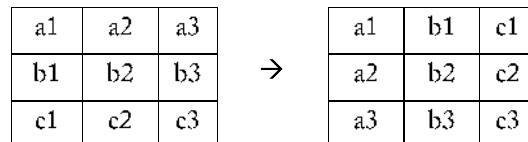
b) Generación de series numéricas. Si introducimos la expresión `(0_5)` obtenemos la lista `(0 1 2 3 4 5)`. De igual manera, podemos también aplicar una razón a la sucesión de números, como en `(0_10s2)`, que retorna los números de 0 a 10, pero de 2 en 2: `(0 2 4 6 8 10)`.



06- listas 1

Figura 8. Funciones para el procesamiento de listas

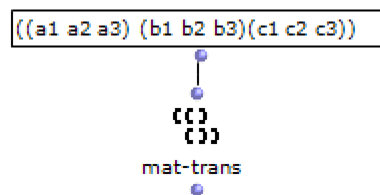
A una lista con sublistas en su interior, como `((a1 a2 a3)(b1 b2 b3)(c1 c2 c3))`, podemos considerarla una matriz, en la cual cada sublista representa una fila distinta. La función `mat-trans` transforma esa disposición, de modo tal que las filas se convierten en columnas, y las columnas en filas: `((a1 b1 c1)(a2 b2 c2)(a3 b3 c3))`.



a1	a2	a3
b1	b2	b3
c1	c2	c3

→

a1	b1	c1
a2	b2	c2
a3	b3	c3

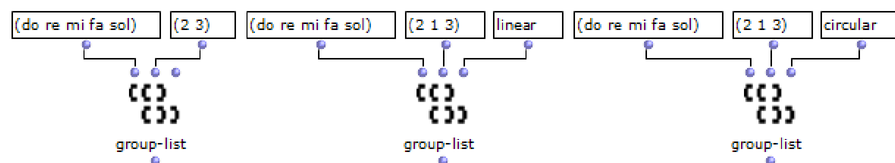


06- listas 1

Figura 9. Transposición de una matriz

`group-list` agrupa en sublistas los elementos de una lista de acuerdo a cantidad de elementos especificados. La lista `(do re mi fa sol)`, agrupada de acuerdo a `(2 3)` queda `((do re)(mi fa sol))`. Como tercer argumento es posible especificar

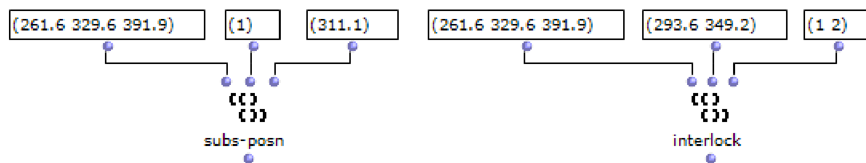
el modo en que se realiza la segmentación: lineal o circular. En el primer caso (lineal) la operación finaliza cuando concluye la lista dato, mientras que en el segundo caso, la lista es leída cíclicamente. A continuación, se aprecian en el gráfico los distintos comportamientos, que arrojan respectivamente `((do re) (mi) (fa sol))` y `((do re) (mi) (fa sol do))`.



06- listas 1

Figura 10. Agrupamiento de listas

La sustitución de los elementos de una lista puede llevarse a cabo mediante `subs-posn`. A la lista dato sigue un subíndice o una lista de subíndices que determinan la posición del elemento a reemplazar, comenzando por 0, y luego, los nuevos valores a ubicar en la lista. En el ejemplo que sigue reemplazamos la segunda frecuencia en Hertz de la lista (subíndice 1) por otra, convirtiendo al acorde mayor en menor. A la derecha observamos el uso de la función `interlock`, que inserta elementos delante de aquellos identificados por los subíndices indicados. Al acorde mayor le agregamos las frecuencias de las notas *re* y *fa*, formando una escala. En el primer caso, el resultado es `(261.6 311.1 391.9)`, y en el segundo `(261.6 293.6 329.6 349.2 391.9)`.

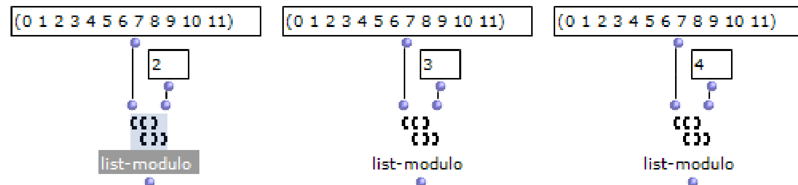


06- listas 1

Figura 11. Sustitución de elementos

La función `list-modulo`, por otra parte, segmenta una lista generando *n* sublistas en las cuales los elementos son elegidos de modo tal que sus posiciones se encuentran a una distancia *n* del primero, del segundo, y así siguiendo hasta llegar a *n*. Según se

ve en el ejemplo que sigue, obtenemos de la escala cromática ambas escalas por tonos. En este caso, *n* es igual a 2. Partiendo de 0 y tomando los elementos a un paso igual a 2 se genera la primera escala: `(0 2 4 6 8 10)`. Luego, partiendo de uno, obtenemos los grados impares en otra sublista: `(1 3 5 7 9 11)`.



06- listas 1

Figura 12. Módulo de los elementos de una lista

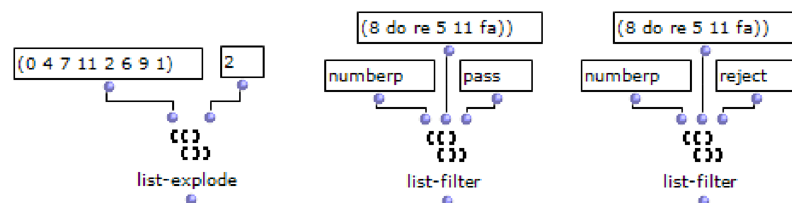
Del mismo modo, con *n* igual a 3 y a 4 obtenemos los acordes de séptima disminuida y las triadas aumentadas, respectivamente.

```
OM => ((0 3 6 9) (1 4 7 10) (2 5 8 11))
```

```
OM => ((0 4 8) (1 5 9) (2 6 10) (3 7 11))
```

De forma más simple, `list-explode` divide una lista dada en un número *n* de sublistas de igual cantidad de elementos, siempre y cuando esto sea posible. La lista `(0 4 7 11 2 6 9 1)`, por ejemplo, resulta separada en `((0 4 7 11) (2 6 9 1))`, dos acordes mayores con séptima mayor.

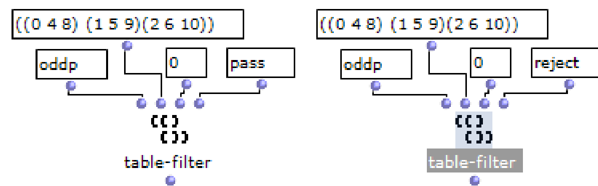
Con `list-filter`, en cambio, la lista dato es filtrada de acuerdo a un predicado, y al especificar si los elementos que cumplen la condición dada son admitidos (*pass*) o rechazados (*reject*). En el segundo ejemplo, de los que siguen, dejamos pasar solamente los números, mientras que en el tercero los rechazamos.



07- listas 2

Figura 13. Agrupamiento y filtrado de listas

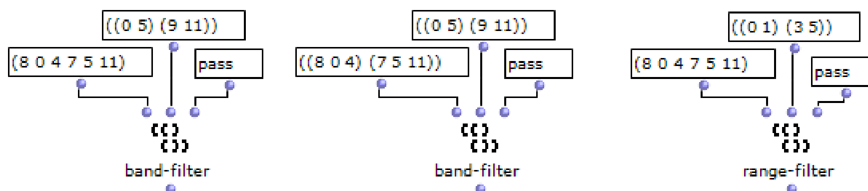
Otros filtros aplicados a los elementos de una lista son `table-filter`, `band-filter` y `range-filter`. El primero de ellos procesa una lista de listas, comparando a través de un predicado si un elemento de cada sublista, identificado a través de un subíndice dado, cumple o no la condición propuesta. En los ejemplos siguientes vemos tres acordes, representados por sublistas conteniendo grados cromáticos, filtrados o no de acuerdo a si el primer elemento de cada sublista (subíndice 0) es impar o no. El primer caso arroja `((1 5 9))`, mientras que el segundo, mediante *reject*, devuelve el resto de las sublistas con primer elemento par `((0 4 8) (2 6 10))`.



07- listas 2

Figura 14. Filtros de listas

`band-filter`, por otra parte, filtra a partir de límites, establecidos por un valor mínimo y otro máximo, aplicados recursivamente. Si cada elemento de la lista se encuentra comprendido entre estos límites, puede ser admitido o rechazado (con *pass* o *reject*). Si se establece más de un filtro de límites, se aplican en orden sobre todos los elementos de la lista. En el ejemplo de la izquierda observamos dos límites: (`(0 5) y (9 11)`). Significa que los elementos que se encuentren comprendidos entre cualquiera de ambos límites resulta un candidato. Dada la lista `(8 0 4 7 5 11)` el resultado obtenido es `(0 4 5 11)`. El 0, el 4 y el 5 se encuentran entre 0 y 5, y el 11 entre 9 y 11.



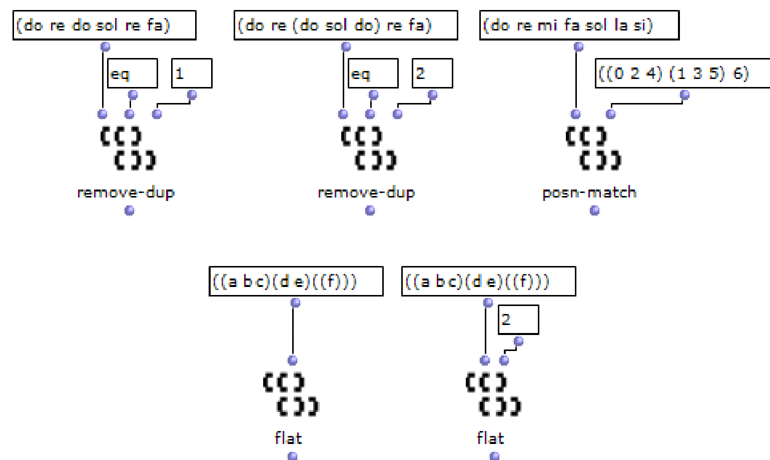
07- listas 2

Figura 15. Filtros de listas

El *patch* central también utiliza `band-filter`, pero en este caso aplicado a una lista dato con sublistas. El resultado es `((0 4) (5 11))`.

A la derecha vemos la aplicación de otro filtro, `range-filter`, cuyo funcionamiento es similar al anterior. Aquí no aplicamos límites numéricos sino subíndices mínimos y máximos, referidos a la posición de los elementos dentro de la lista. Cabe recordar que los subíndices se cuentan a partir de 0. Los elementos que pasan son el 8, el 0 (posiciones 0 y 1), el 7, el 5 y el 11 (posiciones 3 a 5), dando por resultado la lista `(8 0 7 5 11)`.

Finalmente, encontramos las funciones `remove-dup`, `posn-match` y `flat`. La primera elimina los elementos repetidos de una lista. El criterio utilizado por defecto es que esos elementos sean iguales (*eq*). La función puede aplicarse a listas dentro de listas, por lo cual es preciso establecer el nivel de las sublistas que se verán alcanzadas por este filtro. Un valor de 1 afecta únicamente a los elementos de la lista principal, un 2 sólo a los de sus sublistas, un 3 a las sublistas de esas sublistas, y así siguiendo.



07- listas 2

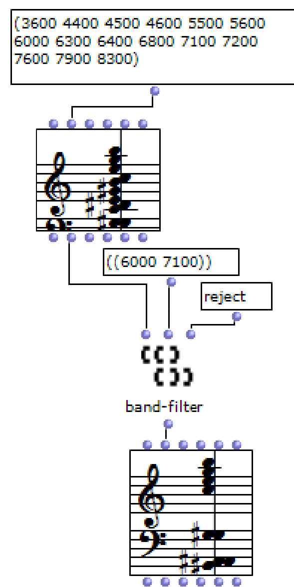
Figura 16. Filtros de listas

`posn-match` sirve para la construcción de listas a partir de una dada. El proceso se realiza especificando las posiciones de los elementos que formarán parte de la lista a crear. Si las posiciones se disponen en sublistas, los paréntesis empleados se trasladan a la lista resultado.

Por último, utilizamos `flat` para eliminar los los paréntesis internos de un árbol, o sea de una lista que posee sublistas. Si utilizamos el segundo *inlet* de la función

podremos especificar la profundidad a la que se llega durante el proceso de eliminación de paréntesis dentro del árbol.

Como ejemplo de aplicación queremos crear un filtro "rechazo de banda" aplicado a una sucesión de alturas expresadas en *midicents*. Para ello podemos recurrir a la función `band-filter`, en modo *reject*, y borrar las notas comprendidas entre un mínimo y un máximo. Si la banda a eliminar es la octava central, nos queda:



08 – listas 3

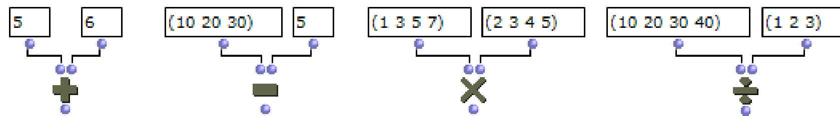
Figura 17. Filtro rechazo de banda

La secuencia original y la transformada pueden observarse abriendo el editor (con doble *click*) de cada objeto `chord`.

Figura 18. Secuencia original y filtrada con `band-filter`

Funciones aritméticas

Las funciones aritméticas propias de OM superan, en varios casos, a las primitivas de *LISP*. No sólo permiten operar con números, sino también con listas de números. Accedemos a estas funciones a través del menú *Functions/Basic Tools/Arithmetic*. Veamos unos ejemplos a través de las operaciones básicas.



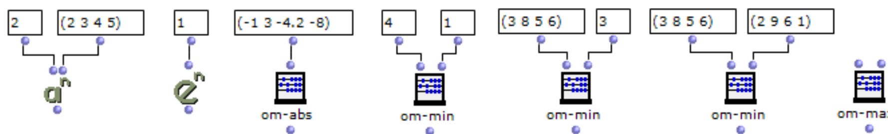
09- operaciones aritméticas

Figura 19. Operaciones básicas

En el *patch* de la izquierda sumamos dos números, lo cual es posible empleando la primitiva de *LISP*. Pero en la operación que sigue, restamos una lista de un número. El resultado es una nueva lista que se obtiene a partir de restarle a cada elemento de la lista el número dado. Al evaluar el objeto podremos visualizar el resultado.

En el caso de la multiplicación, operamos sobre dos listas. El primer elemento de una lista se multiplica por el primer elemento de la otra, y así hasta finalizar. Si una de las listas posee menor cantidad de elementos que la otra, el proceso finaliza al agotarse la lista más corta, como se aprecia en el ejemplo de la división.

Del mismo modo, operan las funciones `om^` (potenciación), `om-e` (número e = 2,7182818… elevado a la n), `om-abs` (valor absoluto), `om-min` (valor mínimo) y `om-max` (valor máximo).



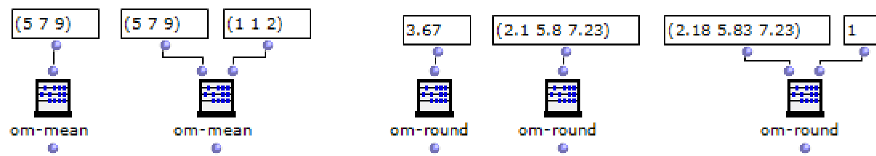
09- operaciones aritméticas

Figura 20. Otras operaciones

Si `om-min` recibe dos números devuelve el menor de ellos. Cuando se trata de un número y una lista, el número se coteja con cada elemento de la lista y el menor de ambos pasa a formar parte de la lista de resultados. Cuando se evalúan dos listas, la comparación se realiza entre el primero de una y el primero de la otra, el segundo con el segundo, y así sucesivamente. Lo mismo ocurre con `om-max`.

La función `mean` calcula el promedio de una lista de números, y también la media ponderada, en la cual a cada número le corresponde un valor de importancia distinto, denominado *peso*.

Si deseamos calcular el promedio de tres calificaciones (5, 7 y 9, por ejemplo) simplemente sumamos las notas y las dividimos por la cantidad de notas. En nuestro caso, 5 + 7 + 9 dividido 3, que da 7.



09- operaciones aritméticas

Figura 21. Promedio y redondeo

En el promedio ponderado, en cambio, podemos asignar a una nota (al 9, por ejemplo) un peso mayor que al resto. Si el peso del 9 es el doble que el de las demás notas lo multiplicamos por dos y luego sumamos ese resultado al 5 y al 7. Finalmente, dividimos por la suma de todos los pesos (1 para el 5, 1 para el 7 y 2 para el 9). La fórmula de la media ponderada es la siguiente, donde p es el peso y V el valor a promediar:

$$media\ ponderada = \frac{p_1V_1 + p_2V_2 + \cdots + p_nV_n}{p_1 + p_2 + \cdots + p_n}$$

En nuestro ejemplo,

$$media\ ponderada = \frac{1.5 + 1.7 + 2.9}{1 + 1 + 2} = 7{,}5$$

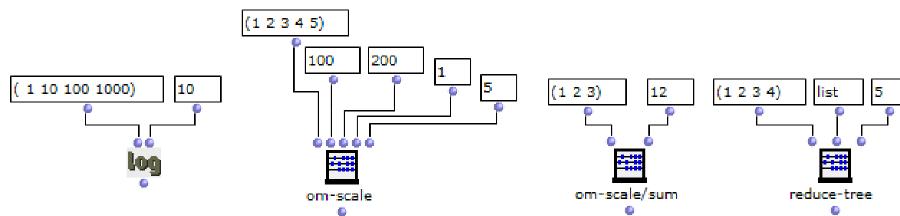
El redondeo de un número o una lista de números, se realiza mediante `om-round`. La función acepta tres parámetros. El primero es el número o la lista de números a

redondear, el segundo la cantidad de decimales (por defecto 0) y el tercero es un número que divide a la cifra ingresada en el primer *inlet*. En el primer ejemplo ingresamos el número 3.64 y obtenemos 4 como resultado. En el segundo, la lista (2.1 5.8 7.23), y obtenemos (2 6 7). Por último, la lista (2.18 5.83 7.23), para ser redondeada en un decimal (1), lo cal arroja (2.2 5.8 7.2).

Mediante log calculamos el logaritmo en base 10. Recordemos que el logaritmo del número *a* en base *b* es *c*, si *b* elevado a la *c* es igual a *a*:

$$log_b a = c \ \ si \ \ b^c = a$$

Al ser la base igual a 10, el logaritmo en base 10 de 100 es 2, ya que 10 al cuadrado es 100.



09- operaciones aritméticas

Figura 22. Otras funciones numéricas

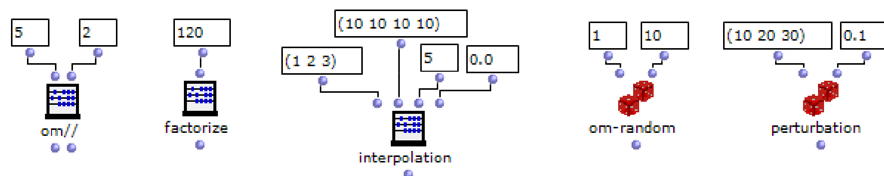
En relación con om-scale, la función escala un número o una lista de números a un rango dado. A partir del segundo *inlet* se especifican el valor mínimo de salida, el máximo de salida, el valor mínimo de entrada y el máximo de entrada. En el ejemplo, la lista (1 2 3 4 5) es escalada a través de los parámetros 100, 200, 1 y 5, lo cual significa que los números de entrada entre 1 y 5 son escalados proporcionalmente entre 100 y 200: (100 125 150 175 200).

Om-scale/sum, por otra parte, toma una lista y escala sus valores de modo tal que la suma de los mismos sea igual al número ingresado por el segundo *inlet*. Dada la lista (1 2 3), cuya suma es 6, y el argumento 12, el resultado de aplicar la función a esos datos es (2 4 6), cuya suma da el resultado esperado.

reduce-tree aplica una función binaria (de dos argumentos) de manera recursiva sobre los elementos de una lista, cuyo resultado se va acumulando elemento tras elemento. Es similar a la función reduce de *LISP*. En el tercer *inlet* puede incluirse un valor inicial. En el ejemplo utilizamos la función list, por lo cual la primera aplicación da (1 5). Luego se aplica al segundo elemento, el 2, quedando (2 (1

5)). Luego al tercer elemento, 3, lo que arroja (3 (2 (1 5))), y finalmente al último elemento que da por resultado final (4 (3 (2 (1 5)))).

La función om// admite números o listas en sus *inlets* y devuelve por un lado el resultado de la división entera, y por otro el resto de tal división. Podemos utilizarlo, por ejemplo, para extraer el grado cromático y el número de octava de una nota MIDI. Si los argumentos son 64 (nota MIDI) y 12, obtenemos 5 (octava central en la norma MID) y 4 (grado, que es la nota *mi*).



09- operaciones aritméticas

Figura 23. Otras funciones numéricas

factorize descompone un número en sus factores primos. El número 120 puede descomponerse como $2^3 * 3^1 * 5^1$, y el resultado se muestra ((2 3)(3 1)(5 1)). Los números primos son sólo divisibles por sí mismos y por la unidad, y resultan útiles, entre otras cosas, para evitar la coincidencia temporal de eventos en un período determinado, como en el caso de las polirritmias.

Para efectuar una interpolación entre números o listas de números recurrimos a la función interpolation. En el primer *inlet* especificamos el número o lista de partida, y en el segundo el número o lista de llegada. El entero ingresado en el tercer *inlet* determina la cantidad de pasos a obtener entre el inicio y el fin de la interpolación. En el cuarto *inlet* se especifica el modo o tipo de curva que se emplea para llevarla a cabo; un 0 significa que la interpolación es lineal, otros valores que es exponencial. En el ejemplo interpolamos dos listas. La de partida es (1 2 3) y la de llegada (10 10 10). El resultado de 5 pasos obtenidos de forma lineal da por resultado:

```
OM => ((1.0 2.0 3.0) (3.25 4.0 4.75) (5.5 6.0 6.5) (7.75 8.0
8.25) (10.0 10.0 10.0))
```

Si observamos cómo evoluciona el primer elemento de la lista (el número 1) notamos que se incrementa linealmente sumando 2,25 a cada paso, hasta llegar a 10. En el caso del número 3, que se encuentra más cerca del 10, sumando 1,75 en cada paso.

Podemos aplicar el concepto de interpolación a determinadas estructuras musicales, a fin de lograr estados intermedios entre una de origen y otra de llegada. Por ejemplo, entre dos acordes, hallando un número de acordes intermedios entre ellos, lineal o exponencialmente.

Si el número de partida es *ni*, el de llegada es *nf*, la cantidad de pasos es *p*, y el número que define el tipo de curva es *c*, cada término de la interpolación es calculado de la siguiente forma:

$$n = ni + (nf - ni).p^{e^c}$$

donde *e* es el número de Euler (2,7182818…). Si *c* = 0, *e* elevado a la 0 da 1, por lo cual la interpolación es lineal. Vemos en la ecuación que al número de partida (*ni*) se le suman porciones de la diferencia entre el número final y el inicial.

A través de la función `om-random` podemos obtener números o listas de números generados aleatoriamente dentro de un rango especificado. `perturbation`, por otra parte, recibe un número o una lista por el *inlet* izquierdo y aplica una desviación aleatoria, dada por el número o la lista de números conectada al *inlet* derecho. El valor de desviación aleatoria, denominado *percent*, puede variar entre 0 y 1, y el resultado se calcula en base a:

```
(* n (+ 1(om-random (- percent)(float percent))))
```

Si *percent* vale 1 y el número a desviar es 10, nos queda

```
(* 10 (+ 1(om-random (- 1)(1.0))))
```

O sea un valor aleatorio entre 0 y 20. Si en cambio es 0.1, arroja un valor al azar entre 0.9 y 1.1.

Dada la lista `(10 20 30)` con un factor de desviación de 0.1 podríamos obtener aleatoriamente:

```
OM => (10.367278 21.712477 29.556574)
```

Como aplicación nos proponemos ahora generar una secuencia de alturas en la cual los intervalos crezcan exponencialmente, al igual que las duraciones de los sonidos.

En primer lugar creamos una sucesión de números naturales entre 1 y 15, empleando `expand-lst`, con argumento `(0_15)`. A cada uno de esos números los vamos a elevar a una potencia determinada, en nuestro ejemplo, a la 1.62; los multiplicaremos por 100 y luego redondearemos al entero más cercano. Así quedarán definidos

nuestros intervalos en *midicents*. La superposición de los intervalos sobre una nota dada (la nota MIDI 3000) nos permitirá construir la secuencia melódica.

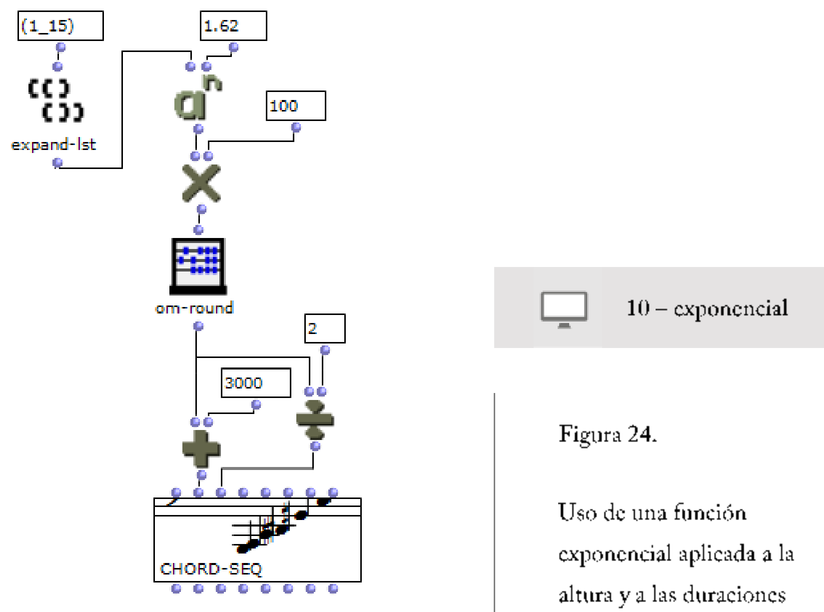


Figura 24.

Uso de una función exponencial aplicada a la altura y a las duraciones

Nótese que para representar los resultados no utilizamos la clase `chord`, sino `chord-seq`, dado que esta última nos va a permitir generar una secuencia de notas con tiempos de ataque diferentes. A fin de establecer los momentos de ataque, utilizamos la misma sucesión exponencial, pero dividimos sus valores por dos para que la duración total baje a la mitad. El tercer *inlet* es el que recibe la información temporal, medida en milisegundos.

Como era de esperar, si observamos las notas obtenidas veremos que sus afinaciones en cents no se corresponden con las del sistema temperado, lo que significa que nuestra secuencia es microtonal. Desde el editor de `chord-seq` seleccionamos 1/4 en la opción *Approx*, lo que significa que cada altura la veremos escrita y la escucharemos redondeada al cuarto de tono más cercano. El resultado final es:

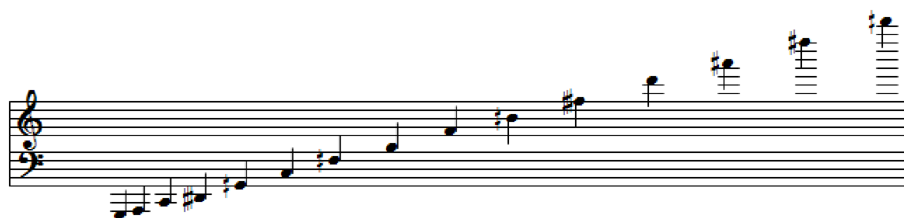
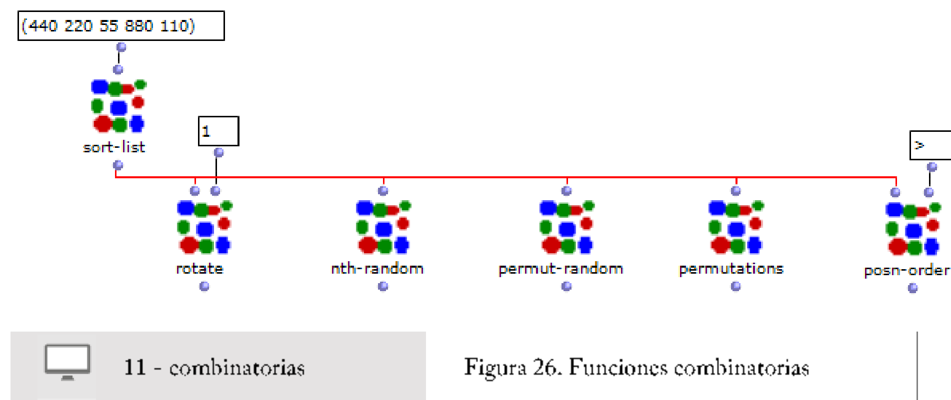


Figura 25. Secuencia con intervalos y duraciones exponenciales

Funciones combinatorias

En el menú *Funtions/Basic Tools/Combinatorial* encontramos algunas funciones que operan sobre listas, y que transforman la ubicación de sus elementos.

La función `sort-list` ordena de menor a mayor los elementos numéricos de la lista aportada como dato. En el ejemplo que sigue, distintas frecuencias de la nota *la*. Evaluando el objeto podremos apreciar el resultado. Una vez ordenada la lista, aplicamos otras funciones. La primera de ellas es `rotate`, que realiza una permutación circular[29] de los elementos, a partir del elemento cuya posición se especifica.



11 - combinatorias

Figura 26. Funciones combinatorias

Con `nth-random` obtenemos un elemento de la lista al azar, mientras que con `permut-random` generamos una permutación aleatoria.

La función `permutations`, por otra parte, arroja todas las permutaciones posibles[30] de los elementos de la lista tomada como dato.

---

[29] Una permutación es, en general, cualquier ordenamiento que hagamos de los elementos de un conjunto. Una permutación circular, en cambio, es un tipo de ordenamiento particular donde el primer elemento pasa al último lugar. Dada la lista (*a, b, c, d*), su primera permutación es (*b, c, d, a*), la segunda (*c, d, a, b*) y así siguiendo, con el mismo criterio.

[30] La cantidad de permutaciones de una lista de longitud *n* (donde *n* es la cantidad de elementos) es *n*!, que se lee "*n* factorial". El factorial de un número se resuelve multiplicando al número por *n*-1, *n*-2, *n*-3, etc., hasta llegar a 1. Factorial de 5, por ejemplo es 5.4.3.2.1 = 120, lo cual significa que cinco elementos pueden ordenarse de 120 maneras diferentes.

Finalmente, `posn-order` ordena los subíndices de una lista de acuerdo a la función especificada. Si la función es >, como en el ejemplo, ordena de mayor a menor, resultando la lista `(4 3 2 1 0)`.

Para el ejemplo siguiente partimos de una yuxtaposición de acordes tríadas, dispuestos en una clase `chord-seq`. La cantidad de permutaciones de esos acordes podríamos obtenerla calculando la longitud de toda la lista de permutaciones posibles, como se observa en la figura. Si bien el método no es el más eficiente, podríamos ver todos los ordenamientos al evaluar `permutations`.

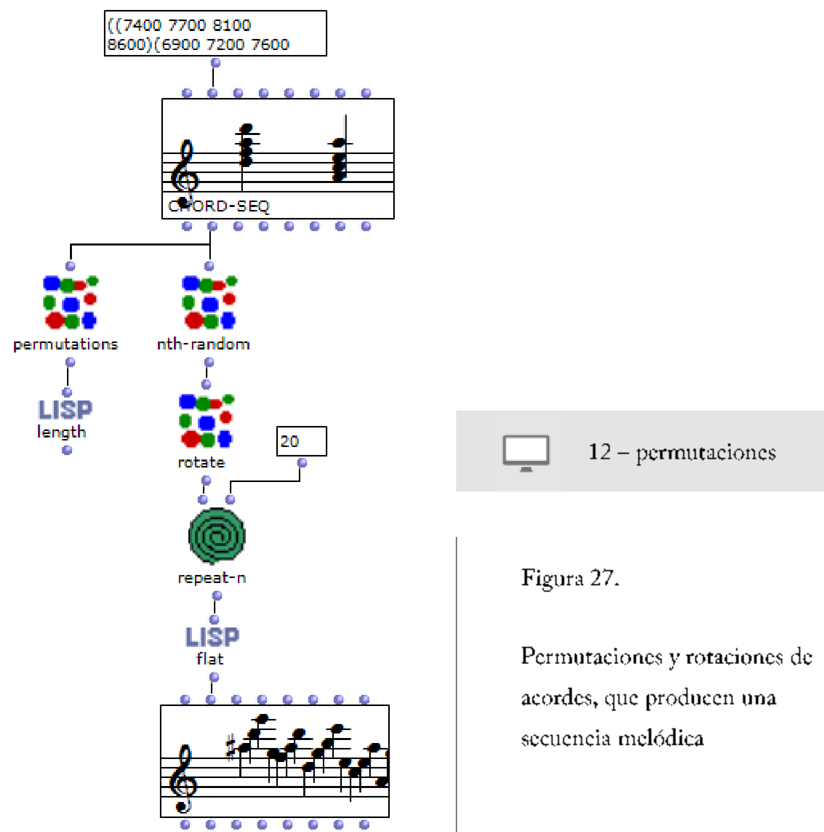


Figura 27.

Permutaciones y rotaciones de acordes, que producen una secuencia melódica

Cada acorde se ubica dentro de una sublista, pues así lo espera `chord-seq`. Con `nth-random` elegimos un acorde al azar, y luego rotamos las notas, en relación con el orden en que fueron ingresadas en la lista. La función `repeat-n` (que veremos más en detalle al tratar las iteraciones) solicita 20 evaluaciones de lo anterior y almacena los resultados en una nueva lista. Finalmente con `flat` eliminamos los paréntesis internos, e ingresamos la información en la clase `chord-seq`, que interpreta a esa lista como una sucesión melódica.

Series numéricas

Las series numéricas son conjuntos ordenados de números. A continuación analizaremos algunos casos, dado que OM posee algunas funciones destinadas a su generación y tratamiento. Las mismas se encuentran en *Functions/Basic Tools/Series*.

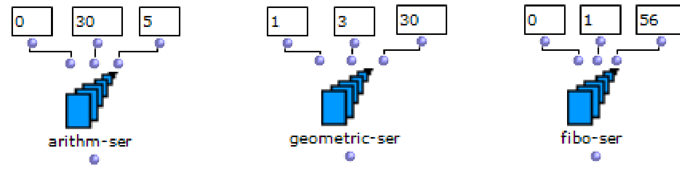
Una serie aritmética parte de un número dado y cada término siguiente se obtiene sumando una constante al término anterior. A esa constante la denominamos razón. Si partimos de 1, por ejemplo y fijamos como razón al número 2, obtenemos la serie de números impares: 1, 1 + 2 = 3, 3 + 2 = 5, 5 + 2 = 7, 7 + 2 = 9, etc.

En una serie geométrica, en cambio, la razón es multiplicada en lugar de ser sumada. La serie 1, 2, 4, 8, 16, etc. es una serie geométrica cuya razón también es 2.

El motivo por el cual las series aritméticas y geométricas tienen importancia en la música se basa en la relación que existe entre los estímulos y las sensaciones sonoras. Es bien sabido que cuando una frecuencia se duplica, percibimos un intervalo de octava. La frecuencia es una magnitud física que forma parte del estímulo, mientras que la altura es una sensación. Si partimos de 100 Hz, por ejemplo, y construimos una serie geométrica con razón igual a 2, obtenemos: 100 Hz, 200 Hz, 400 Hz, 800 Hz, etc. Mientras las frecuencias se van duplicando, nuestro sistema perceptual experimenta el ascenso a través de escalones iguales de altura: 1 octava, 2 octavas, 3 octavas, etc. Vemos, entonces, que si los estímulos se suceden en progresión geométrica, las sensaciones lo hacen en progresión aritmética[31].

La función `arithm-ser` genera una serie aritmética. En el primer *inlet* se establece el número de partida, en el segundo un número que sirve de límite superior máximo (para que la serie no sea infinita) y en el tercero la razón. La función `geometric-ser`, que genera series geométricas, tiene sus *inlets* en un orden distinto al de `arithm-ser`. El primero es el número de partida, el segundo la razón y el tercero el límite máximo. En los ejemplos siguientes vemos su empleo. La serie aritmética lograda es `(0 5 10 15 20 25 30)` y la geométrica `(1 3 9 27)`.

[31] Este principio es conocido como Ley de Weber-Fechner.



13 - series

Figura 28. Series numéricas

Otra relación frecuentemente utilizada es la proporción áurea. Dados dos segmentos, *a* y *b*, decimos que se encuentran en proporción áurea si se cumple:

a b
a+b

$$\Phi = \frac{a+b}{a} = \frac{a}{b}$$

La ecuación posee dos soluciones, una positiva y otra negativa,

$$\frac{1+\sqrt{5}}{2} \quad \text{y} \quad \frac{1-\sqrt{5}}{2}$$

tomándose la positiva como el número áureo Φ (*phi* mayúscula).

A la derecha del gráfico anterior vemos la función `fibo-ser`, que calcula los términos de la sucesión de Fibonacci[32]. Tal sucesión también ha sido ampliamente utilizada, dado que se relaciona con la proporción áurea. A medida que avanzamos sobre sus términos, el cociente entre dos términos consecutivos se aproxima más y más al número de oro (1.6180339887…).

La sucesión de Fibonacci se construye a partir de dos números de partida, en general 0 y 1, y cada término siguiente resulta de la suma de los dos anteriores.

$$f_0 = 0, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

El cálculo de cualquier término de la serie puede obtenerse empleando la fórmula:

$$f_n = \frac{\Phi^n - (-\Phi)^{-n}}{\sqrt{5}}$$

que muestra claramente la relación existente entre la sucesión y el número de oro.

---

[32] Leonardo Bigollo o Leonardo de Pisa (c. 1170-c. 1250), más conocido como Fibonacci, fue quien introdujo la sucesión en Europa, luego de estudiar con matemáticos árabes.

En el *patch* de la figura 28, los puntos de partida son 0 y 1, y el límite máximo se enncuentra en 56, resultando `(0 1 1 2 3 5 8 13 21 34 55)`.

Si pensamos que los números representan grados cromáticos, los primeros términos definen una escala, diseñada por Béla Bartók.

do – do# - re – mib - fa - lab

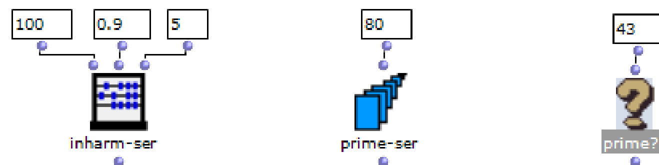
Del mismo modo, podemos continuar generando la sucesión de grados cromáticos, con los términos superiores, aplicando módulo 12.

En los ejemplos que siguen utilizamos la función `inharm-ser`, que sirve para crear series de armónicos a los cuales se les puede aplicar un índice de distorsión para convertirlos en parciales. Partiendo de 100 Hz proponemos a la función la generación de 5 componentes con un valor de distorsión (*dist*) igual a 0.9, lo que da por resultado:

```
OM => (100.0 186.6066 268.78754 348.2202 425.66995)
```

La desviación aplicada a un parcial se calcula en base a:

$$f_{parcial} = f_{inicial} * nrode\ parcial^{dist}$$



13 - series

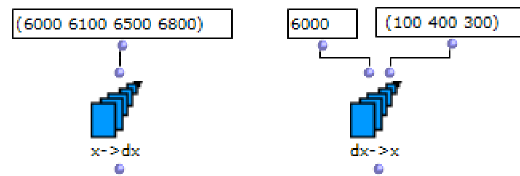
Figura 29. Series inarmónicas y de números primos

Por otra parte, la función `prime-ser` calcula una sucesión de números primos entre 0 y un valor máximo especificado en su *inlet*. Recordemos que los números primos son aquellos divisibles únicamente por sí mismos y por la unidad. Un segundo *inlet* permite especificar la cantidad máxima de términos. Si el argumento es 80, los números primos calculados son:

```
OM => (1 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73
79)
```

A esta función la acompaña el predicado `prime?`, que determina si un número ingresado por su *inlet* es primo o no.

Cerrando esta colección de funciones destinadas a series numéricas, encontramos los objetos `x->dx` y `dx->x`. El primero de ellos extrae las diferencias entre términos consecutivos. Si consideramos que los números que ingresan a la función representan determinadas notas expresadas en *midicents*, las diferencias equivalen a los intervalos entre esas notas, medidos en cents. En el siguiente ejemplo vemos que las notas 6000 (*do*), 6100 (*do#*), 6500 (*fa*) y 6800 (*sol#*), dispuestas melódicamente, contienen los intervalos 100 (segunda menor), 400 (tercera mayor) y 300 (tercera menor), devueltos por la función.

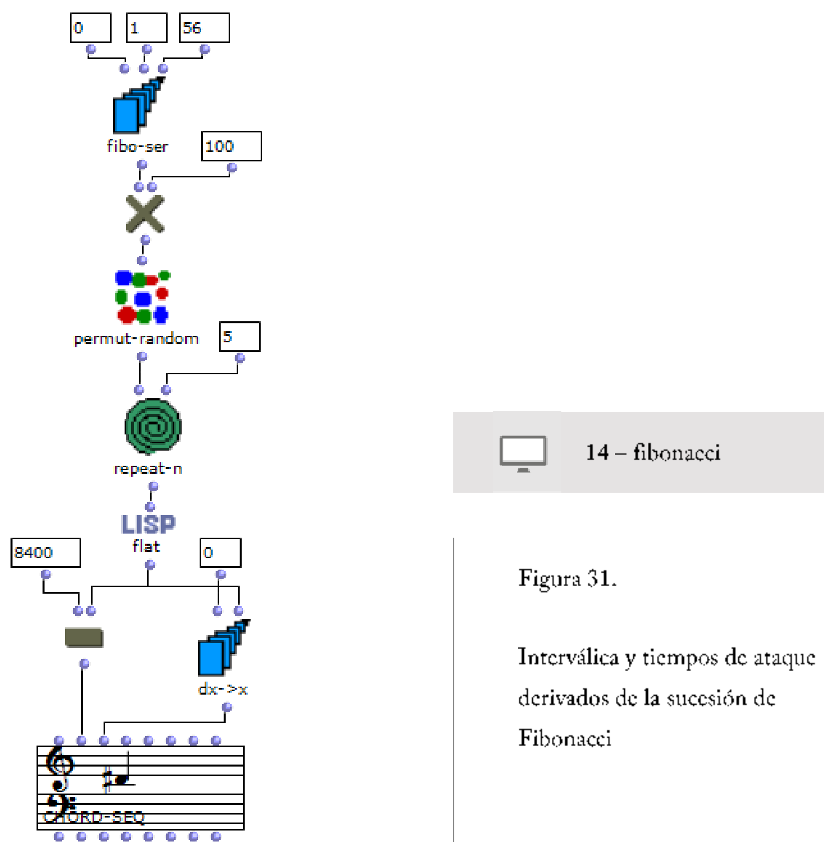


13 - series

Figura 30. Funciones de diferencia y acumulación

De forma inversa, si indicamos la nota de comienzo y los intervalos a `dx->x`, podremos reconstruir la secuencia original de notas `(6000 6100 6500 6800)`, teniendo en cuenta que la función calcula los valores de las notas sumando a la nota de partida el primer intervalo, a la que resulta el segundo intervalo y así hasta completar la totalidad de la secuencia.

Para la realización del programa que sigue utilizamos la sucesión de Fibonacci para crear tanto las alturas como los tiempos de ataque de los eventos sonoros. Los once términos obtenidos de la sucesión se multiplican por 100 para convertirse en intervalos de altura (en cents) y duraciones (en milisegundos). Esos números son permutados al azar, y el procedimiento se repite cinco veces, arrojando un total de 55 valores (11 por 5). Los intervalos de altura se restan de la nota MIDI tomada como referencia (8400), y las duraciones se van acumulando a partir de 0 para convertirse en tiempos de ataque. Finalmente, la clase `chord-seq` es instanciada al recibir la evaluación. El resultado se percibe como una registración fija en la cual los eventos sonoros ocurren sin un ritmo definido, debido a las proporciones de los términos de la sucesión.



14 – fibonacci

Figura 31.

Interválica y tiempos de ataque derivados de la sucesión de Fibonacci

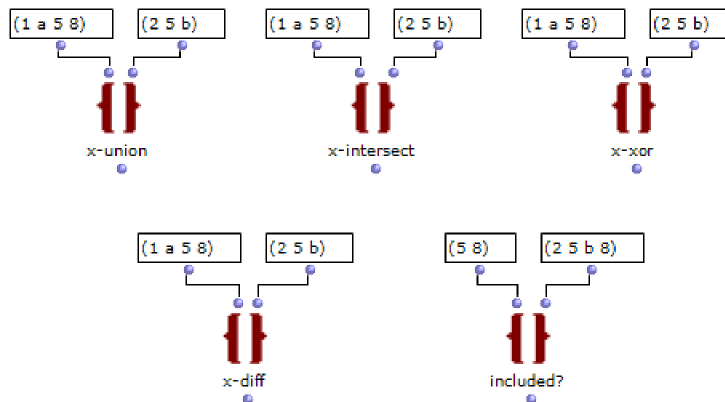
A continuación se muestra uno de los resultados posibles, dado que éstos dependen del azar.

Figura 32. Secuencia con intervalos y duraciones según Fibonacci

Conjuntos

El menú *Functions/Basic Tools/ Sets* contiene cinco funciones aplicables a conjuntos de elementos. Se trata de `x-union`, `x-intersect`, `x-or`, `x-diff` e `included?`

- `x-union`: combina dos o más listas formando una sola, sin repeticiones.
- `x-intersect`: devuelve la intersección (elementos comunes) de dos o más listas.
- `x-or`: devuelve los elementos que aparecen en una lista pero no en la(s) otra(s).
- `x-diff`: devuelve los elementos presentes en la primera lista que no se encuentran en las restantes.
- `included?`: verifica si la primera lista se encuentra incluida en la segunda.



15 - conjuntos

Figura 33. Funciones de diferencia y acumulación

Como ejercicio nos proponemos crear una secuencia de alturas y filtrar luego determinada intervalo. Si deseamos eliminar de la secuencia las clases interválicas de tercera mayor, debemos descartar los intervalos de 400 cents (3M ascendente), -400 cents (3M descendente), 800 cents(6m ascendente) y – 800 cents (6m descendente).

Para ello vamos a utilizar `x-or`, que devuelve los elementos de una lista que no se encuentran en la otra.

La conversión de notas MIDI en *midicents* a una sucesión de intervalos la realizamos con `x->dx`. Una vez quitados los intervalos no deseados reconstruimos la secuencia con `dx->x`.

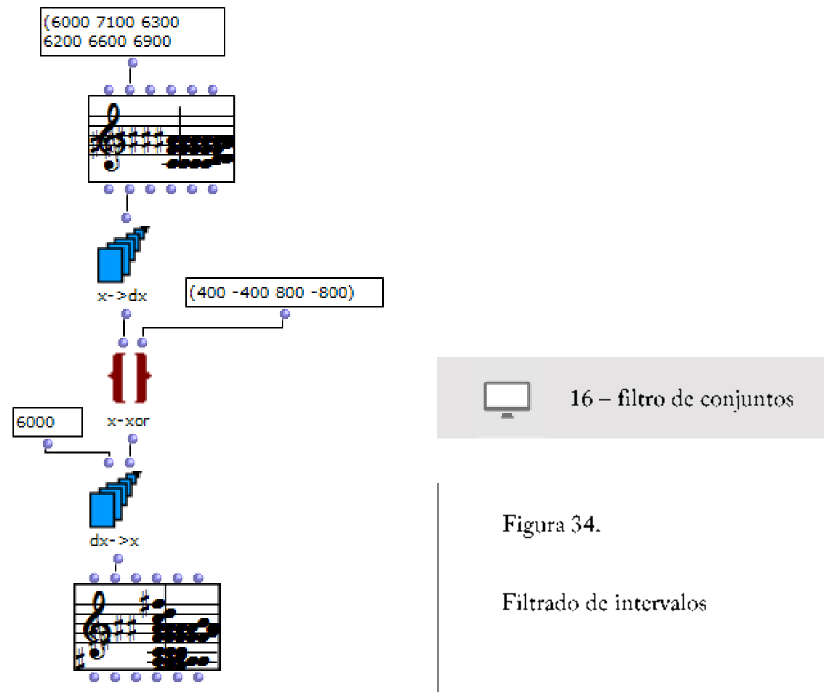


Figura 34.

Filtrado de intervalos

A continuación se muestran los resultados.

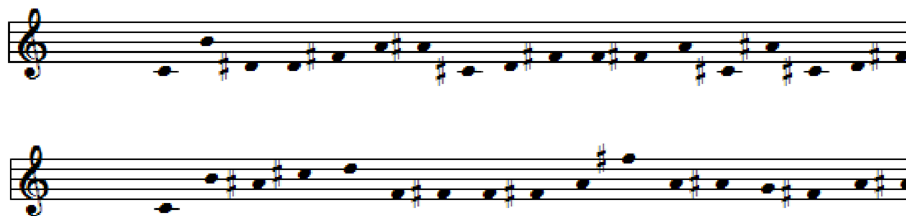


Figura 35. Parte de la secuencia original y de la secuencia filtrada

Curvas

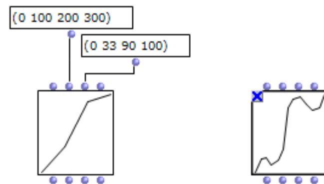
Solemos utilizar curvas o gráficos para representar la variación de determinados parámetros musicales en el tiempo, trazar trayectorias espaciales, o establecer relaciones entre conjuntos de datos, entre otras tantas aplicaciones.

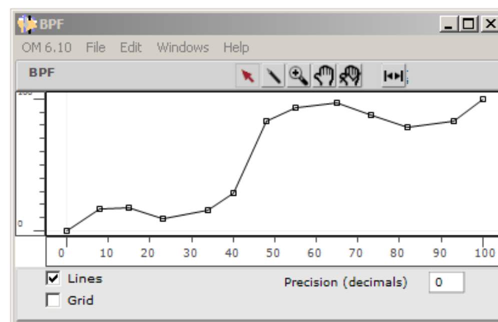
OM posee clases y funciones destinadas a la construcción de curvas de distintas características, que resultan de gran ayuda al momento de definir el comportamiento de las estructuras musicales, o establecer los datos requeridos por los procedimientos ideados.

Curvas en dos dimensiones

Una de las clases, destinada a la representación gráfica en dos dimensiones de funciones, es `bpf` (*break-points function*). Como ocurre con las funciones matemáticas, a cada valor de entrada corresponde un único valor posible de salida, o dicho de otro modo, a cada valor sobre el eje *x* corresponde un único valor sobre el eje *y*.

En las instancias de la clase `bpf` definimos cada punto aportando las coordenadas a través de listas de números, o bien gráficamente, dibujando sobre el editor asociado a la clase.





17 – curvas 1

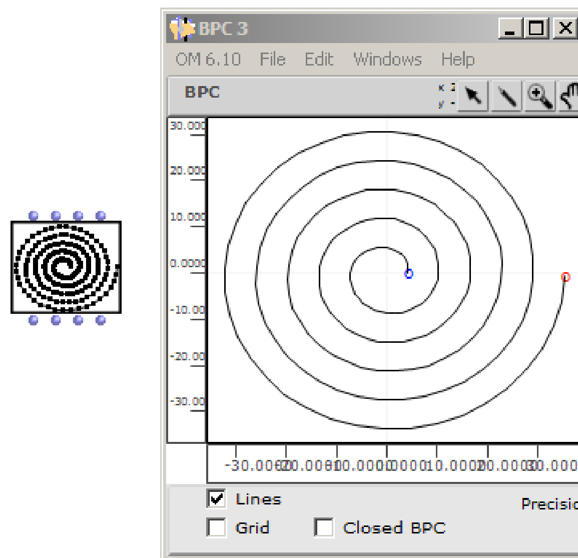
Figura 36. Clase `bpf`, para graficar funciones

En el segundo *inlet* especificamos la lista de coordenadas *x*, mientras que en el tercero establecemos los valores de *y*. El cuarto *inlet* sirve para definir la precisión en

la ubicación de los puntos, a través de la cantidad de decimales a considerar. Un valor 0 significa que las coordenadas se expresan con números enteros. Si las listas de coordenadas aportadas son de diferente longitud, la más corta repite su último valor hasta alcanzar la cantidad de elementos de la lista más larga.

Si dibujamos sobre el editor, creamos nuevos puntos presionando *Control* + *click*. Para visualizar todos los puntos de la curva hacemos *click* sobre su contorno, lo cual nos permite identificarlos, y así moverlos o borrarlos según deseemos. Finalmente, obtenemos las listas de coordenadas de la curva dibujada evaluando los *outlets* 2 y 3.

Otra clase, con un comportamiento similar a `bpf`, es `bpc` (*break-points curve*). Pero en este caso, dirigida a todo tipo de curvas, no solamente aquellas que representan a una función, como es el caso de las curvas cerradas o las espirales.

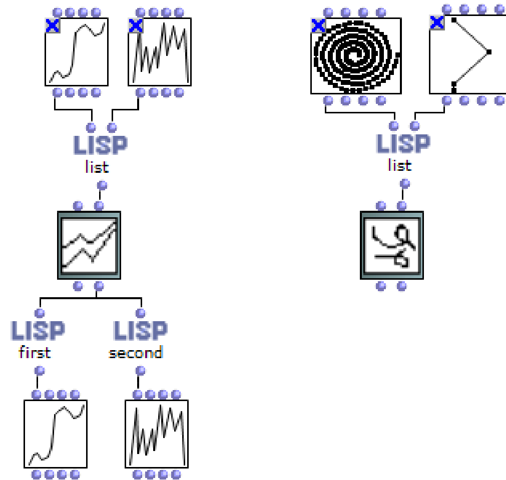


17 – curvas 1

Figura 37. Clase `bpc`, para graficar curvas

Dos o más curvas pueden ser almacenadas en un tipo de clase que las contiene. En el caso de las funciones, la clase se denomina `bpf-lib`. Para el ejemplo siguiente, generamos dos funciones con la clase `bpf` y enlistamos sus instancias con `list`. Esa lista es conectada al segundo *inlet* de `bpf-lib`, y se almacena en el objeto luego de la evaluación correspondiente. Posteriormente, recuperamos las funciones individuales extrayendo los objetos de la lista de salida.

Las curvas, por otra parte, se almacenan en la clase `bpc-lib`.



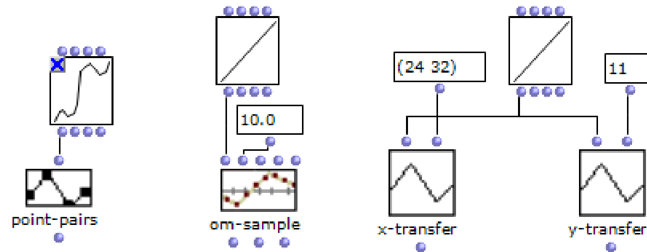
17 – curvas 1

Figura 38. Clases `bpf-lib` y `bpc-lib`

OM dispone de diversas funciones relacionadas con las clases `bpf` y `bpc`. La función `point-pairs`, por ejemplo, conectada a cualquiera de ambas clases devuelve la lista de coordenadas, dispuestas en sublistas con la forma (*x y*).

La función `om-sample` muestrea una curva. El muestreo se realiza en función de la cantidad de muestras a tomar, si se especifica un número entero; o como frecuencia de muestreo, si el número posee decimales. En este último caso, el valor ingresado representa el paso entre dos muestras consecutivas, partiendo del valor mínimo hasta llegar al máximo del eje *x*. Un número igual a 1.0 avanza de uno en uno, mientras que el 2.0 lo hace de a dos. El rango de lectura del eje *x* se puede acotar mediante los argumentos opcionales *xmin* y *xmax* (tercer y cuarto *inlets*), al igual que la cantidad de decimales de las coordenadas, a través del quinto *inlet*.

En el ejemplo de la página que sigue empleamos la clase `bpf`, la cual trae por defecto una recta determinada solamente por dos muestras, cuyas coordenadas son (0, 0) y (100, 100). Si empleamos `om-sample` con una frecuencia de muestreo de 10.0 obtenemos, tanto para el eje *x* como para el eje *y*, los siguientes valores: `(0.0 10.0 20.0 30.0 40.0 50.0 60.0 70.0 80.0 90.0 100.0)`.

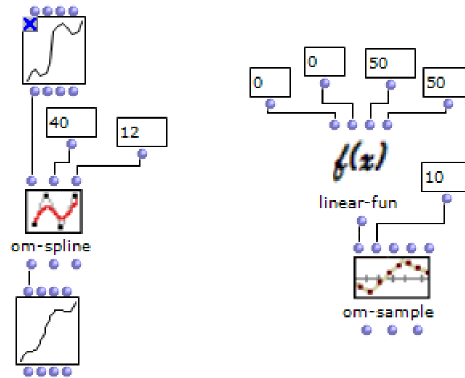


18 – curvas 2

Figura 39. Funciones de las clases `bpf` y `bpc`

Cuando deseamos obtener la coordenada *x* de un punto cualquiera de una curva alojada en un `bpf`, que no coincide con ninguna muestra existente, podemos recurrir a `x-transfer`. Al especificar un valor cualquiera de *y*, la función devuelve el valor interpolado de *x* correspondiente. La cantidad de decimales puede indicarse en el tercer *inlet*, que es opcional. De modo inverso, para un valor de *x* cualquiera podemos obtener el valor de *y* que le corresponde, utilizando `y-transfer`. En el caso de `x-transfer` pueden explicitarse varios valores de *x*, pero siempre dentro de una lista.

Una curva generada con `bpf` o `bpc`, que tenga pocos *breakpoints*, puede verse un tanto quebrada por la falta de resolución. Para evitar este problema es posible recurrir a la función `om-spline`, que devuelve una curva suavizada por medio de una interpolación polinómica. Mientras mayor es el grado del polinomio a emplear, más suavizada resulta la curva. El grado se especifica en el tercer *inlet* (por defecto es igual a 3), mientras que la resolución, medida en cantidad de muestras a generar, se ingresa a través del *inlet* dos. Como es usual en este tipo de funciones, el primer *inlet* (*self*) recibe un objeto, en este caso del tipo `bpf` o `bpc`. Nótese en el ejemplo siguiente que la curva a suavizar se encuentra bloqueada. Esto se logra seleccionando la caja y presionando la tecla *b*. De otro modo, al evaluar el resultado se genera una nueva instancia de la clase, que borra el contenido de la curva y muestra la recta que por defecto aparece en la clase `bpf`.

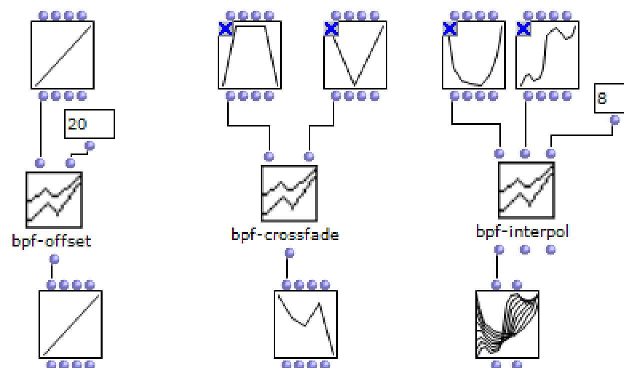


18 – curvas 2

Figura 40. Funciones `om-spline` y `linear-fun`

A la derecha del gráfico vemos la función `linear-fun`, que crea un segmento de recta que parte de las coordenadas (x_0, y_0) y finaliza en (x_1, y_1), o sea, una función lineal. Para obtener las muestras intermedias recurrimos a `om-sample`.

Cuando deseamos desplazar sobre el eje *x* una función representada en una clase `bpf` utilizamos `bpf-offset`. En el ejemplo de abajo, al segmento de recta creado por defecto en el `bpf` le efectuamos un corrimiento de 20 unidades. Al editar el `bpf` inferior podremos apreciar el resultado.



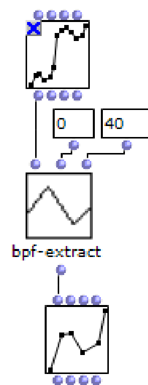
18 – curvas 2

Figura 41. Otras funciones relacionadas con curvas.

`bpf-crossfade` produce una curva que resulta de la interpolación dentro del área de superposición de dos curvas almacenadas en `bpf`, mientras que `bpf-interpol` genera una o varias curvas por interpolación entre las dos curvas dadas. La cantidad

de curvas que debe generar `bpf-interpol` se especifica en el tercer *inlet*. La función admite como argumentos opcionales el factor de interpolación, que por defecto es lineal, con un valor igual a 0; la cantidad de decimales y el modo, que sólo puede asumir alguna de estas dos opciones: *points* o *sample*. Si las curvas tienen diferente cantidad de muestras, y elegimos *points*, la resultante toma la cantidad menor y la curva dato con cantidad sobrante es truncada. Si el modo, en cambio, es *sample* las curvas son nuevamente muestreadas con igual cantidad de puntos antes de efectuarse la interpolación. En nuestro ejemplo, al haber generado varias curvas, el resultado es recogido con la clase `bpf-lib`.

La función `bpf-extract` extrae de un `bpf` una parte de la curva que allí se encuentra. Precisamente aquella comprendida entre las coordenadas x_1 y x_2, especificadas en los *inlets* 2 y 3.



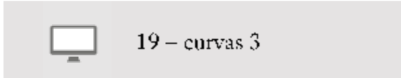


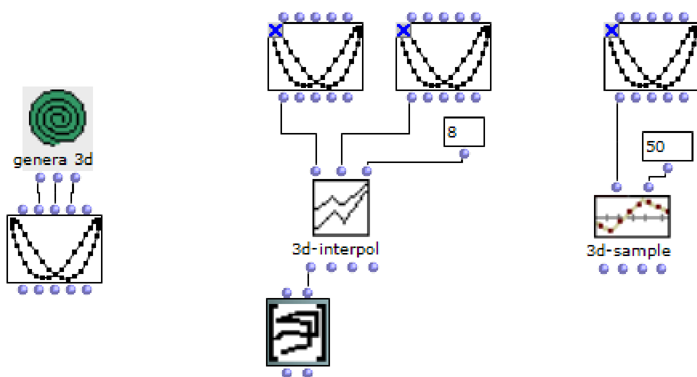
Figura 42.

Función `bpf-extract`

Curvas en tres dimensiones

La clase `3d` permite representar curvas en tres dimensiones, observables desde su propio editor. A través del mismo podemos rotar la curva con el mouse, y realizar un *zoom* presionando *Shift* + *click* y arrastrando el puntero.

En el ejemplo siguiente, a la izquierda, vemos la clase `3d`, y en el medio a la clase `3d-lib`, que recibe una familia de curvas que surgen de la interpolación entre dos curvas dadas, lograda mediante `3d-interpol`.

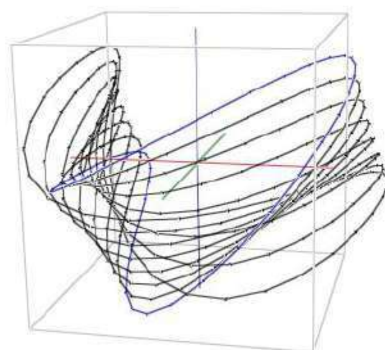


19 – curvas 3

Figura 43. Curvas en 3D.

Al hacer doble *click* sobre la caja de `3d-lib` accedemos al editor, donde se muestran las curvas resultantes del proceso de interpolación.

A la derecha de la figura 43, se aprecia la función `3d-sample`, cuyo funcionamiento es similar al de la versión en dos dimensiones.



19 – curvas 3

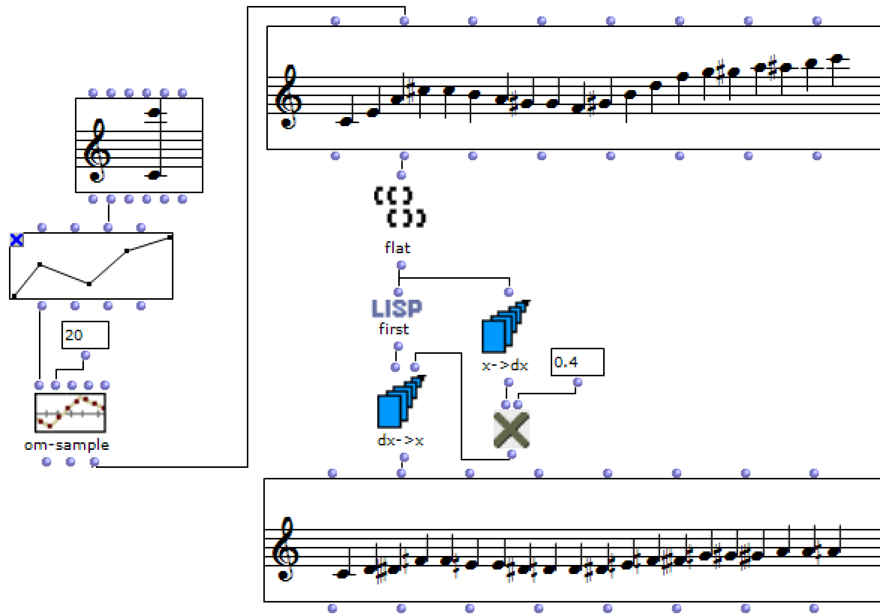
Figura 44.

Editor de `3d-lib`

Como ejemplo de aplicación vamos a generar una sucesión de alturas cuyo contorno melódico se establezca por medios gráficos, y posteriormente realizaremos una compresión melódica microtonal.

Para su realización, ingresamos en el tercer *inlet* de `bpf` una lista de dos números que determinan el rango de la función. Esos números son notas que extraemos de un objeto `chord`, y que fijan el ámbito en el que se va a desenvolver la melodía. Luego, muestreamos la función con `om-sample`, obteniendo 20 muestras de acuerdo a lo solicitado.

Para representar el resultado utilizamos la clase `chord-seq`. En el tercer *inlet* escribimos `(0 300)`, lo cual indica que los ataques se van a producir cada 300 milisegundos (escrito así, el intervalo temporal se proyecta para todas las notas), y en el cuarto *inlet* establecemos una duración de igual valor.



20 – curvas 4

Figura 45. Contorno y compresión melódica

El segundo *outlet* de `chord-seq` devuelve las notas MIDI en *midicents*, pero a diferencia de `chord`, éstas se encuentra en sublistas, lo cual puede verse evaluando ese *inlet* con *Control* + *click*. Si bien nosotros creamos solamente notas en orden sucesivo, la clase puede representar una cadena de acordes, separados unos de otros mediante sublistas. Eso explica el porqué de los paréntesis internos agregados por la clase, que eliminamos con `flat`.

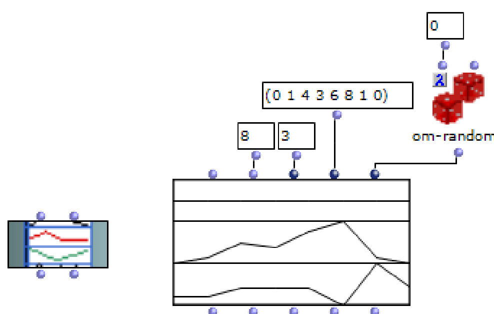
Posteriormente, realizamos la compresión melódica. Con `x->dx` calculamos los intervalos melódicos, y los multiplicamos por un número para agrandarlos o achicarlos. Una vez logrado esto, reconstruimos la melodía con `dx->x`, comenzando por la primera nota de la secuencia original. Nótese que al abrir el editor del segundo chord-seq, se observa que elegimos la resolución de la altura (*Approx*) en 1/4 de tono.

Matrices

Las matrices son estructuras capaces de almacenar datos de forma multidimensional. OM sólo admite matrices bidimensionales, constituidas básicamente por filas y columnas, pero con la diferencia -en relación con otros lenguajes- que los datos de una misma matriz pueden ser de diversos tipos: números, acordes, objetos de la clase `bpf`, etc. Las matrices de OM fueron incorporadas originalmente para albergar parámetros destinados a la síntesis del sonido, pero su empleo se hace extensivo a cualquier aplicación.

Las matrices se crean a partir de la clase `class-array`, accesible desde el menú *Classes/Basic Tools/Array*. Las funciones asociadas a esa clase las encontramos en *Functions/Basic Tools/Array*.

La clase posee, por defecto, dos entradas y salidas: *self*, para conectar instancias o una lista de instancias de la clase, y *numcols*, que determina el número de columnas de la matriz. Las filas se agregan presionando *Control* + *k* en el teclado, y se eliminan con *Shift* + *Control* + *k*. Una vez creadas las filas, si pasamos el cursor del mouse sobre el *inlet* veremos leyendas tales como *:k0*, *:k1*, etc. A ese tipo de expresiones se las denomina en *LISP key parameters*, y constituyen un modo de ingresar parámetros adicionales a una función. Haciendo *click* sobre esos nombres podremos escribir otros que resulten más comprensibles en relación con los datos almacenados, recordando siempre anteceder nuestros nombres de clave con dos puntos (*:clave*, por ejemplo). En el gráfico que sigue vemos, a la izquierda, la clase `class-array` tal cual aparece al ubicar la caja sobre el *canvas*. Luego de apretar *m* podemos ver la información en su interior, en forma de gráfico. En nuestro caso agregamos 8 columnas (*inlet* 2) y tres filas (*inlets* 3 a 5).



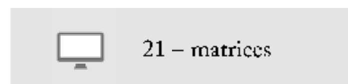
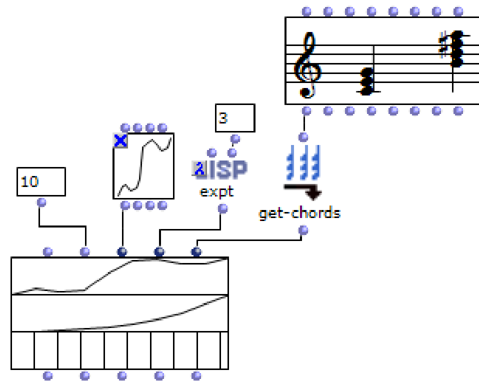


Figura 46.

La clase `class-array`

En la fila 1 establecimos una constante, el número 3, que se repite automáticamente en todas las columnas. En la fila 2 una lista de números cuya longitud es igual al número de columnas. Si la longitud de la lista fuera menor al número de columnas, sus elementos se repetirían cíclicamente hasta completar todas las columnas. Por último, en la fila 3, conectamos una función en modo *lambda*[33]: `om-random`. Cuando esto ocurre, la función es llamada tantas veces como columnas existen y asume como argumento el número de columna (1 a *n*) correspondiente. En este caso, el límite inferior del `om-random` es siempre 0 y el superior 1 para la primera columna, 2 para la segunda y así sucesivamente. La clase posee un editor, que se abre al hacer doble *click* sobre su caja. Desde allí podremos modificar los datos gráficamente.

Para el próximo ejemplo, creamos una matriz de 10 columnas y 3 filas en las que introducimos una función en `bpf`, la cual es muestreada de acuerdo a la cantidad de columnas; una función en modo *lambda*, que eleva los números de columna al cubo y dos acordes, extraídos de `chord-seq` mediante la función `get-chords`. Al abrir el editor se puede observar que los dos acordes se repiten cíclicamente a lo largo de las columnas, al igual que ocurre con las listas cuya longitud es menor al número de columnas.



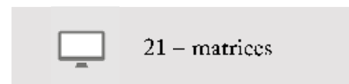


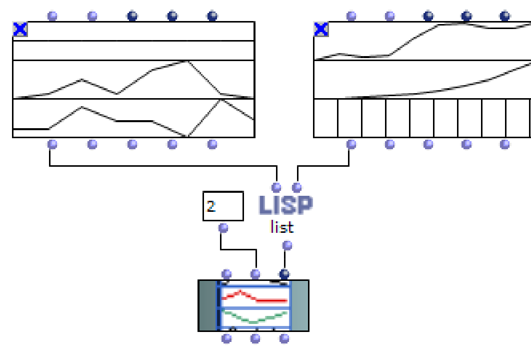
Figura 47.

Diversos objetos como elementos de una matriz

Varias matrices pueden combinarse en una sola matriz, como se aprecia a continuación. Ambas instancias son enlistadas y conectadas a la única fila creada en la nueva matriz (*:k0*). Al existir dos columnas, la primera matriz de la lista se ubica en la primera columna, y la segunda matriz en la columna restante. De haber más

[33] El modo *lambda* en OM será tratado posteriormente. Se emplea cuando se desea utilizar una función como argumento de otra.

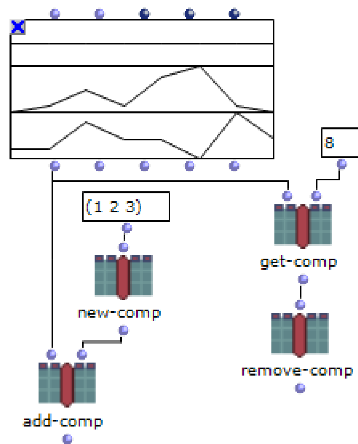
columnas las matrices ingresadas se repetirían cíclicamente. Al abrir el editor se puede apreciar la distribución de los elementos en filas y columnas.



21 – matrices

Figura 48. Combinación de matrices

Algunas de las funciones que acompañan a la clase `class-array` son `new-comp`, que crea una nueva columna; `add-comp`, que agrega la columna nueva a la matriz; `get-comp`, que obtiene la instancia de una columna en particular y `remove-comp`, que elimina una columna dada su instancia. Hay que considerar que la evaluación de estos objetos incide de forma destructiva sobre la matriz, modificando sus datos. Veamos el ejemplo que sigue.



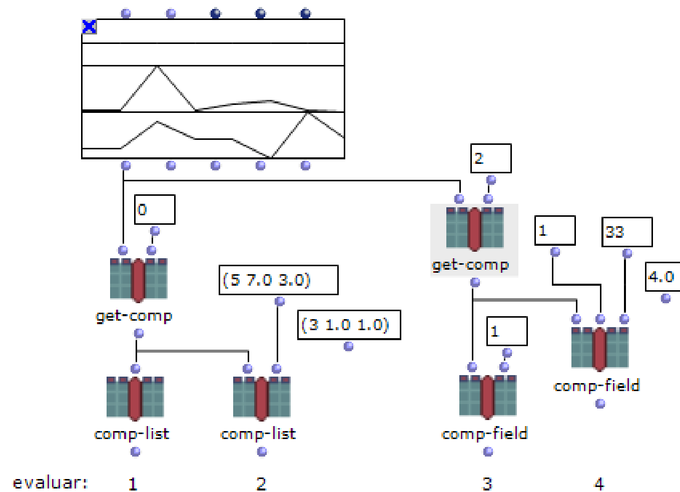
21 – matrices

Figura 49.

Funciones relacionadas con el uso de matrices

Con `new-comp` creamos una nueva columna, en cuyas filas se ubican los elementos 1, 2 y 3. La columna es agregada al final con `add-comp`, lo cual puede apreciarse al evaluar este objeto y observar lo que sucede en la matriz. La columna agregada (8) puede eliminarse con la mera evaluación de `remove-comp`.

Las funciones `comp-list` y `comp-field` sirven tanto para leer datos como para modificarlos, dependiendo de la cantidad de *inlets* que hayamos habilitado. `comp-list`, con un solo argumento, lee los valores de las filas de la columna referenciada con `get-comp` (y devuelve la lista `(3 1.0 1.0)`, en el ejemplo siguiente). Al mismo tiempo, si agregamos un *inlet* y cargamos nuevos valores (la lista `(5 7.0 3.0)` del ejemplo), la evaluación de `comp-list` modifica la columna de la matriz destructivamente.



21 – matrices

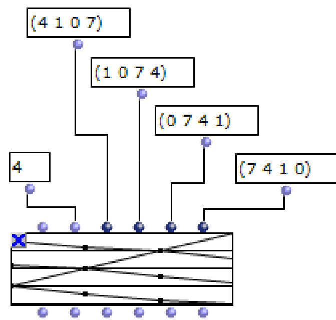
Figura 50. Otras funciones

De forma similar actúa `comp-field`, que permite modificar el valor de una fila particular, dentro de la columna referenciada. En este caso, el valor 4.0 es reemplazado por 33.

Para el ejemplo de la figura 51 emplearemos la clase `class-array` para almacenar los elementos de un cuadrado romano. El cuadrado romano posee la misma información en todas sus filas y columnas, y se obtiene a través de permutaciones circulares del contenido de la primera fila.

Si consideramos que los números representan grados cromáticos, las filas serían las voces, y las columnas los acordes, formándose así una polifonía. En tal caso, obtendríamos la misma interválica tanto horizontal como verticalmente.

4	1	0	7
1	0	7	4
0	7	4	1
7	4	1	0

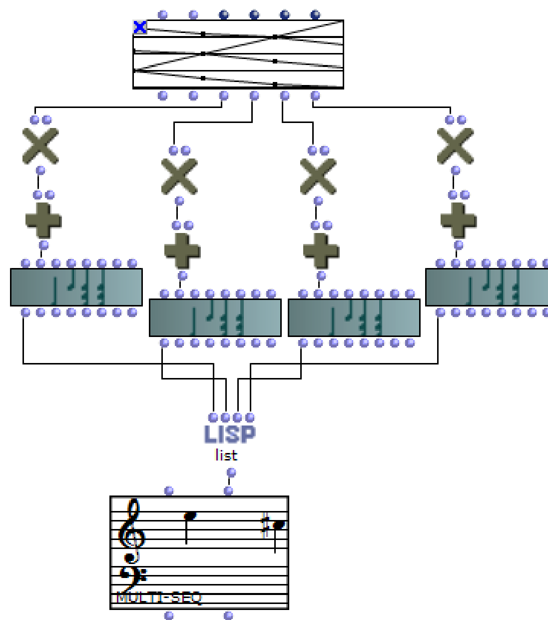


22 – matrices

Figura 51.

Uso de una matriz para almacenar un cuadrado romano

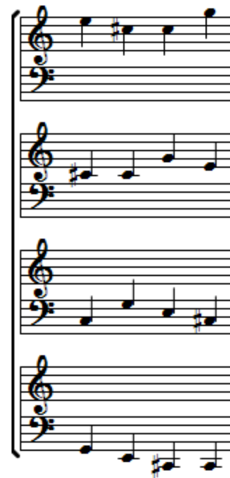
Una vez evaluada la clase, bloqueamos su contenido seleccionando el objeto y apretando la tecla *b*. Si lo duplicamos (con *Control + d*) podremos utilizarlo en otro momento. A continuación, convertimos los grados en notas MIDI –multiplicando por 100 y sumando la nota de base para cada voz. Luego, entramos los datos de cada fila en el *inlet* de notas de la clase `chord-seq`, según se aprecia en la figura.



22 – matrices

Figura 52. Utilización de los datos de la matriz

Al enlistar los objetos `chord-seq` podemos ahora incluirlos en la clase `multi-seq`, que hasta el momento no habíamos empleado. Esta clase nos va a permitir ver el resultado como una polifonía, ubicando cada secuencia en una voz distinta. Si hacemos doble *click* sobre su caja se abrirá el editor, mostrando la información ingresada en notación musical.



22 – matrices

Figura 53.

La matriz combinatoria en notación musical, representada en el editor de `multi-seq`

El cuadrado romano es la matriz combinatoria más simple y elemental. En el Capítulo 4 de este libro veremos la forma de construir otras más complejas, mediante el uso de una librería especialmente creada.

Control del flujo de datos

Al tratar los alcances de *LISP* hemos visto distintas funciones destinadas a controlar el flujo de la información en un programa. OM, además de las primitivas de *LISP*, cuenta con funciones propias para establecer condiciones, bucles y operadores lógicos. A las mismas se accede a través del menú *Functions/Kernel/Control*.

Condicionales

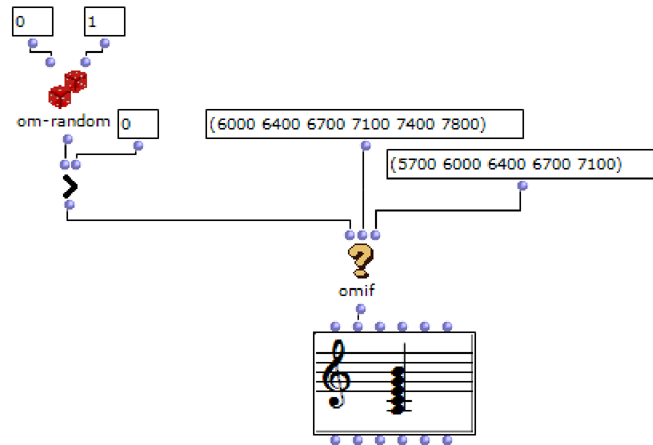
Según ya vimos, a través de los predicados es posible evaluar la veracidad o falsedad de una comparación. En *OpenMusic*, contamos con las funciones `OM<`, `OM>`, `OM<=`, `OM>=`, `OM=` y `OM/=`, capaces de comparar dos números y devolver `t` o `nil`.

23 – control de flujo 1

Figura 54. Predicados

Los predicados, en combinación con la función `omif`, nos permiten tomar decisiones vinculadas al flujo de datos dentro de un *patch*, como se observa en el siguiente ejemplo, en el que se decide aleatoriamente la elección de uno u otro acorde.

Dado que la cadena de evaluaciones se desarrolla de abajo hacia arriba, al ser validada la caja del acorde, el segundo *inlet* (el de las notas del acorde) solicita lo propio a la función `omif`. El primer *inlet* de `omif` (*if*), a su vez, solicita el resultado de un test. Si `om-random` devuelve 1, el predicado `om>` devuelve verdadero (`t`), pues 1 es mayor que 0. En este caso, `omif` evalúa aquello que se encuentra conectado a su segundo *inlet* (*then*) y copia el resultado a la salida. Caso contrario, si el predicado determina que la condición es falsa, se evalúa lo que se encuentra conectado al tercer *inlet* (*else*) de `omif`. El empleo de este tercer *inlet* es opcional, pero en nuestro caso resulta útil.



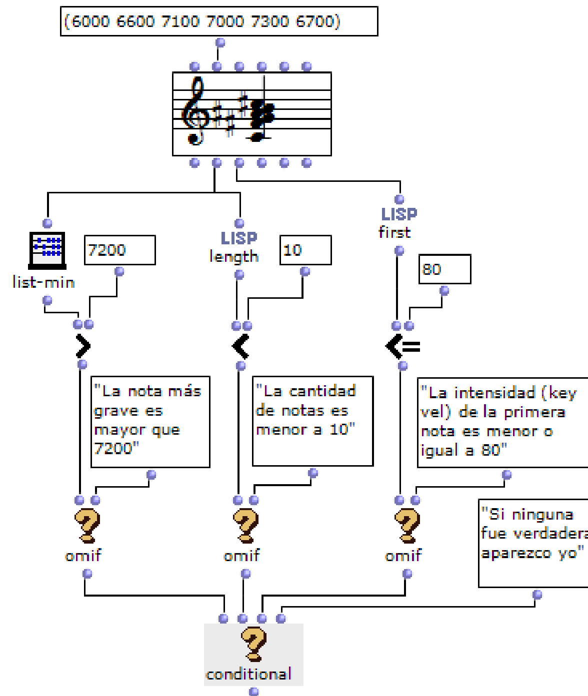
23 – control de flujo 1

Figura 55. La función `omif`

Cuando es necesario realizar una serie de evaluaciones, y no solamente una, recurrimos a `conditional`. Si conectamos dos o más cajas `omif` a sus *inlets*, la función las evalúa de izquierda a derecha, hasta que el resultado de alguna de ellas sea distinto de `nil`. Según veremos luego, `conditional` se comporta como un operador *or*.

En el ejemplo que sigue, partiendo de un acorde, verificaremos si alguna de las tres condiciones enunciadas se cumple o no: que la nota más grave del acorde se encuentre por encima de una nota dada; que la cantidad de notas sea menor a un número dado y que la intensidad de la primera nota tenga un *key velocity* menor o igual a 80. Al segundo *outlet* de `chord` (notas del acorde) conectamos la función `list-min`[34], que retorna el número más pequeño de una lista de números. De este modo obtenemos la nota más grave. También al segundo *outlet* conectamos la función `length`, que nos devuelve la cantidad de elementos de la lista de notas (la densidad polifónica). Finalmente, conectamos `first` al tercer *outlet*, que devuelve la lista de *key velocities* o intensidades de las notas del acorde.

[34] List-min se encuentra en el menú *Functions/Basic Tools/Arithmetic*.



24 – control de flujo 2

Figura 56. Condicionales

Primero preguntamos si la nota más grave es mayor a la nota 7200 (*do* 6), lo cual es falso. Luego, si la cantidad de notas es menor a 10, lo cual es cierto. En tercer lugar, si la intensidad de la primera nota del acorde es menor o igual a 80, que es falso. Y luego, por si ninguna condición resultara verdadera, afirmamos algo, sin condicionamientos. La primera condición no se cumple, por lo cual `conditional` la ignora. La segunda resulta verdadera y el segundo *inlet* del `omif` es evaluado, dando por respuesta la cadena de caracteres `"La cantidad de notas es menor a 10"`. Ni la tercera condición (intensidad de la nota) ni la afirmación siguiente serán tenidas en cuenta por `conditional`, dado que ya encontró una respuesta verdadera y lo considera suficiente.

Operadores lógicos

OM posee dos operadores lógicos, se trata de `omand` y `omor`, cuyo principio de funcionamiento es similar al de las macros `and` y `or` de *LISP*. La función `omand` evalúa cada una de sus entradas siguiendo el orden de izquierda a derecha. Si cualquiera de ellas resulta ser `nil` se suspende la ejecución y el objeto devuelve un

`nil`. Caso contrario, devuelve el valor de la última de sus entradas. Veámoslo en *LISP*:

```
> (and 1 nil 3)
NIL

> (and 1 2 3)
3
```

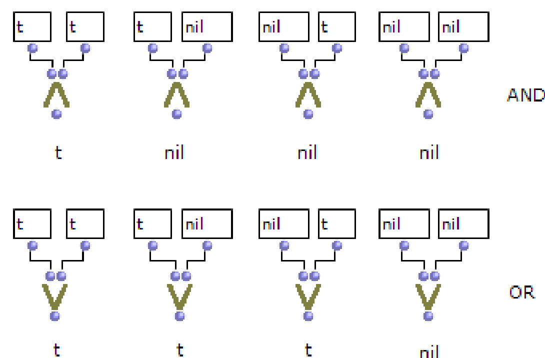
La función `omor` también evalúa cada una de sus entradas. Si cualquiera de ellas resulta no ser `nil` se suspende la ejecución y el objeto devuelve el valor de esa entrada. Caso contrario, devuelve `nil`.

```
> (or 1 2 3)
1

> (or nil nil 3)
3

> (or nil nil nil)
NIL
```

Dadas dos condiciones, al utilizar `omand` obtendremos un resultado verdadero (`t`) sólo si ambas se cumplen. Basta que una de las condiciones no se cumpla, o que ninguna lo haga, para que `omand` devuelva falso (`nil`). Este principio se hace extensivo a un mayor número de condiciones; para que `omand` devuelva verdadero, todas las condiciones deben estar dadas.



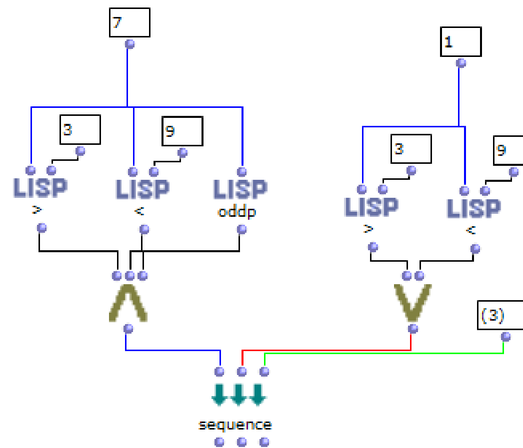
25 – control de flujo 3

Figura 57. Operadores lógicos

En relación con `omor`, siempre devuelve verdadero, salvo que todas las condiciones sean falsas. Aquí hay que considerar que no se trata de un *or* exclusivo, en el cual siempre deben estar presentes ambas posibilidades, es decir, `t` y `nil`, para que el

resultado sea verdadero. En este caso, con que al menos una o todas las condiciones sean verdaderas, devuelve verdadero.

En el gráfico siguiente, a la izquierda, vemos un uso posible de omand. Las condiciones son que el número a considerar sea mayor que 3, menor que 9 e impar. Solamente los números 5 y 7 cumplen las tres condiciones y hacen que omand devuelva t en su evaluación.



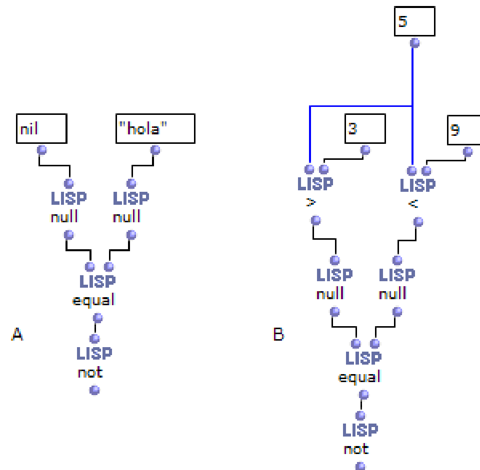
26 – control de flujo 4

Figura 58. Uso de omand y omor

En el *patch* de la derecha del gráfico anterior observamos el uso de omor. En este caso, el número considerado debe ser mayor que 3 o menor que 9, por lo cual sólo califican el 9 o los números mayores que 9, y el 3 o los menores a 3. No obstante, al ser un *or* no exclusivo, mientras se cumpla alguna de las condiciones, o ambas, el resultado es t.

En el mismo patch observamos, además, la función sequence, que fuerza la evaluación de todo lo que se halle conectado a sus *inlets*. Para obtener los resultados de las evaluaciones individuales podemos hacer *Control* + *click* sobre cualquier *outlet*.

A modo de ejercicio intentaremos construir un *or* exclusivo que compare números o cadenas de caracteres y que devuelva verdadero sólo si coexisten t y nil. En el ejemplo que sigue, *patch A*, comparamos la cadena "hola" con nil, y obtenemos t. Si pusiéramos dos cadenas o dos nil en ambas entradas, el resultado sería nil.



27 – control de flujo 5

Figura 59. Construcción de un *or* exclusivo

La función `null` de *LISP* convierte a un dato en `nil` y a `nil` en `t`. De esta manera, el dato (la cadena de caracteres) pierde su condición y se convierte en un booleano (concretamente en `nil`). A continuación nos preguntamos si ambos son distintos –la cadena convertida en `nil` y el `nil` convertido en `t`– mediante la combinación de `not` y `equal`. Si son diferentes se cumple la condición de un *or* exclusivo: o uno verdadero o el otro verdadero, pero no ambos verdaderos o falsos. Finalmente, en *B*, utilizamos nuestro *or* exclusivo para comparar dos números, como en el *patch* anterior.

Iteración

A fin de lograr la repetición de un proceso determinado empleamos un tipo de abstracción especial denominada `omloop`, a la cual se accede a través del menú *Functions/Kernel/Control*. Ubicando el objeto en el área de programación, y haciendo doble *click* sobre su ícono, se despliega una ventana que permite programar las características del bucle deseado. La ventana muestra diversos componentes, los cuales pueden dividirse en tres grupos: de iteración, de acumulación y de evaluación.

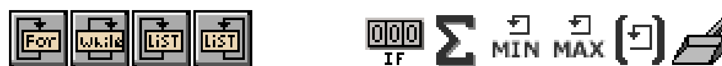


Figura 60. Objetos de iteración y acumulación

Los **componentes de iteración** determinan el modo en que se produce el proceso de repetición. Se trata de los siguientes objetos.

`forloop`: es similar a cualquier bucle *for* utilizado en la mayoría de los lenguajes. Consta de un valor numérico de partida, un valor de llegada y un incremento, que permite llegar por pasos desde el valor inicial al final. Si el número inicial es 1 y el final 4, con un incremento de 1, `forloop` generará -a petición de los evaluadores- cuatro iteraciones, arrojando sucesivamente los números 1, 2, 3 y 4.

`whileloop`: determina la continuidad o la interrupción de un bucle a través de la evaluación de una condición. Mientras extraemos los elementos de una lista, por ejemplo, podríamos detener la operación al detectar la presencia de un número, el cual sería detectado con `numberp`.

`listloop`: extrae y devuelve los elementos del primer nivel de una lista aportada como dato. Dada la lista `(a b (1 2 3) c)`, devuelve en cada iteración los elementos *a*, *b*, *(1 2 3)* y *c* en orden sucesivo.

`onlistloop`: en el primer ciclo devuelve la lista dato, y luego el `cdr` de las listas que resultan en cada iteración. Si procesamos la lista `(a b c d)`, por ejemplo, obtenemos *(a b c d)*, *(b c d)*, *(c d)* y *(d)*. Además, creando un nuevo *inlet*, es posible cambiar la función `cdr` utilizada por defecto, por otra. Si aplicamos, en cambio, `cddr` sobre la lista anterior obtendremos el `cdr` del `cdr`. En nuestro ejemplo el resultado sería *(a b c d)* y *(c d)*.

Por otra parte, los **acumuladores** son los encargados de almacenar los resultados que surgen de cada iteración. En este grupo encontramos los que a continuación se detallan.

`count`: cuenta los elementos extraídos por `listloop`, siempre y cuando sean éstos distintos a *nil*. Posee tres *outlets*. El primero devuelve la cuenta parcial en cada iteración; el segundo devuelve el resultado final y el tercero reinicializa el contador a 0, en el caso que sea evaluado.

`sum`: efectúa la suma de números contenidos en una lista, los cuales son extraídos con `listloop`. Posee tres *outlets*. El primero devuelve las sumas parciales en cada iteración; el segundo devuelve el resultado final y el tercero reinicializa la suma a 0, en el caso que sea evaluado.

`min / max`: busca el número más pequeño o el mayor de una lista, respectivamente. Los números son extraídos con `listloop`. Poseen tres *outlets*. El primero devuelve

el menor/mayor número hallado en cada iteración; el segundo devuelve el resultado final y el tercero reinicializa la memoria del menor/mayor número encontrado a -4.294.967.296 o 4.294.967.296, para comenzar una nueva comparación, de ser requerido.

`collect`: reúne los resultados de cada ciclo del bucle en una lista. El primer *outlet* devuelve en forma de lista lo recibido por cada iteración. El segundo, el resultado de la recolección al finalizar el bucle. El tercero inicializa el contenido de `collect` en `nil`, cuando se le solicita evaluación.

acum: acumula los resultados de procesar los elementos de una lista a través de una función dada. Posee tres *inlets*, en los cuales se reciben los valores de la lista, se establece un valor inicial para el procesamiento y se determina la función de procesamiento, en modo *lambda*. Los ejemplos que veremos luego ayudarán a comprender mejor los alcances de este acumulador.

Los **evaluadores** se encuentran al final de la cadena de objetos del bucle y son los encargados de solicitar la evaluación de esa cadena hasta que se cumpla la condición de finalización del bucle.

Figura 61. Componentes de evaluación

El objeto `initdo` no aparece inicialmente en la ventana de `omloop`, pero puede incluirse haciendo doble *click* sobre el fondo de la ventana y escribiendo su nombre. Su función es solicitar la evaluación de lo que se halle conectado a su *inlet*, antes que el bucle comience. Dicho de otro modo, permite realizar tareas de inicialización, previas a la ejecución de la iteración.

El objeto `eachtime` solicita y recibe el resultado de cada paso de la iteración, mientras que `finally` recibe el resultado final, y lo envía al *outlet* de `omloop`.

En el ejemplo siguiente, a `omloop` le agregamos dos *inlets*. En el primero ingresa una lista con grados cromáticos, y en la segunda los nombres de las notas. A través de la programación dentro de `omloop` trataremos de vincular en forma de sublistas cada grado con su nombre.

28 – iteración

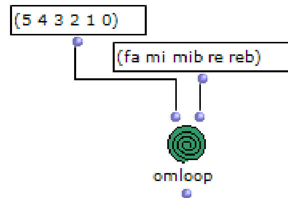
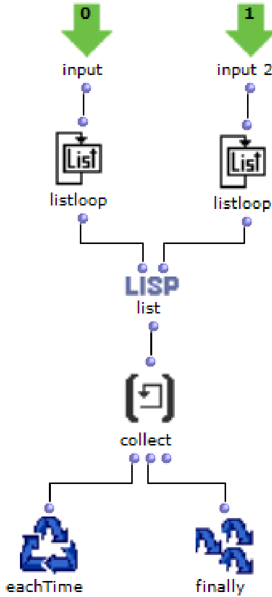


Figura 62.

`Omloop` *A*

Según se aprecia, en el contenido de `omloop` utilizamos dos componentes `listloop` para obtener los elementos individuales de cada lista. Luego conformamos la sublista mediante `list` y posteriormente recolectamos el resultado con `collect`. El resultado obtenido es:

```
OM => ((5 fa) (4 mi) (3 mib) (2 re) (1 reb))
```



28 – iteración

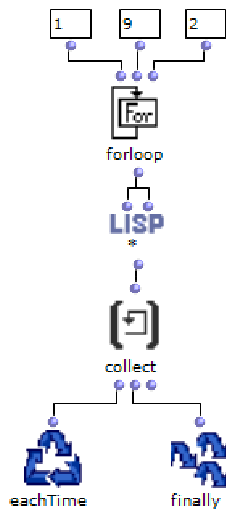
Figura 63.

Contenido del `omloop` *A*

Pero según se puede notar, el número 0, contenido en la primera lista, no aparece en el resultado. Eso es debido a que el proceso finaliza cuando se termina de recorrer la lista que posee la menor cantidad de elementos.

Debemos prestar especial atención al modo en que se conecta `collect` con `eachtime` y `finally`. El tercer *outlet* de `collect`, que sirve para reinicializar a `nil` los contenidos almacenados, no se utiliza en la mayoría de los casos.

El ejemplo que sigue emplea `forloop` para generar los números impares de 1 a 9 elevados al cuadrado. Que el número sea impar lo determina que el *for* comience por uno y el incremento sea 2.

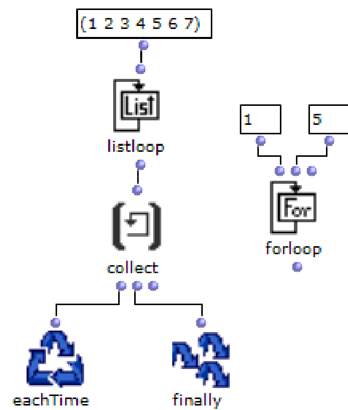


28 – iteración

Figura 64.

Contenido del `omloop` *B*

El empleo de `forloop` puede limitar la cantidad de valores leídos de una lista mediante `listloop`. La combinación de ambos, en el *patch* de la figura siguiente, determina que la longitud del resultado sea menor que la longitud de la lista tomada como dato.



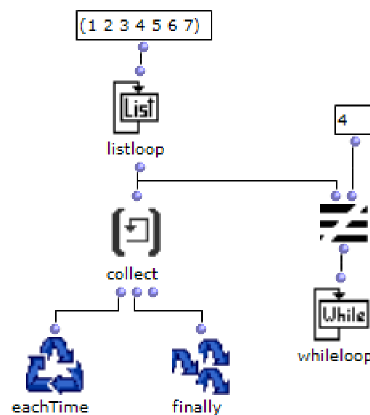
28 – iteración

Figura 65.

Contenido del `omloop` *C*

```
OM => (1 2 3 4 5)
```

Algo similar ocurre con `whileloop`. En el caso que sigue, lo empleamos para que el bucle continúe mientras el número leído no sea 4. Para establecer la condición "que sea distinto de" utilizamos el objeto `om/=` que se encuentra en el menú *Functions/Kernel/Control/Predicates*. De igual modo, podríamos haber utilizado la función primitiva de *LISP* `/=`. En ambos casos, las funciones comparan números y no listas.



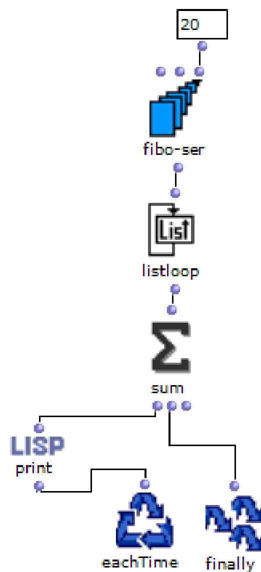
28 – iteración

Figura 66.

Contenido del `omloop` *D*

Según se observa en el *listener*, evaluando el objeto `omloop` *D*, el resultado es

```
OM => (1 2 3)
```



28 – iteración

Figura 67.

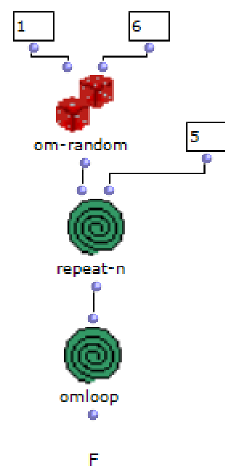
Contenido del `omloop` *E*

La versión *E* del mismo *patch* calcula la suma de todos los términos de la sucesión de Fibonacci menores a 20 (0, 1, 1, 2, 3, 5, 8, 13). La suma se efectúa con `sum`,

mientras que los términos de la sucesión los obtenemos con `fibo-ser`. Obsérvese que entre `eachtime` y `sum` colocamos un `print`. Éste nos permite visualizar los resultados parciales de la suma para cada ciclo del bucle.

```
OM > 0 1 2 4 7 12 20 33
OM => 33
```

La repetición de un proceso también puede ser lograda mediante `repeat-n`. El primer *inlet* recibe la operación a iterar, mientras que el segundo establece la cantidad de repeticiones. En el ejemplo *E* simulamos el hecho de arrojar un dado 5 veces, y anotar el número más pequeño obtenido, y el más grande.

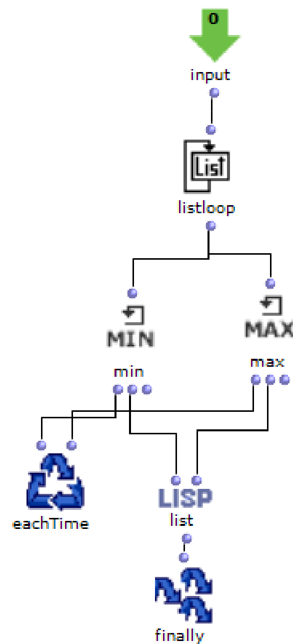


28 – iteración

Figura 68.

`omloop` *F*

`om-random` recibe los argumentos necesarios para producir números al azar entre 1 y 6. El objeto `omloop` recibe una lista con los cinco números al azar y la recorre mediante `listloop`. Cada elemento es comparado con los anteriores a fin de determinar el mínimo y el máximo, que se almacenan en una lista, antes de ser devueltos por `finally`. Dado que con `min` y `max` se establecen dos ramas de procesamiento en paralelo, es necesario asignarle un segundo *inlet* a `eachTime` a fin de que estos dos objetos sean evaluados.

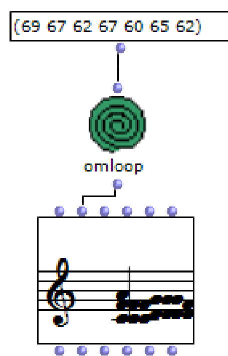


28 – iteración

Figura 69.

Contenido del `omloop` *F*

El ejemplo G, por otra parte, muestra el uso de `onlistloop`. Recordemos que el objeto devuelve la lista dato, y luego aplica por defecto `cdr` a la lista original y a los resultados de esa aplicación. En este caso lo emplearemos para crear un giro melódico que pierde su primera nota en cada repetición. Las alturas las declaramos con el formato de nota MIDI (*do* central es la nota MIDI 60) y los resultados los mostramos en notación musical a través del objeto `chord`.



28 – iteración

Figura 70.

`omloop` *G*

Haciendo doble *click* sobre el objeto `chord`, observamos la secuencia generada. Para ver las notas en sucesión, y no como un acorde, simplemente elegimos la opción *order* en el cuadro de selección inferior izquierdo.

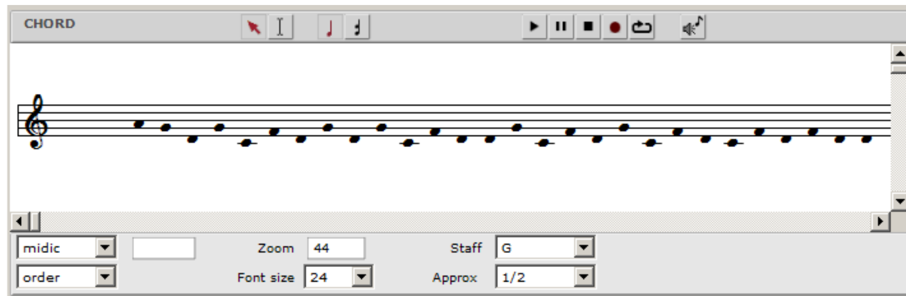
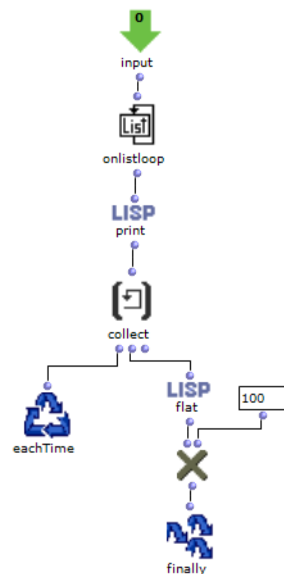


Figura 71. Editor del objeto `chord`

La programación dentro de `omloop` la observamos en la figura 72. Luego de imprimir y coleccionar los resultados parciales de *onlistloop*, se eliminan los paréntesis con `flat` y los elementos de la lista final (las notas MIDI) se multiplican por 100 para ser convertidas en *midicents*.



28 – iteración

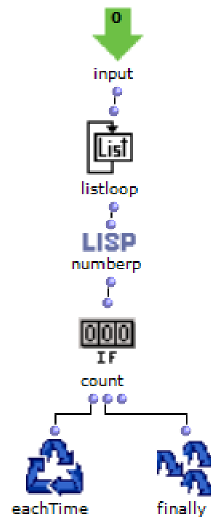
Figura 72.

Contenido del `omloop` *G*

Al evaluar el objeto `omloop` la impresión en el *listener* arroja la siguiente información, que permite comprender el modo de operación de `onlistloop`.

```
OM > (69 67 62 67 60 65 62)
OM > (67 62 67 60 65 62)
OM > (62 67 60 65 62)
OM > (67 60 65 62)
OM > (60 65 62)
OM > (65 62)
OM > (62)
OM => (6900 6700 6200 6700 6000 6500 6200 6700 6200 6700 6000
6500 6200 6200 6700 6000 6500 6200 6700 6000 6500 6200 6000
6500 6200 6500 6200 6200)
```

En el ejemplo *H* procedemos a contar los elementos numéricos de una lista. Para saber si el elemento es un número utilizamos `numberp`. Si el resultado es verdadero (`t`) el objeto `count` incrementa la cuenta en uno y si es falso (`nil`) lo ignora.

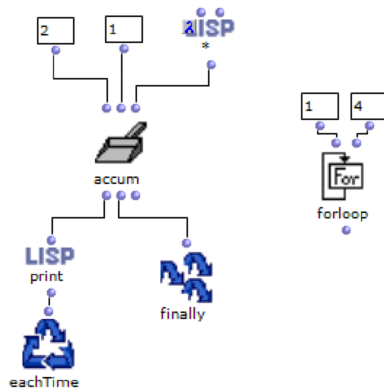


28 – iteración

Figura 73.

Contenido del `omloop` *H*

El ejercicio *I* hace uso de `accum` para calcular el resultado de 2^4 mostrando los resultados intermedios. Desde ya, habría otras maneras mucho más eficientes y directas de hacerlo, pero a través de este ejemplo se pretende mostrar de forma simple el funcionamiento del acumulador. Partiendo del número 1 se procede a multiplicar por 2 y a acumular el resultado parcial, repitiendo ese procedimiento cuatro veces. La cantidad de veces, en este caso, está determinada por `forloop`.



28 – iteración

Figura 74.

Contenido del `omloop` *I*

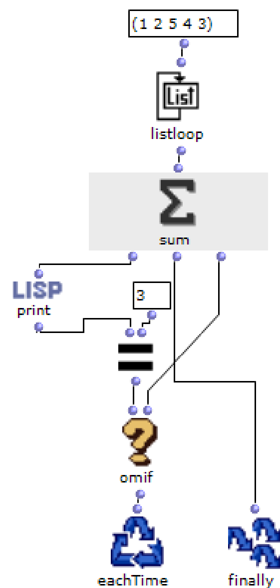
La impresión del valor parcial del acumulador por cada ciclo del bucle es

`OM > 2`	*eachTime*
`OM > 4`	*eachTime*
`OM > 8`	*eachTime*
`OM > 16`	*eachTime*
`OM => 16`	*Finally*

Observando atentamente el *patch* notamos una letra griega lambda (λ) sobre el objeto de la multiplicación. Según ya mencionamos, esta letra aparece cuando presionamos la tecla "l" con el objeto seleccionado, y determina que la función puede ser utilizada como argumento de otra función. En nuestro caso, la función * se convierte en argumento de la función `accum`. Posteriormente, nos referiremos a las funciones *lambda* con más detalle.

En el ejemplo *J* hacemos uso del tercer *outlet* de los componentes de acumulación, que al ser evaluado permite inicializar la memoria del objeto. Aquí deseamos obtener la suma de los valores numéricos de los elementos de la lista dato, pero inicializar la suma al valor 0 si el resultado parcial da 3. El primer *outlet* de `sum` arroja las sumas parciales, por lo cual allí es donde vamos a preguntar si el número calculado es 3, utilizando `omif`. En el primer *inlet* de `omif` conectamos la condición que debe cumplirse (¿es $x = 3$?). Si es cierto (`t`) el objeto `omif` solicitará evaluación a aquello conectado a su segundo *inlet*, con lo cual la memoria de `sum` se inicializará. Los resultados presentados a través de `print` en el *listener* son:

`OM > 1`	Primer número leído (1).
`OM > 3`	El segundo número (2) se suma al anterior y da 3. Reiniciar.
`OM > 5`	Se reinició el acumulador y se lee el tercer número (5).
`OM > 9`	El cuarto número (4) se suma al anterior.
`OM > 12`	El quinto (3) se suma al anterior.
`OM => 12`	El resultado final es 12.



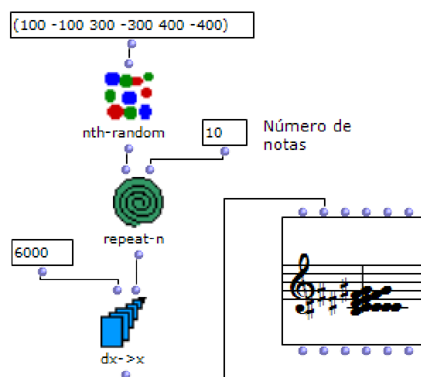
28 – iteración

Figura 75.

Contenido del `omloop` *I*

A continuación, veremos otro uso de `repeat-n`. A través del siguiente *patch* generamos una secuencia de 10 notas utilizando solamente tres intervalos melódicos (segunda menor, tercera menor y tercera mayor). Los intervalos están expresados en *midicents*, y los valores positivos representan a los intervalos ascendentes, mientras que los negativos a los descendentes.

Cada intervalo melódico es elegido al azar, utilizando `nth-random`. Si evaluamos el objeto `repeat-n` veremos una lista generada al azar con los elementos de la lista dato de semitonos. Cada vez que lo evaluemos el resultado seguramente cambiará, dado que es aleatorio. Dados los intervalos, construimos la sucesión de alturas con `dx->x`, estableciendo en el primer *inlet* la nota de comienzo (6000). Las notas siguientes son generadas por este objeto sumando a la nota de partida, y luego a las siguientes, los intervalos positivos o negativos.



29 – iteración

Figura 76.

Secuencia al azar a partir de interválica dada

A fin de aplicar la iteración nos proponemos generar una serie de números aleatorios a través de una distribución normal, también llamada distribución de Gauss. De acuerdo a esta distribución, la mayor densidad se concentra en torno a la media y decrece hacia los extremos, siguiendo la forma de una campana.

Cada término de la secuencia a obtener es calculado a través de una repetición, en la cual se suma una cantidad dada de números aleatorios entre 0 y 1. Al valor obtenido se le resta la mitad del número de repeticiones del bucle, se lo multiplica por la desviación estándar y se le suma la media[35]. La desviación estándar es una medida de la dispersión de los números en relación con su promedio.

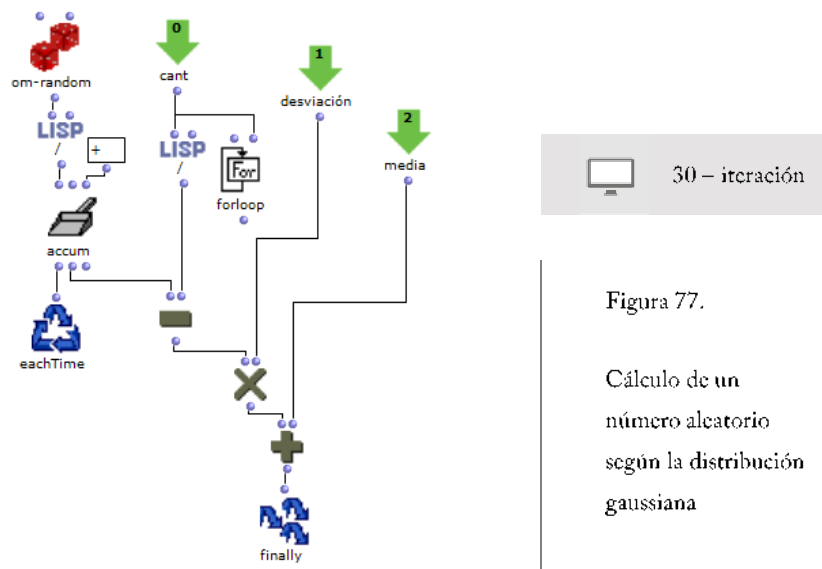


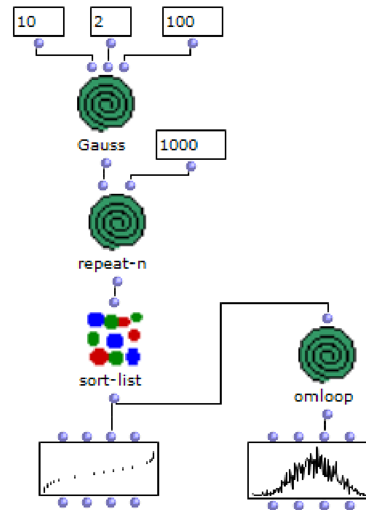
Figura 77.

Cálculo de un número aleatorio según la distribución gaussiana

El *patch* completo es el que se muestra a continuación. La cantidad de repeticiones fijada para la obtención del número aleatorio es 10, mientras que la desviación la fijamos en 2 y la media en 100. A través de `repeat-n` generamos 1000 números aleatorios acordes a la distribución normal, ordenamos la lista y graficamos con `bpf` la secuencia. Luego, con otro bucle calculamos la densidad de aparición de esos números, y al dibujarla vemos representada la típica campana de Gauss.

---

[35] Existen diversos métodos para realizar este proceso. Para el ejemplo hemos adaptado la programación que se presenta en P. Windsor y G. De Lisa. *Computer Music in C*. Tab Books. Blue Ridge Summit, PA. USA. 1991.

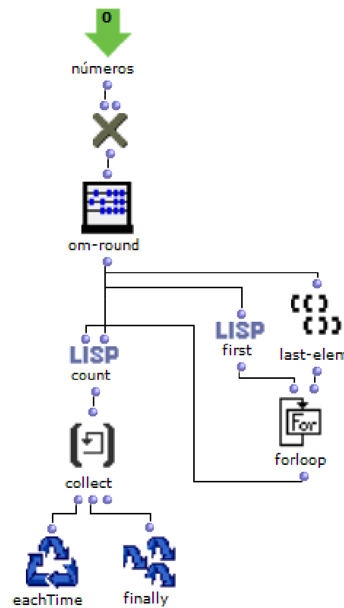


30 – iteración

Figura 78.

Obtención de la colección de números siguiendo la distribución normal

La curva de densidad es obtenida contando la cantidad de repeticiones de cada valor (con `count`), partiendo del más chico y llegando hasta el mayor con `forloop`.



30 – iteración

Figura 79.

Obtención de la curva de densidad

Realizando los escalamientos necesarios, la lista de números al azar podría ser ahora utilizada como control de algún parámetro sonoro o musical.

Notación musical

Una de las principales ventajas del entorno gráfico de programación es la posibilidad de ver los resultados de nuestros programas en notación musical. OM cuenta con las siguientes clases, destinadas a la construcción de estructuras musicales.

- `note`
- `chord`
- `chord-seq`
- `multi-seq`
- `voice`
- `poly`

Las tres primeras forman parte de las clases denominadas "armónicas"; `voice` integra la categoría de clases "rítmicas" y tanto `multi-seq` como `poly` pertenecen al grupo de las clases "polifónicas". Por otra parte, si consideramos el modo en que el tiempo es representado, otra clasificación resulta posible. Las secuencias de acordes (`chord-seq` y `multi-seq`) expresan la duración de los eventos y de los intervalos de ataque en milisegundos. Las voces y la polifonía creada por superposición de voces (`voice` y `poly`) lo hacen a través de la notación tradicional, empleando figuras y silencios. Por último, tanto las notas como los acordes forman parte de ambos grupos, ya que son considerados átomos.

Todas estas clases poseen un editor asociado que permite modificar el contenido de sus instancias por medios gráficos. Se accede al editor haciendo doble *click* sobre la caja de la clase.

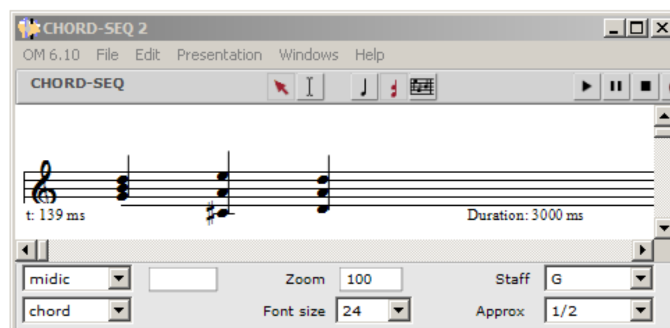


Figura 80. Editor del objeto `chord-seq`

Desde allí, podremos agregar notas (*Control + click*, eligiendo la herramienta adecuada); borrarlas (*Backspace*); ver el canal MIDI de cada nota, duración, dinámica, puerto MIDI u *offset* del menú desplegable; seleccionar el tamaño de fuente y los pentagramas y claves a mostrar, y aproximar los *midicents* desde el semitono hasta un 1/16 de tono.

Si seleccionamos la caja de una clase y presionamos *m* podremos ver su contenido. Si presionamos, en cambio *n* veremos el nombre de la clase, escrito sobre la misma caja.

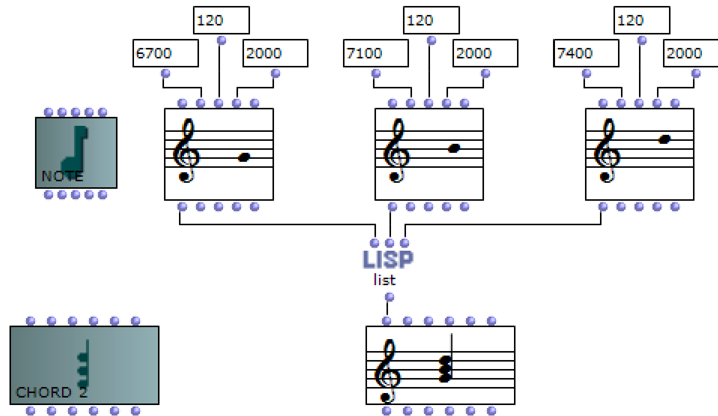
La clase `note` posee 5 entradas y salidas:

- *self*: la entrada acepta una instancia de la misma clase para representar su contenido. La salida devuelve una instancia de la clase.
- *midic*: nota expresada en *midicents*. Una nota en *midicents* está compuesta por un número de nota MIDI y una cantidad de cents[36] de desafinación, en relación con el sistema temperado y el *la* de 440 Hz. Como referencia para los números de nota, la norma MIDI considera que el *do* central equivale a 60.
- *vel*: intensidad de la nota, medida en términos de *key velocity* (1 a 127).
- *dur*: duración de la nota en milisegundos.
- *chan*: canal MIDI en el que es transmitida.

Una lista de notas conforma un acorde. El acorde puede ser entendido como tal, o como un conjunto de notas que conservan el orden en el que fueron ingresadas. En tal sentido, una serie dodecafónica podría representarse en OM empleando la clase `chord`.

---

[36] Un *cent* es la centésima parte de un semitono.



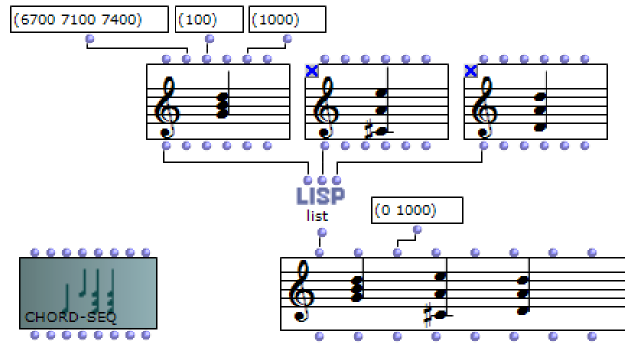
31 – notación 1

Figura 81. Las clases `note` y `chord`

A continuación veremos los parámetros del acorde, tanto de entrada como de salida. Debemos considerar ahora que los *inlets* 2 a 5 requieren necesariamente de listas, dado que a cada elemento de la lista le va a corresponder una nota del acorde. En los casos en que todas las notas del acorde tengan el mismo valor de algún parámetro, resulta suficiente escribir ese único valor entre paréntesis.

- *self*: la salida devuelve una instancia de la clase, la entrada acepta una instancia de la misma clase o una lista de objetos `note` para representar su contenido.
- *lmidic*
- *lvel*
- *loffset*: lista de retardos de cada nota a partir del ataque del acorde, en milisegundos. Puede determinar un arpegio, por ejemplo.
- *ldur*
- *lchan*

Una lista de acordes, por otra parte, conforma una secuencia de acordes. Estas secuencias se representan a través de la clase `chord-seq`.



31 – notación 1

Figura 82. Las clases `chord` y `chord-seq`

Los parámetros de una secuencia de acordes son:

- *self*
- *midic*
- *lonsets:* lista de tiempos de ataque, en milisegundos. Si solo se especifican dos ataques, `(0 1000)` por ejemplo, ese intervalo se propaga para el resto de los acordes que siguen.
- *ldur*
- *lvel*
- *loffset*
- *lchan*
- *legato:* porcentaje de duración de la nota. un 100% determina el máximo *legato*. Es un número, no una lista, y afecta a todos los acordes de la secuencia.

Una voz es una secuencia melódico-rítmica. Los parámetros de la clase `voice` son:

- *self*: la entrada admite una voz o una secuencia de acordes.
- *tree*: un árbol rítmico, que se tratará a continuación.
- *chords*: un acorde o lista de acordes de los cuales se toman las notas para la voz. También una instancia de las clases `chord` o `chord-seq`.
- *tempo*: indicación metronómica de la negra.
- *legato*: porcentaje de solapamiento entre dos acordes, calculado a partir de la duración del segundo acorde.
- *ties*: sublistas (una por acorde) que indican las notas a ligar.

Árboles rítmicos

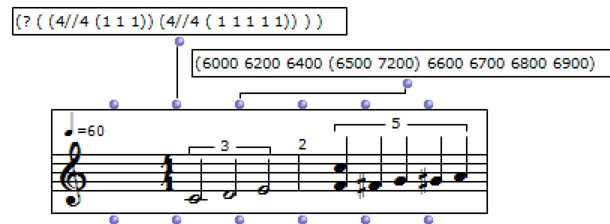
Los árboles rítmicos contienen la información relativa a los ritmos de un fragmento musical a ser representado por una clase `voice`. Incluyen la longitud del fragmento en número de compases y una lista de compases con sus subdivisiones, expresadas como proporciones. Cuatro negras en un compás de 4/4, por ejemplo, se definen de alguno de los siguientes modos:

a) `( ? ( ( 4//4  (1 1 1 1) ) ) )`
b) `( ? ( ( (4 4) (1 1 1 1) ) ) )`

El signo de interrogación puede ser puesto en lugar del número de compases, dado que OM es capaz de calcular la cantidad, cada vez que detecta ese signo.

Un tresillo de blancas en un compás, y un quintillo de negras en otro, quedarían representados así:

```
(? ( (4//4 (1 1 1)) (4//4 ( 1 1 1 1 1)) ) )
```



32 – notación 2

Figura 83. Árboles rítmicos

Cuando deseamos realizar una subdivisión interna (dos corcheas en lugar de una negra, por ejemplo) ponemos paréntesis al número en el que se va a producir la división del tiempo y agregamos en una sublista el modo en que se subdivide. Si deseáramos escribir cuatro negras en un compás, pero la tercera subdividida en dos corcheas, nos quedaría así:

```
( ? ( ( 4//4  (1 1 (1 (1 1)) 1) ) ) )
```

Y una negra, dos corcheas, otra negra y cuatro semicorcheas quedarían de este modo:

```
(? ( (4//4 ( 1 (1 (1 1)) 1 (1 (1 1 1 1)) ))))
```

(? ((4//4 (1 (1 (1 1)) 1 (1 (1 1 1 1))))))

(6000 6200 6400 6500 6600 6700 6800 6900)

♩=60

32 – notación 2

Figura 84. Subdivisiones

En la descripción de las divisiones o subdivisiones de un compás lo que resulta relevante es la proporción entre los números, más que el valor de los mismos. Un compás de 4/4 con blanca y dos negras puede expresarse tanto con 2 1 1, como con 4 2 2, como se aprecia en el ejemplo siguiente.

(? ((4//4 (2 1 1)) (4//4 (4 2 2))))

(6000 6200 6400 6500 6600 6700)

♩=60

2

32 – notación 2

Figura 85. Proporción numérica

Los silencios son representados mediante números negativos, mientras que las ligaduras de prolongación se especifican agregando un punto y un cero al número (figura) donde la ligadura finaliza. En el ejemplo que sigue, completamos un compás de 4/4 mediante 3 1 2.0 y -2. Las proporciones numéricas determinan que las figuras posibles son negra con puntillo (3), corchea (1), negra (2) y negra (2). Como a la primera negra se le agrega ".0", ésta se liga con la corchea que la precede. Y el signo menos sobre la última negra hace que la figura se convierta en silencio.

(? ((4//4 (3 1 2.0 -2))))

(6000 6200)

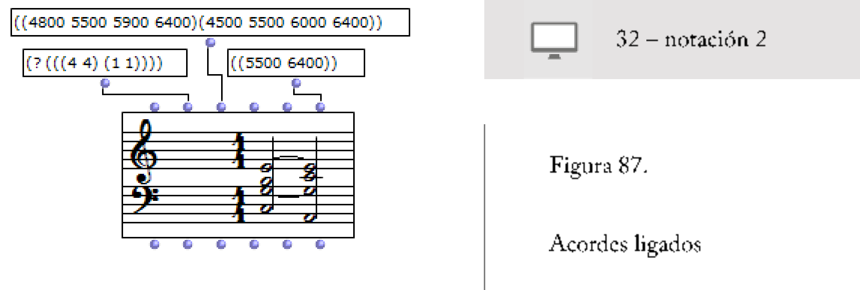
♩=60

32 – notación 2

Figura 86.

Silencios y ligaduras

Cuando trabajamos con acordes dentro de una voz, y aplicamos esta forma de escribir las ligaduras, todas las notas del acorde resultan ligadas. Cuando deseamos realizar ligaduras parciales es necesario declararlas en el último *inlet* de `voice`.



32 – notación 2

Figura 87.

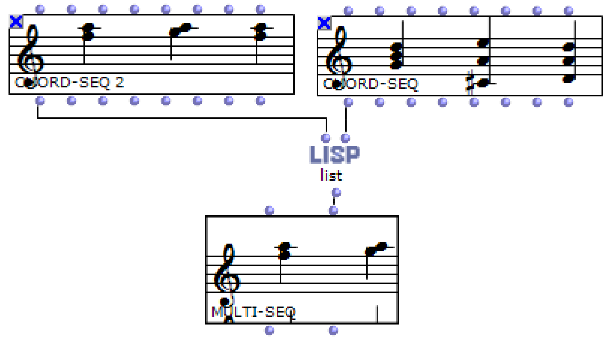
Acordes ligados

Los objetos polifónicos son `multi-seq` y `poly`.

Los parámetros de `multi-seq` son:

- *self*
- *chord-seqs:* una lista de objetos `chord-seq`.

En el siguiente ejemplo, ponemos en una lista dos objetos `chord-seq` y luego ingresamos esa lista en el segundo *inlet* de `multi-seq`. Después de evaluar, y al abrir el editor, observamos que cada secuencia es ubicada en un pentagrama distinto, del grupo formado. Obsérvese que los objetos de ambas secuencias fueron bloqueados intencionalmente, pues de otro modo, al evaluar `multi-seq` aparecería en ambos pentagramas la nota *do*, que se muestra por defecto cuando creamos una nueva instancia (nota MIDI 6000 del segundo *inlet* de `chord-seq`).



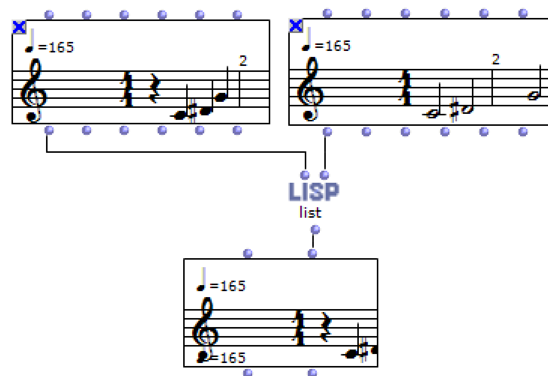
33 – notación 3

Figura 88. La clase `multi-seq`

En relación con poly, sus parámetros son los siguientes:

- *self*
- *voices:* una lista de objetos voice.

En el ejemplo que sigue, combinamos el contenido de dos objetos voice en una clase poly.

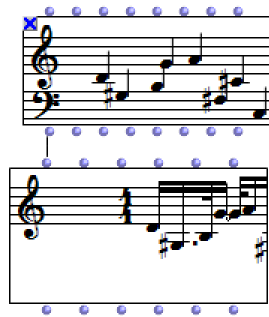


33 – notación 3

Figura 89. La clase poly

Funciones relacionadas con la notación

En *Functions/Score* encontramos diversas funciones útiles para aplicar en notación musical. Pero antes de comenzar a analizarlas, veamos qué sucede cuando conectamos una secuencia de acordes con una clase voice. El objeto chord-seq, que se muestra a continuación, contiene una secuencia cuyos datos fueron generados previamente. Según vimos, en este objeto las duraciones y los tiempos de ataque se establecen en milisegundos. Al conectarlo a voice, y al evaluar este último, las notas aparecen representadas en notación musical a través de un proceso de cuantización.



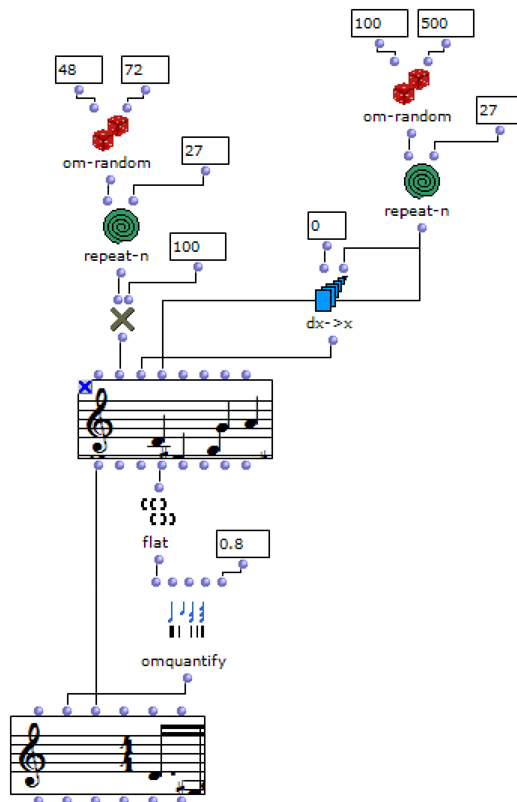
34 – notación 4

Figura 90.

Cuantización

En las preferencias del menú de OM, en *Quantification*, es posible establecer los valores por defecto que determinan el modo en que se realiza la cuantización rítmica. Sin embargo, si deseamos realizar esta tarea directamente sobre el *patch*, podemos recurrir a la función `omquantify`.

Para el ejemplo siguiente, generamos aleatoriamente los datos de una secuencia. Una vez obtenida la lista de duraciones, los tiempos de ataque son calculados con dx->x, comenzando en el tiempo 0 y acumulando los valores de la lista. El cuarto *outlet* de `chord-seq` devuelve la lista de duraciones, pero poniendo a cada valor entre paréntesis, por lo cual es necesario aplicar `flat`.



34 – notación 4

Figura 91.

La función `omquantify`

Se observa, por otra parte, que el tercer *inlet* de `voice` no sólo admite una lista de acordes, sino la instancia misma del objeto `chord-seq`, según lo mencionamos al detallar sus entradas y salidas.

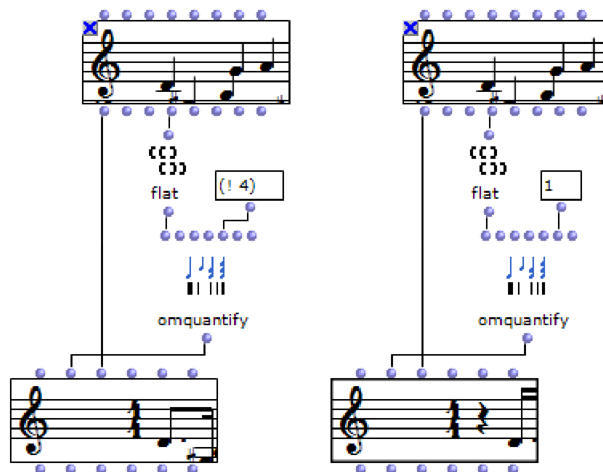
Los parámetros de `omquantify` son los que siguen:

- *self*: lista de duraciones a cuantizar.
- *tempi*: un número o una lista de números, uno por cada compás, que determinan el *tempo*.
- *measures*: una lista que especifica el o los indicadores de compás. Por ejemplo, `((4 4)(2 8)(3 4))`. Por defecto es `(4 4)`.
- *max*: subdivisión máxima. El número 8 equivale a la fusa.

Los parámetros opcionales son:

- *forbid*: lista de subdivisiones prohibidas de un tiempo del compás. La lista (9 11), por ejemplo, prohíbe la cuantización empleando estos valores irregulares. Si en lugar de números empleamos sublistas, cada una de ellas se vincula a un compás de la secuencia. Si no hay restricciones en un compás específico, se emplea una lista vacía en su lugar. Cuando una lista o sublista posee como primer elemento el signo "!" significa que las subdivisiones indicadas son obligatorias.
- *offset*: un corrimiento del inicio de la secuencia equivalente a una fracción del tiempo del compás.
- *precis*: un valor entre 0 y 1. Mientras mayor es el valor, mayor precisión se emplea en la cuantización, en detrimento de la simplicidad.

En el ejemplo que sigue, a la izquierda, forzamos la cuantización a valores múltiplos de la semicorchea, mediante `(! 4)`. A la derecha, establecemos un *offset* de 1 tiempo de compás, o sea de una negra.

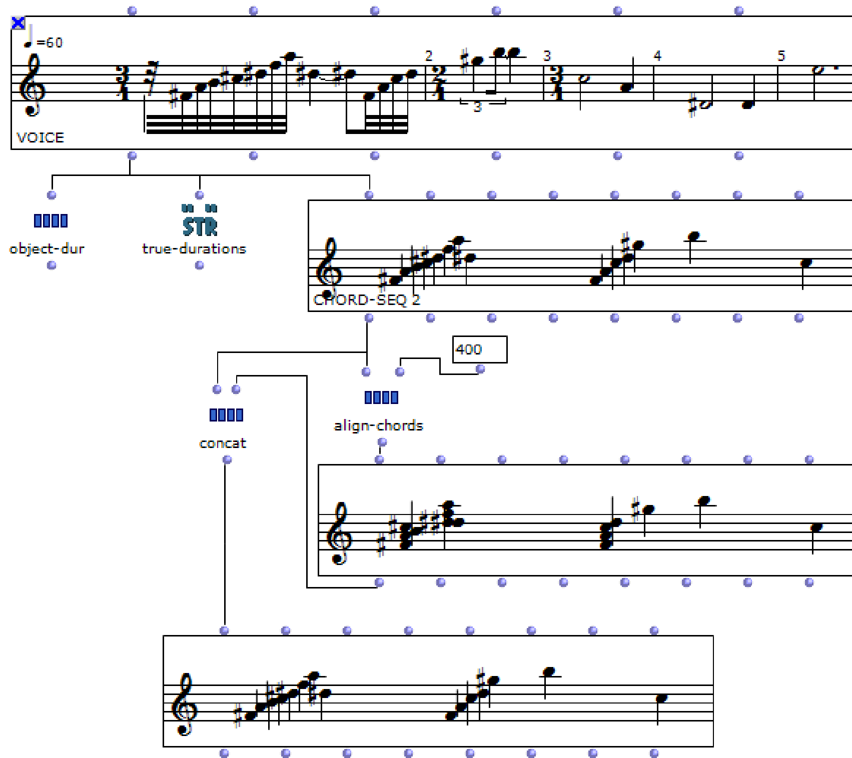


34 – notación 4

Figura 92. Opciones de `omquantify`

Siguiendo con el análisis de las funciones, `object-dur` devuelve la duración total de la secuencia almacenada en un objeto musical, en milisegundos. `true-durations`, por otra parte, retorna la lista de duraciones parciales de una secuencia, considerando los silencios y asignándole a éstos un valor negativo. Finalmente, `align-chord` convierte fragmentos melódicos, cuyas notas se encuentran por debajo de un umbral especificado en milisegundos, en acordes.

En el ejemplo siguiente, hacemos uso de estas funciones. En primer término, calculamos la duración total de una secuencia y la de sus notas y silencios. Seguidamente, la convertimos a escritura analógica mediante `chord-seq`, para poder transformar los giros melódicos, cuya duración es menor a 400 ms, en acordes. Por último, concatenamos la secuencia original y la transformada en una sola, con la función `concat`.

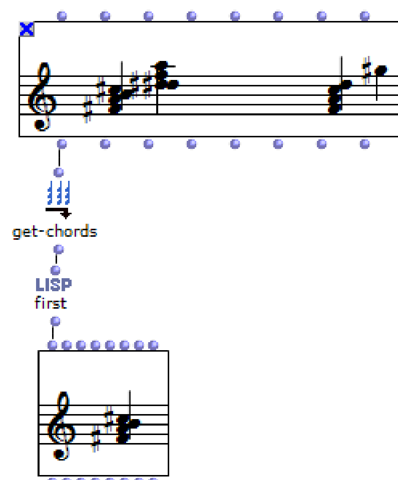


35 – notación 5

Figura 93. Otras funciones

Por medio de la función `get-chords` es posible obtener las instancias de los acordes de una secuencia almacenada en objetos `voice`, `chord-seq`, `poly` o `multi-seq`. La información relativa a los tiempos de ataque es removida por la función.

En el ejemplo que sigue tomamos el primer acorde de una secuencia y lo mostramos a través de la clase `chord-seq`.

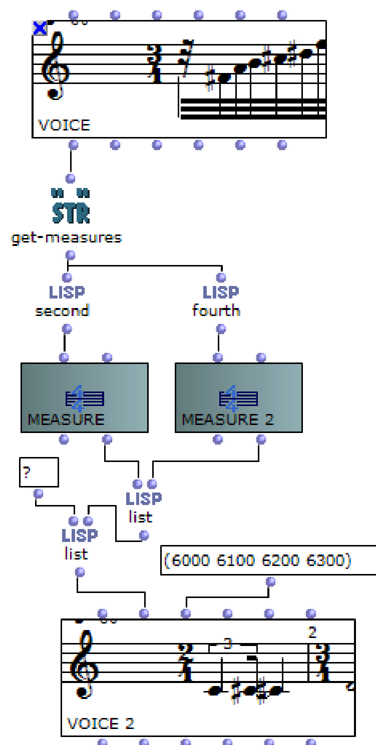


36 – notación 6

Figura 94.

La función `get-chords`

La función `get-measures`, por otro lado, permite obtener las instancias de los compases contenidos en una voz. En el *patch* que sigue, tomamos de una secuencia original el segundo y el cuarto compás. Nótese que `measure` es una clase, y que del segundo *outlet* extraemos la estructura rítmica del compás. Al formar una lista con ambas estructuras, y al agregar el signo "?", creamos un árbol rítmico que puede ser interpretado por `voice`. Paralelamente, agregamos una lista de *midicents*, generando así una nueva secuencia, a partir de dos compases escogidos de otra secuencia, pero con las notas cambiadas.



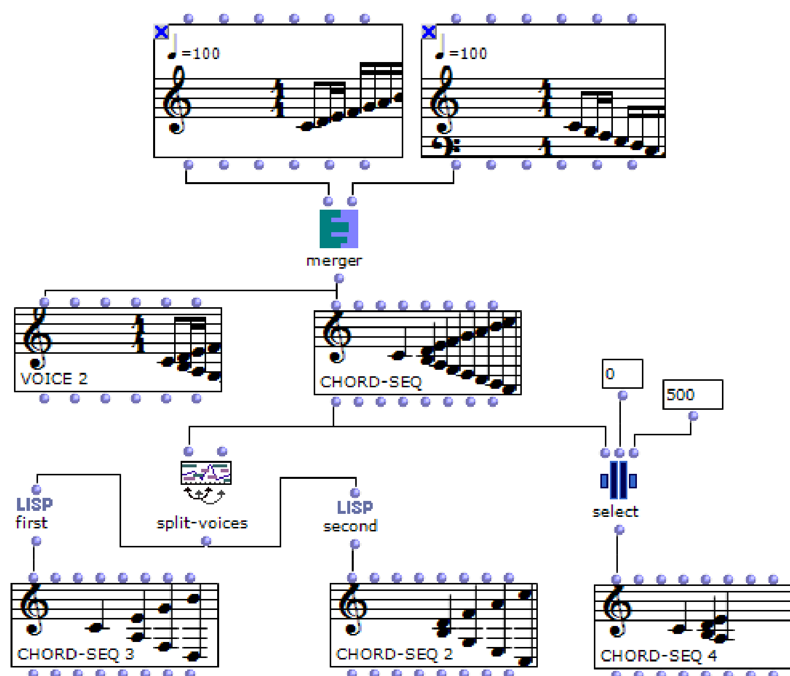
36 – notación 6

Figura 95.

La clase `measure`

La función `merger` fusiona dos voces, compases o secuencias de acordes en una nueva voz, compás o secuencia de acordes. En el ejemplo que sigue, combinamos dos escalas -una ascendente y otra descendente- en una voz (izquierda) y en una secuencia de acordes (derecha). Luego, con `split-voices`, separamos las notas o los acordes que, dispuestos sucesivamente, experimentan un solapamiento con sus vecinos. La superposición de los distintos bicordios –aquellos que surgen de la fusión de ambas escalas- es mínimo, pero suficiente como para que podamos obtener las notas pares por un lado, y las impares por otro. Por otro lado, con `select`, especificamos un intervalo temporal y extraemos los eventos que tienen lugar en su interior. Si utilizáramos la función `select` con una clase `voice` o `poly` -y no con

`chord-seq` o `multi-seq-` en lugar de milisegundos deberíamos establecer números de compás.



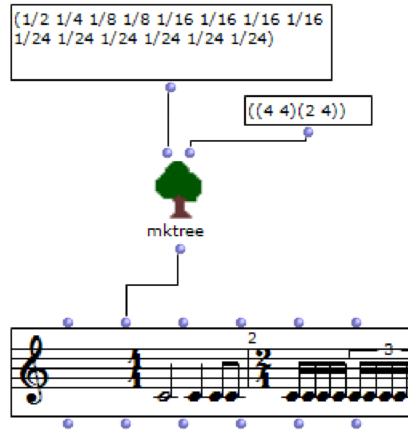
37 – notación 7

Figura 96. Otras funciones

Tratamiento de árboles rítmicos

En el menú *Functions/Score/Trees* hallamos diversas funciones dedicadas al procesamiento de listas destinadas a la representación de ritmos musicales, a las cuales hemos denominado árboles rítmicos. Si bien los árboles permiten expresar ritmos en cualquier nivel de complejidad, contamos con un método de codificación más sencillo y conciso, basado en fracciones. En el ejemplo siguiente asignamos ½ a la blanca, ¼ a la negra y siguiendo el mismo criterio divisivo, fracciones distintas a cada una de las figuras. En un seisillo de semicorcheas, por ejemplo, seis semicorcheas sumadas equivalen a una negra (1/4), y dividiendo el valor de la negra por 6 hallamos la fracción representativa de la semicorchea de seisillo (1/24). La escritura basada en fracciones puede transformarse en un árbol rítmico empleando la función `mktree`, que en su primer *inlet* recibe la lista de fracciones a convertir, y en

el segundo *inlet* un único indicador de compás en una lista, o una lista con varios indicadores de compás dentro de sublistas.

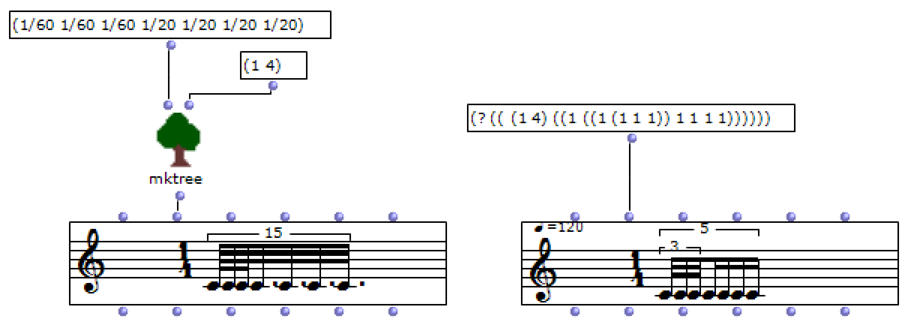


38 – árbol rítmico 1

Figura 97.

Función `mktree`

Las fracciones no pueden representar con total precisión de escritura todas las variantes rítmicas existentes (valores irregulares dentro de valores irregulares, por ejemplo), sin embargo, en la mayoría de los casos cumplen con los propósitos más generales. En el siguiente *patch*, intentaremos representar un quintillo de semicorcheas, en el cual la primera semicorchea sea un tresillo de fusas. Las fracciones correspondientes son 1/20 para la semicorchea de quintillo y 1/60 para la fusa de tresillo en su interior.

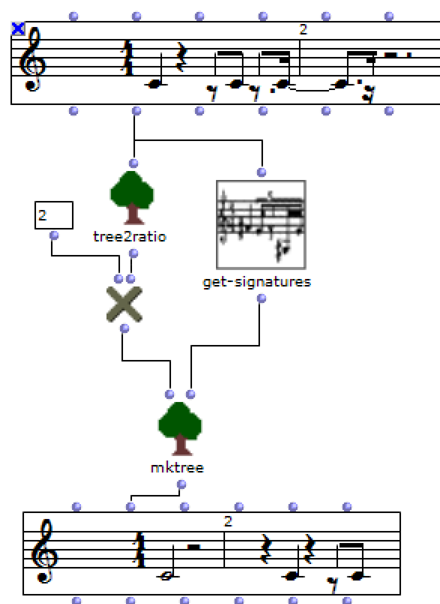


38 – árbol rítmico 1

Figura 98. Comparación de modos de notación

En la representación lograda a través de árboles rítmicos, las ambigüedades quedan descartadas por completo. Nótese cómo se subdividió numéricamente la primera figura del quintillo.

De forma contraria, un árbol puede ser convertido en una lista de fracciones mediante la función `tree2ratio`. En el ejemplo siguiente, convertimos un árbol en fracciones, para poder operar sobre la duración de las figuras y transformar el ritmo, por aumentación o disminución. Aquí debemos tener en cuenta que el factor de aumentación o disminución debe ser representado por un número fraccionario o un número entero. Si utilizáramos un número con decimales, convertiríamos todas las fracciones a decimal, impidiendo que la función `mktree` opere correctamente. Para el ejemplo, recurrimos, además, a la función `get-signatures`, que obtiene los indicadores de compás que `mktree` precisa para construir el árbol. En este caso, multiplicamos las fracciones por dos, cambiando las figuras y silencios por aquellas que duran el doble. Según se observa al evaluar `tree2ratio`, escribimos los silencios con fracciones negativas.



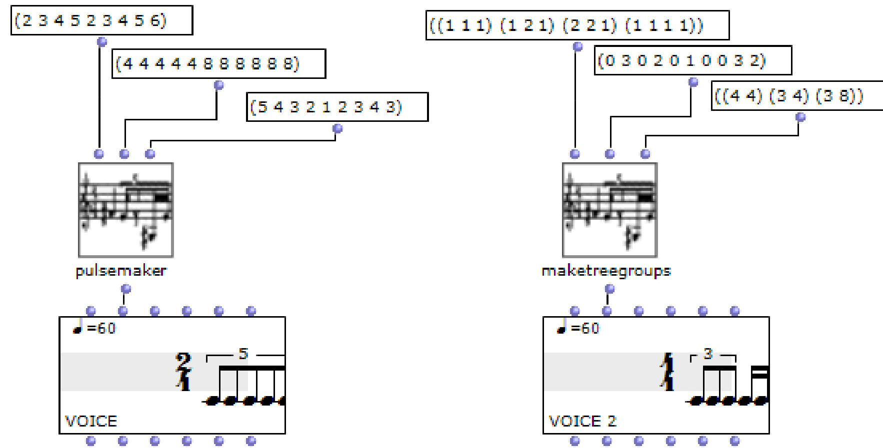
38 – árbol rítmico 1

Figura 99.

Función `tree2ratio`

Cuando deseamos generar secuencias rítmicas periódicas, podemos recurrir a la función `pulsemaker`. En el primer *inlet* especificamos numeradores de compases, y denominadores en el segundo. El tercer *inlet* recibe la cantidad de figuras que determinan el grupo. En el ejemplo que sigue, a la izquierda, el número 5 que corresponde al primer compás de 2/4, es rellenado con corcheas de quintillo,

mientras que el número 4 en un compás de 3/4 produce un cuatrillo. Se observa, además, que la indicación *EMPTY* en el editor de `voice` permite descartar el pentagrama, destacando la escritura rítmica.

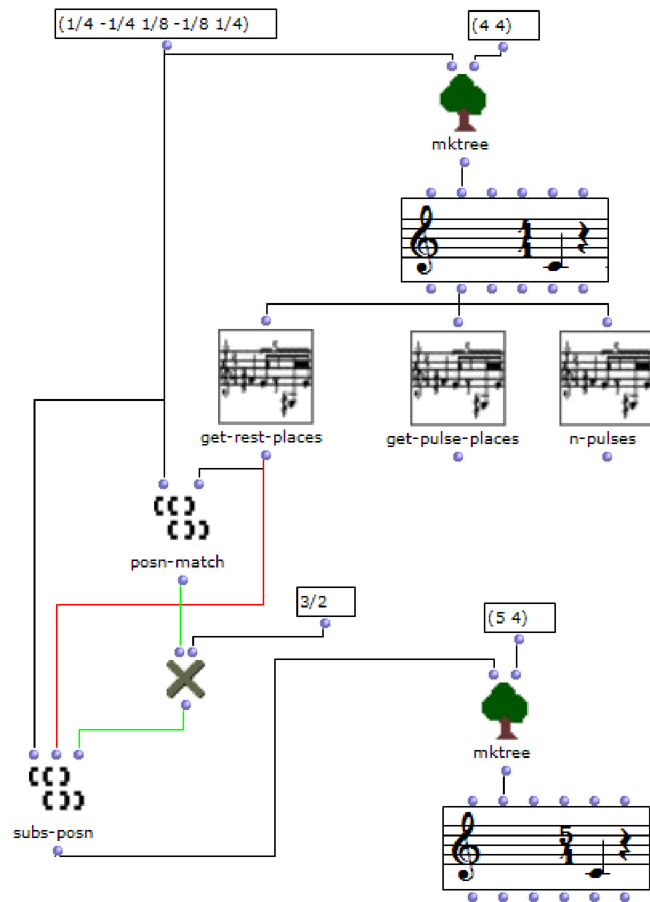


39 – árbol rítmico 2

Figura 100. Generación rítmica

A la derecha del mismo gráfico, utilizamos la función `maketreegroups`, de interés para la creación de árboles en los cuales se produce una alternancia de células rítmicas predefinidas. Cada célula es llamada de acuerdo a su posición dentro de la lista, correspondiendo a la primera de ellas la posición 0 (cero). La lista de posiciones se conecta al segundo *inlet*, mientras que el tercero recibe una lista de uno o más indicadores de compás, encerrados en sublistas.

La posición de las figuras o los silencios dentro de un árbol rítmico puede obtenerse con `get-pulse-places` y `get-rest-places`, respectivamente. La función `n-pulses`, por otra parte, nos devuelve la cantidad de ataques o pulsos que ocurren dentro del árbol. En el *patch* siguiente, generamos una secuencia mediante fracciones, que luego son convertidas al formato de un árbol rítmico con `mktree`. Luego, obtenemos una lista de posiciones de los silencios en la secuencia. Los silencios ocurren en la posición 1 y en la 3, como se aprecia al evaluar `get-rest-places`. Esas posiciones son usadas para identificar cuáles son las fracciones que representan a los silencios, mediante `posn-match`. Finalmente, las fracciones obtenidas son multiplicadas por 3/2 para aumentar su duración, y reemplazadas en la lista original de acuerdo a su ubicación, con `subs-posn`.

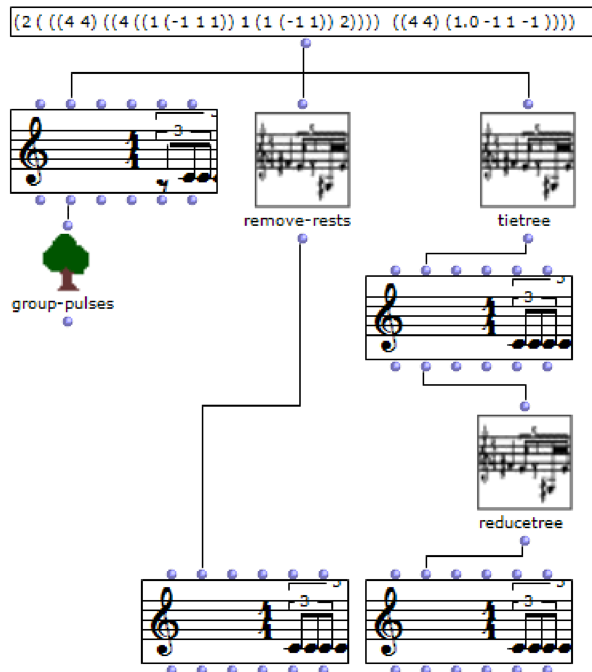


40 – árbol rítmico 3

Figura 101. Funciones rítmicas

Mediante este método podemos, entonces, alterar la duración de algunos pulsos y/o silencios, expandiendo así el material rítmico.

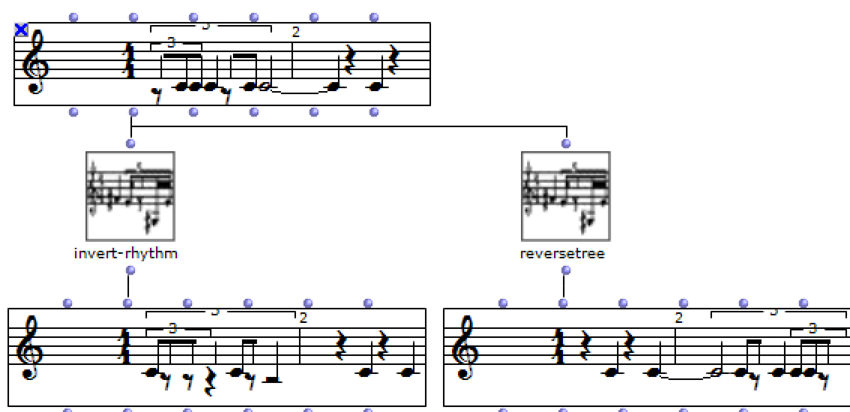
La función `remove-rests` reemplaza los silencios por figuras de igual valor. `tie-tree`, en cambio, reemplaza el silencio por una figura, pero ligándola a la nota anterior. Vale decir, prolonga los sonidos por encima de los silencios. Esta operación puede producir defectos de escritura, que se subsanan mediante `reducetree`, como se aprecia en el ejemplo que sigue.



40 – árbol rítmico 3

Figura 102. Funciones rítmicas

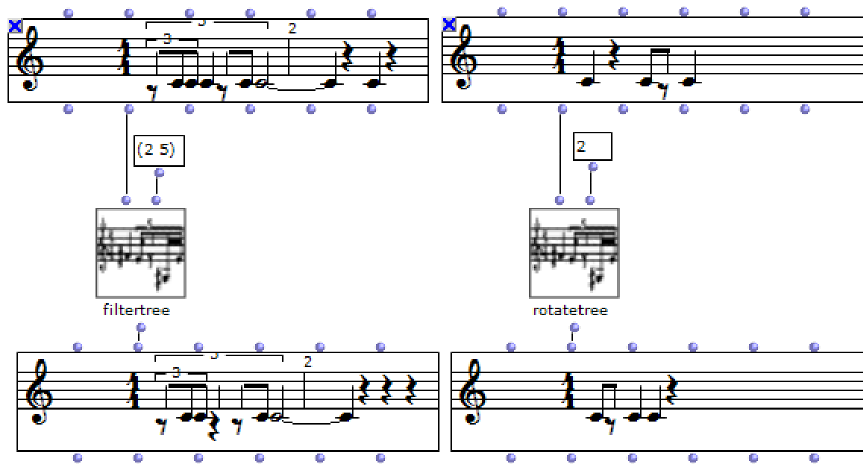
La inversión rítmica –conversión de silencios a figuras y viceversa– y la retrogradación rítmica se ven facilitadas con el empleo de `invert-rhythm` y `reversetree`, respectivamente.



41 – árbol rítmico 4

Figura 103. Funciones rítmicas

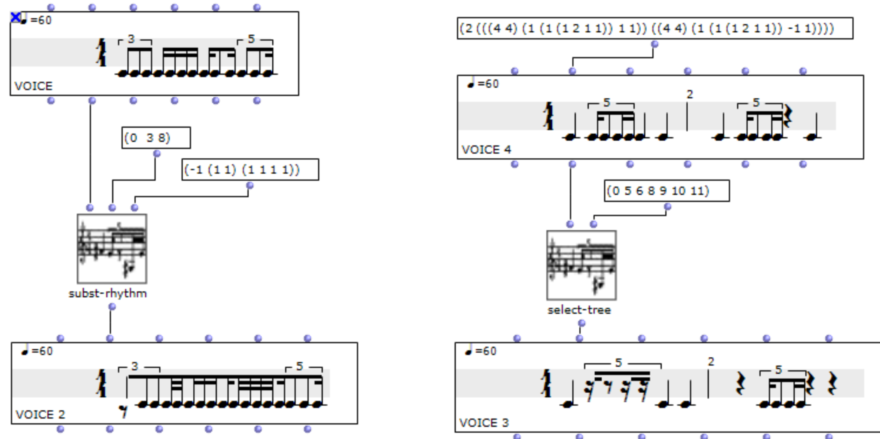
Con `filtertree`, por otra parte, reemplazamos figuras por silencios, especificando las posiciones donde esos cambios deben ocurrir, y con `rotatetree` producimos una permutación circular a partir de una posición elegida.



41 – árbol rítmico 4

Figura 104. Funciones rítmicas

Mediante `subst-rhythm` reemplazamos pulsos por otro valor o por subdivisiones rítmicas. En el ejemplo siguiente, a la izquierda, la primera corchea de tresillos (posición 0) es reemplazada por silencio (cambiando en el árbol el 1 por -1); la primera semicorchea del segundo tiempo (posición 3) es subdividida en dos fusas y la corchea del tercer tiempo (posición 8) es subdividida en 4 fusas. Nótese el modo de especificar estos cambios, y el uso de paréntesis en las subdivisiones.



42 – árbol rítmico 5

Figura 105. Funciones rítmicas

A la derecha, en cambio, elegimos algunas posiciones para quedarnos con su contenido, mientras el resto se convierte en silencios.

Abstracciones

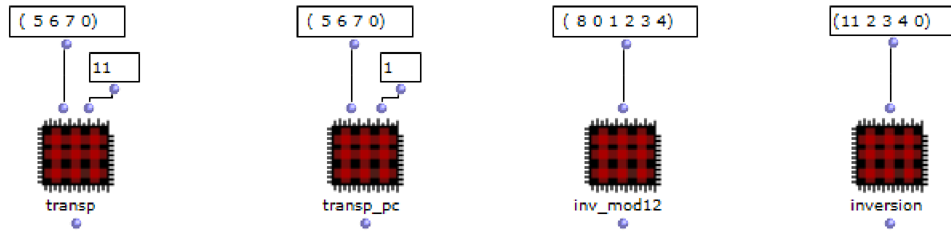
Las abstracciones son *subpatches* encapsulados en una caja. No sólo permiten ahorrar espacio en la ventana de edición al condensar múltiples objetos en uno, sino que ayudan a la generación de programas bien estructurados y a la reutilización del código realizado, en nuevos proyectos.

Utilizamos abstracciones cuando completamos una parte de un programa con una función específica, la cual posee unidad de sentido. Por ejemplo, un *subpatch* para transponer una secuencia de alturas, o invertirla. Se trata de un proceso simple, pero que seguramente lo emplearemos en más de una ocasión. Al utilizar abstracciones, incluso unas dentro de otras, la programación se desarrolla modularmente, a través de capas con distintos niveles de profundidad.

En OM contamos con dos tipos de abstracciones, aquellas que pertenecen a un único *patch*, y otras que se almacenan en el *workspace* y pueden ser compartidas por cualquier programa que hagamos. Estas últimas, se crean del mismo modo que generamos un *patch* cualquiera: yendo al menú *File/New Patch*, cambiando su nombre, editando su contenido y guardándolo en el *workspace*. Seguramente habrán notado que en la ventana de edición se presentan dos íconos con forma de flechas, arriba y a la izquierda de la ventana. Esos íconos, nos permiten generar las entradas y salidas de la abstracción, vale decir sus *inlets* y *outlets*. Para agregar una abstracción guardada en el *workspace*, a un *patch*, simplemente la arrastramos con el mouse hacia la ventana de edición.

Las abstracciones internas, aquellas que son creadas dentro de un *patch* y sólo existen dentro de éste, se pueden crear de dos modos: desde el menú contextual, haciendo *click* derecho sobre el fondo de la ventana y eligiendo *Internal.../Patch*, o bien haciendo doble *click* sobre el fondo y escribiendo "patch" en el cuadrito que aparece, como si deseáramos invocar a una función.

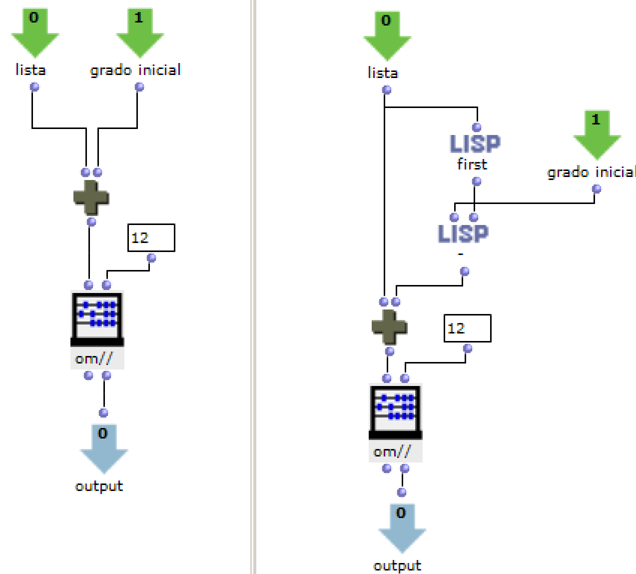
En el ejemplo siguiente, creamos cuatro abstracciones, que reconocemos como internas debido a su color rojo. La primera transpone una lista de grados cromáticos empleando un índice de transposición en semitonos; la segunda transpone pero a partir de un grado cromático dado; la tercera invierte módulo 12, o sea fijando el eje de inversión en la nota *do* y la cuarta invierte poniendo el eje sobre la primera nota de la secuencia, como se realiza habitualmente en la música tonal.



43 – abstracciones

Figura 106. Cajas de abstracciones

Al hacer doble *click* sobre la caja de la abstracción podemos acceder a su contenido. Si abrimos la primera de ellas, veremos que sumamos a la lista dato el índice de transposición, y que aplicamos luego módulo 12 para que el resultado quede siempre contenido entre 0 y 11 (ver gráfico de la izquierda).

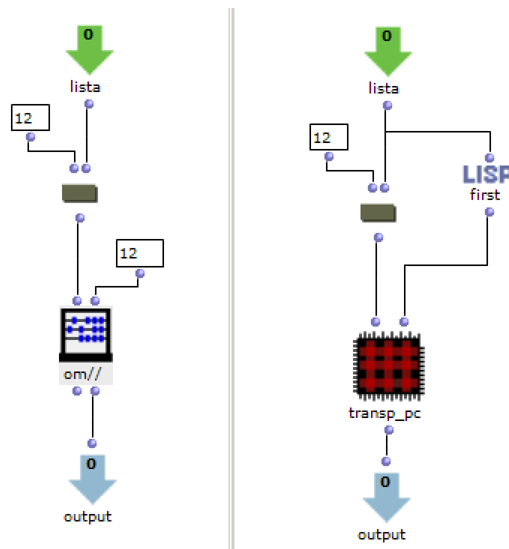


43 – abstracciones

Figura 107. Contenidos

El gráfico de la derecha muestra el contenido de la abstracción `transp_pc`. En este caso, antes de proceder a la suma, restamos el grado a partir del cual debe comenzar la transposición de la primera nota de la lista, para obtener así el índice de transposición necesario.

En las figuras que siguen, vemos el contenido de las abstracciones `inv_mod12` e `inversion`. En el primer caso, restamos 12 menos el grado cromático, y de ese modo, invertimos sobre el eje *do*. Luego, aplicamos módulo 12, dado que 12 menos 0 es 12, y que en ese caso debemos obtener 0 como resultado. A la derecha, vemos que para una inversión tradicional realizamos inversión módulo 12 y luego transponemos el resultado sobre la nota de inicio de la secuencia. Esta última tarea, la realizamos con la abstracción anterior de transposición, lo cual permite apreciar como el código se va estructurando en base a capas.



43 – abstracciones

Figura 108. Contenidos

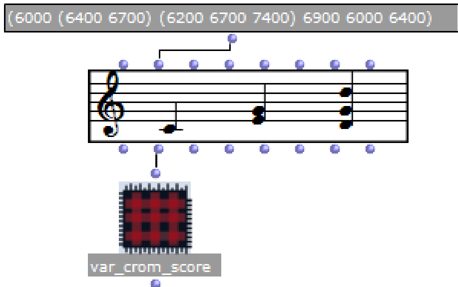
Dentro del mismo *patch* encontramos la abstracción `var_crom`, que devuelve la variedad cromática de una secuencia de grados. Para ello, eliminamos las repeticiones y calculamos la longitud de la lista resultante. Las notas ingresadas corresponden a una escala pentatónica, y obviamente, el resultado obtenido es 5.

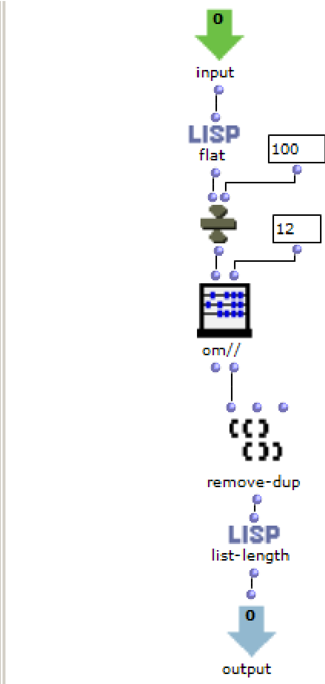
43– abstracciones

Figura 109.

Función `tree2ratio`

Pero si deseáramos tomar la información de un objeto `chord-seq`, por ejemplo, veríamos que las alturas están expresadas en *midicents*, por lo cual nuestra abstracción dejaría de funcionar. En primer lugar porque la lista podría contener sublistas, y luego porque no detectaríamos como igual a una misma nota en octavas diferentes. Las modificaciones a realizar podrían ser las siguientes:





43 – abstracciones

Figura 110. Contenidos

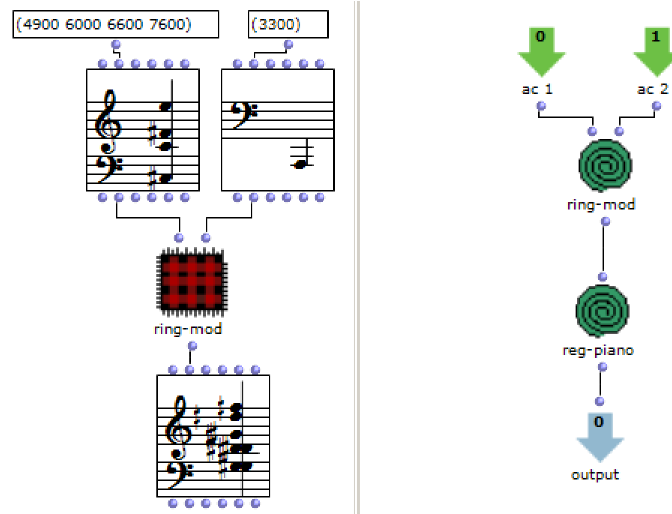
Con `flat` nos aseguramos de eliminar los paréntesis internos. Luego convertimos los *midicents* en grados cromáticos, dividiendo la lista por 100 y aplicando módulo 12. Finalmente, eliminamos las repeticiones y calculamos la longitud de la lista.

A continuación, y con el propósito de practicar tanto el encapsulamiento como las iteraciones, vamos a aplicar la técnica de convolución espectral –propia del empleo de los medios electroacústicos- al ámbito instrumental.

Supongamos que trabajamos sobre dos señales, una compleja armónica y la otra pura. El espectro del primer sonido posee tres armónicos, de 100, 200 y 300 Hz, mientras que el segundo tiene una frecuencia de 20 Hz. La convolución espectral consiste en la suma y la resta de las frecuencias de cada componente de un espectro con las del otro. En nuestro ejemplo, 100 ± 20; 200 ± 20 y 300 ± 20, o sea 80, 120, 180, 220, 280 y 320 Hz. Luego de esta operación, el sonido que resulta es inarmónico y contiene seis parciales.

La convolución espectral proviene de la época de los sintetizadores analógicos, y se realizaba con un circuito denominado "modulador en anillo", por la forma en que se distribuían sus componentes electrónicos. Mediante esta técnica era posible crear sonidos inarmónicos (de altura indefinida) utilizando sonidos armónicos modulados. A partir del empleo de los medios digitales, la convolución de los espectros se obtiene muy fácilmente, multiplicando muestra a muestra ambas formas de onda.

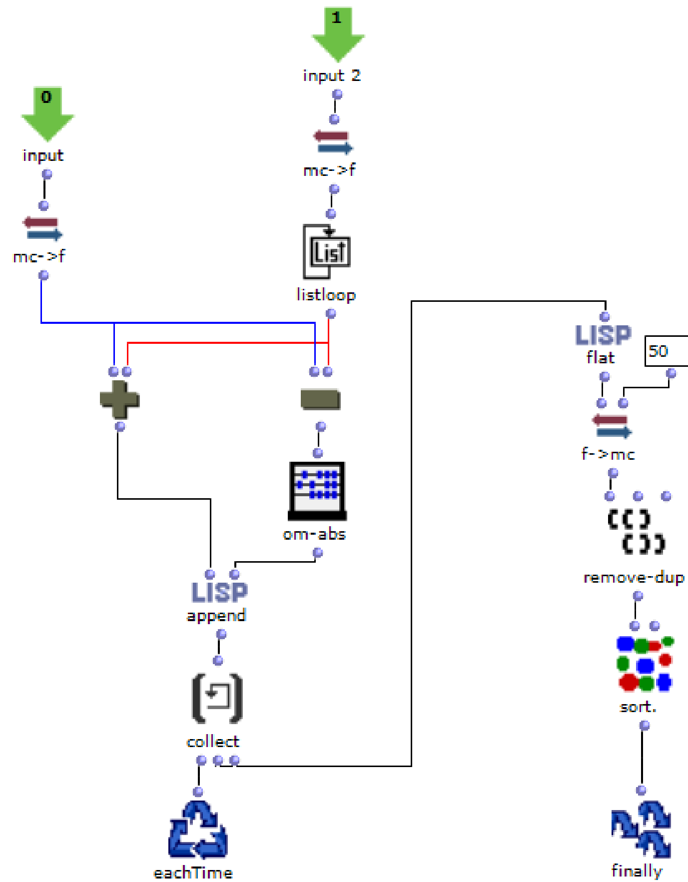
Podemos ahora suponer que un acorde podría representar a un espectro, en el cual las notas reemplazarían a las componentes. Convolucionar dos acordes sería equivalente a convolucionar dos espectros. Para llevar a cabo esta tarea, debemos convertir las notas a frecuencia y realizar la suma y resta de las componentes de un acorde con las del otro. Luego, redondear las frecuencias del resultado al semitono o cuarto de tono, o la división que elijamos, y convertirlas nuevamente en *midicents*. La figura siguiente, ilustra una abstracción que realiza una modulación en anillo instrumental, y a la derecha se ve su contenido, formado por dos objetos `omloop`. El primero calcula la modulación y el segundo elimina todas las notas MIDI que se hallan fuera del registro del piano.



43 – abstracciones

Figura 111. Convolución espectral

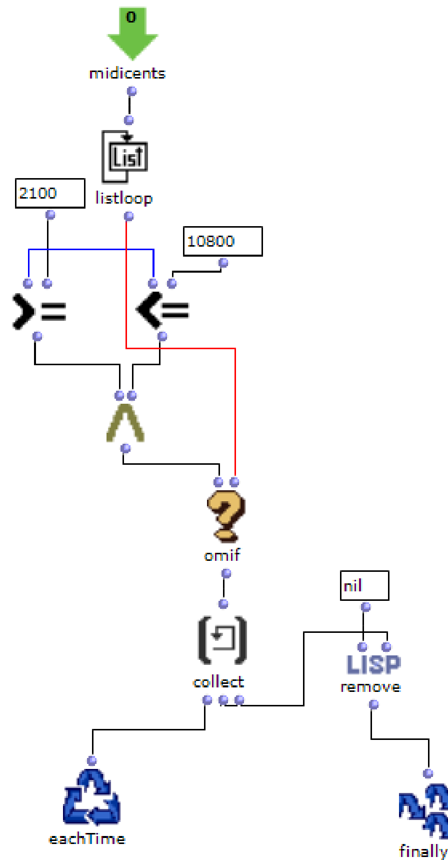
Al observar el contenido de `ring_mod` vemos la conversión de las dos listas de entrada, de *midicents* a frecuencia, utilizando la función `mc->f`. Luego, obtenemos cíclicamente las notas de la lista de la derecha con `listloop` y realizamos la suma y la resta de sus elementos con la lista de la izquierda. Dado que al restar no sabemos cuál de los números es mayor, podría darse el caso que el resultado de la resta fuera negativo. Para evitarlo, tomamos el valor absoluto (el valor positivo) empleando `om-abs`. Posteriormente, la concatenación de ambas listas con `append` -las que resultan de sumar y restar- son recolectadas con `collect`. Antes de enviar el resultado final, la lista recogida es convertida a *midicents*, se remueven las notas repetidas con `remove-dup` y se ordena de menor a mayor con `sort`.



43 – abstracciones

Figura 112. Convolución espectral

La lista resultante pasa luego al *loop* `reg-piano`. Si las notas MIDI se encuentran comprendidas entre 21 y 108 inclusive, que corresponde al rango cubierto por el registro del piano, pasan a la salida. Caso contrario, la función `omif` las reemplaza por `nil`. Una vez barrida la lista, antes de enviar el resultado por `finally`, eliminamos los `nil` con `remove`.



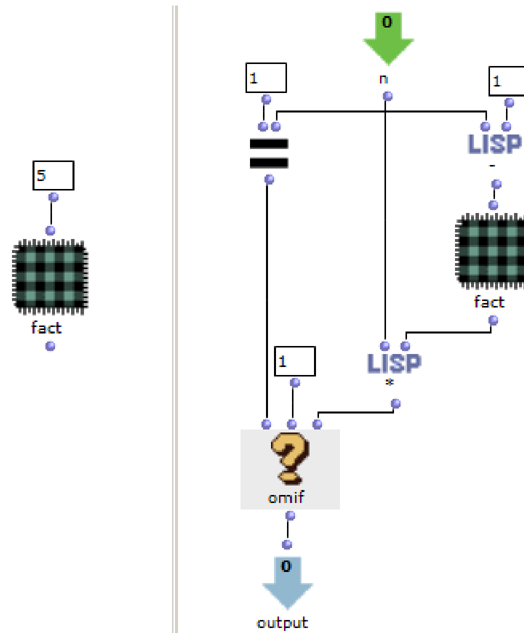
43 – abstracciones

Figura 113. Convolución espectral

Como detalle que puede resultar de interés, al hacer doble *click* sobre el *input* de una abstracción se abre un editor que permite cambiar el nombre de la entrada, y documentar con un breve texto las características de los datos esperados, o el tipo de parámetro relacionado con la función de la abstracción.

Las abstracciones desarrolladas hasta aquí han sido del tipo internas. Si duplicamos una abstracción interna y la modificamos, el cambio producido no se refleja en la copia, pues ambas son independientes. Una abstracción almacenada en el *workspace*, en cambio, refleja sus cambios en todos los *patches* que hagan uso de ella, pues todos leen la misma versión almacenada en el disco de la computadora. Para convertir una abstracción interna en una almacenada, simplemente la arrastramos del *patch* donde reside al *workspace*. Luego de ello, estará disponible para cualquier nueva aplicación que comencemos a programar.

Las abstracciones almacenadas son útiles cuando deseamos programar una función recursiva, como es el caso del cálculo de factorial de un número. Ya nos habíamos referido a esta función al tratar *LISP*. Veamos cómo luce en OM.



44 – factorial

Figura 114. Factorial

Al abrir la abstracción, vemos que su interior posee una copia de sí misma. Esto permite que se autoinvoque de forma recursiva. Si el número que ingresa es 1, la salida es 1 y el bucle finaliza. Caso contrario, se multiplica el número por el resultado de la función `fact`, a la cual se le asigna como nuevo parámetro el número dado menos 1.

Funciones *lambda*

Al tratar *LISP* introdujimos algunos conceptos relativos a las funciones anónimas, también llamadas funciones *lambda*. Según vimos, es un modo de crear funciones que sean argumentos de otras funciones.

En OM, cuando una función es argumento de otra decimos que se encuentra en modo *lambda*. Para que eso pueda ocurrir, seleccionamos la caja de la función empleada como argumento y presionamos la "l" (ele). Inmediatamente, veremos la letra griega en la esquina superior izquierda.

Como ya mencionamos, algunas funciones *LISP* requieren que alguno de sus argumentos sea otra función, como `apply`, `funcall` o `mapcar`. En el ejemplo que sigue, veremos dos maneras de invocar a la función *LISP* de suma, con `apply`. A la izquierda, poniendo a la función `+` en modo *lambda*; a la derecha, llamándola directamente por su nombre. Ambas opciones resultan válidas.

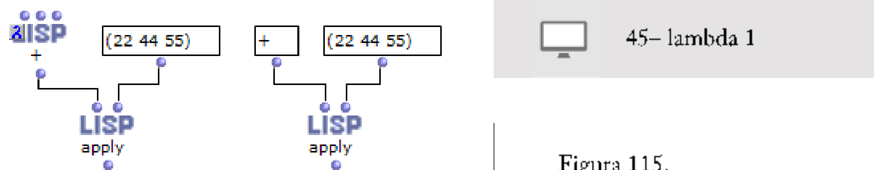
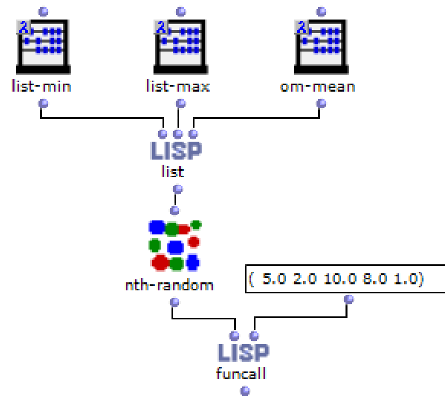


Figura 115.

Función *lambda* con `apply`

Nótese que en el caso de la izquierda, la cantidad de argumentos (ver *inlets*) de `+` debe coincidir con la cantidad de datos que se ingresan como argumento de `apply` (en nuestro caso, 3).

También podríamos incluir distintas funciones *lambda* en una lista –dada la equivalencia entre funciones y datos propia de *LISP*– y llamar a alguna de ellas desde `funcall`, de forma aleatoria, como en el siguiente *patch*.

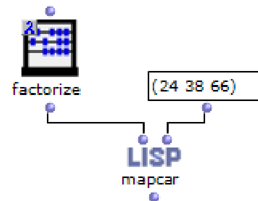


45 – lambda 1

Figura 116. Función lambda con `funcall`

A través de la evaluación sobre `funcall`, se apela al valor mínimo, al valor máximo o al promedio de una lista de números.

Respecto a `mapcar`, también la utilizamos para aplicar una función sobre cada uno de los elementos de una lista. Veamos un ejemplo empleando la función `factorize`, que descompone un número en sus factores primos. El número 24, por ejemplo, que puede ser factoreado como $2^3 + 3^1$.

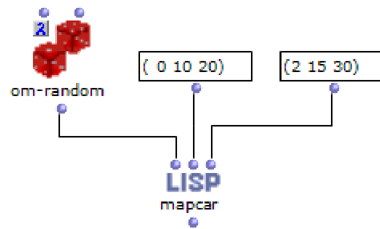


45 – lambda 1

Figura 117. Función lambda con `mapcar`

Resulta claro que sin `mapcar` no habríamos podido aplicar la función a todos los elementos de la lista, pues `factorize` admite solamente un número como argumento. Es posible ver, entonces, que `mapcar` puede reemplazar a algunos de los bucles que solemos implementar mediante `omloop`, de manera eficaz.

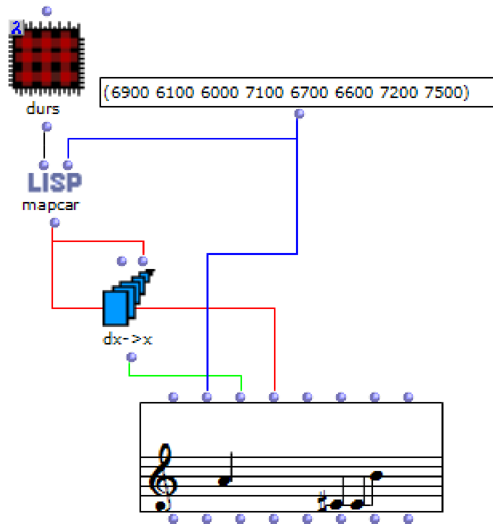
Algo similar ocurre en el *patch* que sigue, donde `mapcar` recibe dos listas de argumentos; una de valores mínimos y otra de valores máximos, que determinan el rango de los números al azar a generar. Un resultado posible es OM => `(1 11 22)`. Vemos, entonces, que una función que produce un número, puede ahora combinarse para generar una lista.

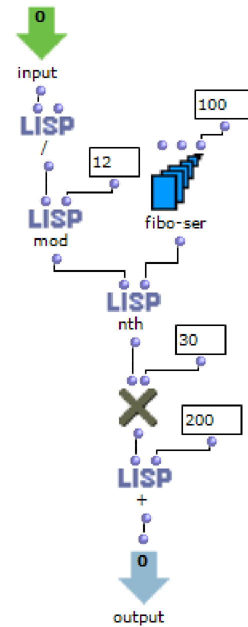


45 – lambda 1

Figura 118. Función *lambda* con dos argumentos

La función `mapcar` no sólo puede procesar primitivas de *LISP* o funciones de OM, sino también el contenido de abstracciones. En el ejemplo que sigue, creamos una abstracción para asignar una duración fija a cada grado cromático, y aplicamos ese proceso a cada elemento de la lista de *midicents*.





46 – lambda 2

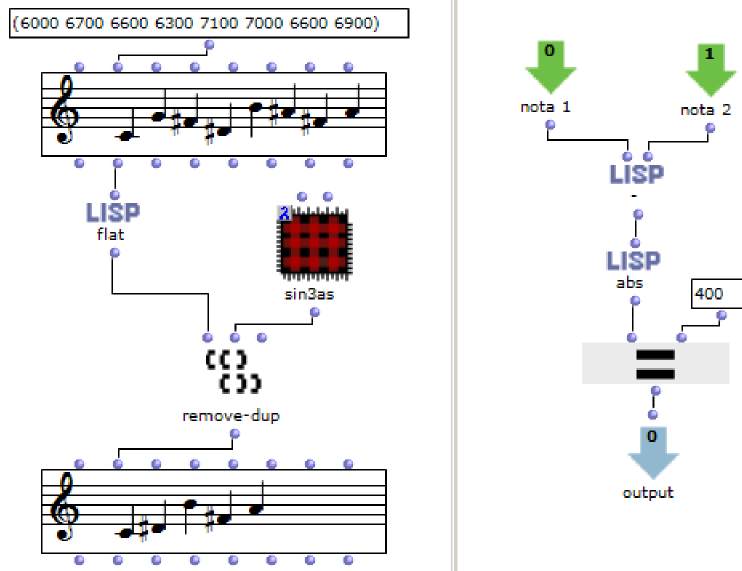
Figura 119. Abstracción en modo *lambda*

Según se observa en la abstracción, el grado cromático se emplea como índice de una lista creada con los primeros 12 términos de la serie de Fibonaci, multiplicados por 30. Para evitar que el grado 0 (*do*) tenga una duración nula, se suma a cada valor una duración de 200 ms.

Una función que admite a otra función como argumento es `remove-dups`, usualmente utilizada para eliminar elementos repetidos de una lista. La función que

ésta emplea, por defecto, para establecer el criterio de igualdad es `eq`; pero nada impide que podamos utilizar otra primitiva, o incluso una abstracción en modo *lambda*.

En el ejemplo que sigue, vamos a borrar de una secuencia algunas notas. Pero el criterio para eliminar a la primera de un par considerado no es que sean iguales, sino que formen una tercera mayor. Las notas las obtenemos de una instancia de la clase `chord-seq`, y a continuación borramos los paréntesis internos de la lista de alturas con `flat`. Luego, procedemos a eliminar las terceras mayores. Si comparamos la secuencia de entrada con la de salida, veremos que el *sol* fue borrado pues formaba una tercera mayor con el *si*; el *fa#* pues formaba el mismo intervalo con el *la#*, y ese mismo *la#* con el *fa#* siguiente.



46 – lambda 2

Figura 120. Filtro de terceras mayores

Dentro de la abstracción, restamos los *midicents* de las dos notas a comparar y hallamos el valor absoluto, pues el resultado puede dar positivo o negativo, y el signo no nos interesa, ya que simplemente expresa la dirección del intervalo. Finalmente, si el resultado es igual a 400, significa que la distancia es de tercera mayor y que se cumple la condición para que la primera nota del par analizado sea borrada. En ese caso, la abstracción devuelve `t` (verdadero), y en caso contrario `nil` (falso).

Modos de evaluación en OM

Al seleccionar un objeto, en general el último de una cadena, y presionar la tecla "v" - o al hacer *Control* + *click* sobre un *outlet*- se realiza una serie de evaluaciones internas de todos los objetos interconectados. La forma en que esas evaluaciones tienen lugar depende del modo de evaluación elegido, visible en las cajas de las clases y las funciones utilizadas. Los cuatro modos alternativos de evaluación son los que a continuación detallamos:

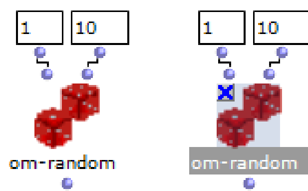
1) **modo bloqueado**: para elegir este modo seleccionamos la caja y presionamos "b". En este modo se conserva la última evaluación realizada, y los cambios que ocurran en sus parámetros, a través de los *inlets* del objeto, son ignorados.
2) **modo de evaluación única**: para elegirlo presionamos "1" (uno). Se evalúa una vez a pedido del usuario, y las evaluaciones internas requeridas por los demás objetos son ignoradas. Sólo vuelve a evaluar cuando el usuario así lo requiere.
3) **modo lambda**: según vimos, lo seleccionamos presionando "l" (ele). El objeto se emplea como argumento de una función.
4) **modo referencia**: es usado en la secuenciación de eventos en el tiempo, empleando maquetas y cajas temporales. En este libro no nos referiremos a estos usos.

Tomemos como ejemplo un objeto `om-random`. En evaluación normal arroja un nuevo número al azar cada vez que presionamos la tecla "v". Pero si está bloqueado, repite el resultado de la primera evaluación. Evaluando el *patch* 47, el de la izquierda, podríamos obtener, luego de tres evaluaciones sucesivas, lo siguiente:

```
OM => 6
OM => 10
OM => 8
```

Mientras que en el segundo caso, el de la derecha,

```
OM => 5
OM => 5
OM => 5
```

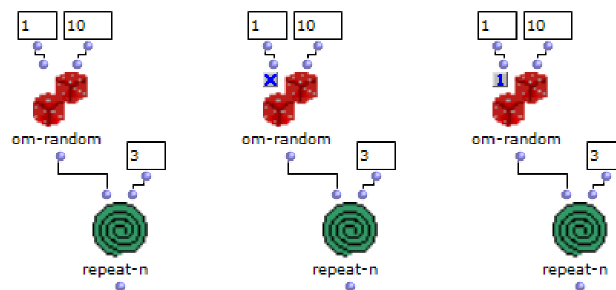


47 – evaluación

Figura 121. Modo bloqueado

Cuando se edita una clase que posee un editor, como `chord`, `chord-seq` o `voice`, la instancia se bloquea automáticamente al abrir el editor a fin de que los cambios realizados sobre el mismo no se pierdan al realizar una nueva evaluación.

Consideremos ahora el siguiente ejemplo, con el uso de `repeat-n`. Una vez que presionamos "v" sobre él, solicita al `om-random` tres evaluaciones internas y las agrupa en una lista.



47 – evaluación

Figura 122. Comparación de modos

Si presionamos "v" tres veces observamos los siguientes resultados. El *patch* de la izquierda podría generar:

```
OM => (3 7 9)
OM => (4 5 8)
OM => (4 10 8)
```

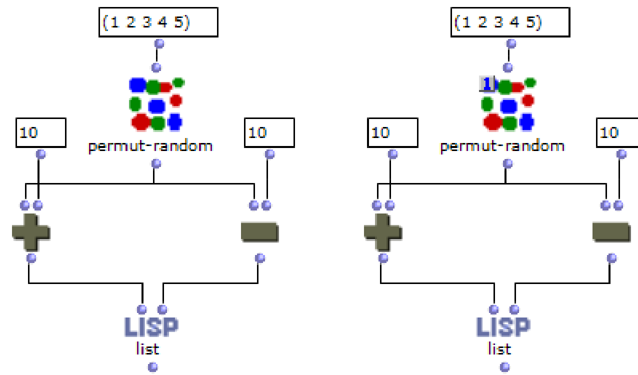
El del medio, que está bloqueado:

```
OM => (6 6 6)
OM => (6 6 6)
OM => (6 6 6)
```

Mientras que el de la derecha que está en modo de evaluación única:

```
OM => (5 5 5)
OM => (10 10 10)
OM => (2 2 2)
```

Según se aprecia, en el último caso, el bloqueo ocurre para las evaluaciones internas, pero no para las evaluaciones efectuadas por el usuario. Esto es realmente útil cuando empleamos funciones basadas en la aleatoriedad; o cuando se producen bifurcaciones a la salida, y para cada rama del procesamiento se desean mantener los mismos valores, como en el caso siguiente:



47 – evaluación

Figura 123. Modo de evaluación única

El *patch* de la izquierda está en modo normal, y cuando el usuario evalúa el objeto `list` se producen dos evaluaciones internas en `permut-random`, debido a la bifurcación a su salida. El resultado podría ser, para una permutación generada al azar (4 2 5 1 3):

```
OM => ((14 12 15 11 13) (-8 -6 -9 -5 -7))
```

Pero vemos que la resta fue realizada sobre otra permutación, la (2 4 1 5 3), que no era lo esperado. A la derecha, en cambio, al poner al objeto en modo de evaluación única, las evaluaciones internas permanecerían bloqueadas hasta una nueva intervención del usuario. La permutación (4 2 5 3 1) se mantendría constante tanto para la suma como para la resta.

```
OM => ((14 12 15 13 11) (-6 -8 -5 -7 -9))
```

Instancias, *slots* y variables globales

En ocasiones, es deseable materializar una instancia determinada de una clase, o sea crear un objeto concreto, para reutilizarlo luego. Para ello debemos presionar *Shift* + *Control* y hacer un *click* sobre un *outlet* de la caja. Si lo realizamos sobre el primer *outlet*, denominado *self*, generaremos el objeto propiamente dicho. Si lo hacemos, en cambio, sobre los *outlets* restantes obtendremos una instancia de sus datos.

Volvamos al ejemplo de la clase `chord`, visto al tratar las funciones y las clases. Materializamos la instancia haciendo *Shift* + *Control* + *click* sobre el primer *outlet*, y obtenemos la caja que la representa. Del mismo modo, obtenemos una instancia de las notas del acorde realizando la misma operación sobre el segundo *outlet* (llamado *lmidic*). La figura que sigue muestra la representación de ambas.

Figura 124.

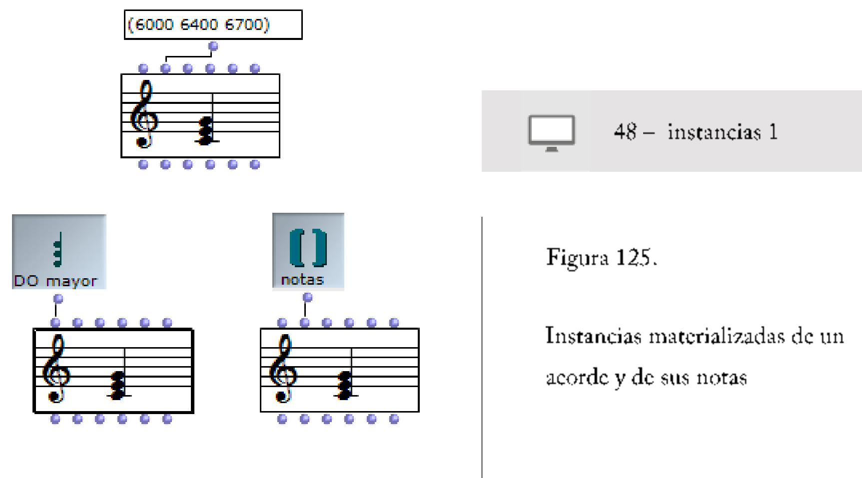
Instancias materializadas de un acorde y de sus notas

En OM, encontramos entonces dos tipos de instancias posibles a partir de una *factory box*: la instancia de clase y las instancias de sus datos, estas últimas también llamadas *list instances*.

Materializada su instancia, el acorde de *do* mayor ha cobrado entidad y podría ser reutilizado. Resulta obvio que un simple acorde mayor no presenta interés como constante de programación, pero podríamos pensar en estructuras musicales complejas que, formando parte de los resultados parciales de una obra, podrían ser reutilizadas en otros *patches*.

En la figura que sigue, vemos que la instancia de la clase `chord` materializada es conectada al primer *inlet* (*self*) de otra clase `chord`. Al evaluar esa clase, se genera automáticamente el acorde. Del mismo modo, la instancia con las notas del acorde es conectada a la entrada de notas de la clase `chord` de la derecha, y al ser evaluada

la clase toma la información de la instancia de datos. Nótese también que los nombres de las instancias pueden ser modificados haciendo doble *click* sobre ellos.



48 – instancias 1

Figura 125.

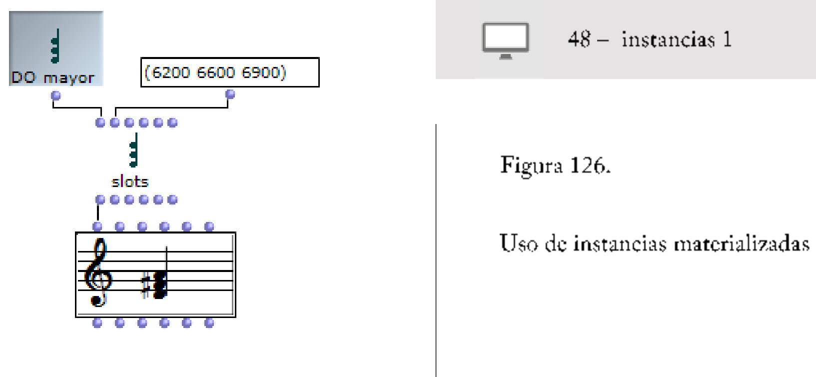
Instancias materializadas de un acorde y de sus notas

Si observamos atentamente las cajas de las instancias notaremos que sólo disponen de un *outlet* (*self*) y ningún *inlet*. La instancia del acorde ha sido generada con sus propios datos y no puede ser inicializada[37], o sea, no puede ser devuelta a su estado original. No obstante ello, podremos obtener sus parámetros o modificarlos, según veremos a continuación.

Slots

A fin de poder leer o transformar los parámetros de una instancia que fue o no materializada, es necesario crear una nueva caja a partir de su propia clase, denominada *slots box*. Para crear la caja de *slots* hacemos doble *click* sobre el fondo de la ventana de programación y escribimos el nombre de la clase. Luego, hacemos nuevamente *click* sobre el fondo, pero con la tecla *Shift* presionada. Para nuestro ejemplo escribimos *chord*, y al hacer *Shift* + *click* en el fondo de la ventana aparece la caja de *slots* que corresponde a una clase `chord`. Según se aprecia en el gráfico que sigue, al conectar la instancia *DO mayor* a la caja de *slots*, se torna factible modificar la lista de notas del objeto y convertir al acorde en *RE mayor*. Se ve, por otro lado, que la clase `chord` y la caja de *slots* de la misma clase poseen el mismo número de *inlets* y de *outlets*.

[37] Para inicializar una instancia no materializada presionamos *Shift* + "i"



48 – instancias 1

Figura 126.

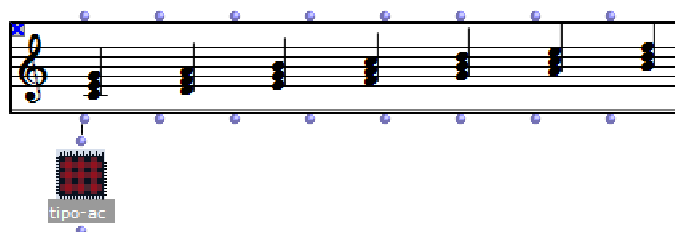
Uso de instancias materializadas

Siguiendo estos criterios, si evaluamos el segundo *outlet* de la caja de *slots* podremos obtener la lista de notas del acorde modificado.

```
OM => (6200 6600 6900)
```

La utilidad de los *slots* se hace presente cuando deseamos emplear una clase como dato de una abstracción, por ejemplo. Supongamos que generamos acordes tríadas sobre una clase `chord` y deseamos luego obtener información de esos acordes para tomar determinadas decisiones. Una posibilidad sería extraer uno por uno los parámetros desde los *outlets* de la clase. Otra, ingresar la clase a la abstracción empleando solamente *self* y luego, dentro de la abstracción, extraer los datos individuales de su caja de *slots*.

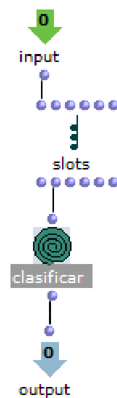
En el siguiente ejemplo, dada una serie de acordes tríadas en una clase `chord-seq` determinamos si son mayores, menores, aumentados o disminuidos. Dado que el proceso involucra solamente a las notas, podríamos obtener los valores de éstas directamente del segundo *outlet* de `chord-seq`. No obstante, para ilustrar de la forma más simple posble lo anterior, vamos a obtenerlas del primer *outlet* (*self*).



49 – instancias 2

Figura 127. Uso de *slots*

Ya dentro de la abstracción observamos la caja *slots*. De allí es de donde podríamos obtener la totalidad de los parámetros, ingresando a la abstracción con un único cable.



49 – instancias 2

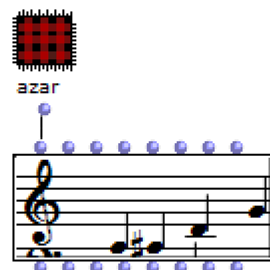
Figura 128.

Uso de *slots*. Contenido

Al evaluar la abstracción obtenemos:

```
OM => (mayor menor menor mayor mayor menor disminuido)
```

De modo contrario, podríamos pensar una abstracción que generara una sucesión de notas aleatorias, con tiempos de ataque e intensidades al azar, y que la información llegara a una clase `chord-seq` a través de un único cable, conectado a *self*.

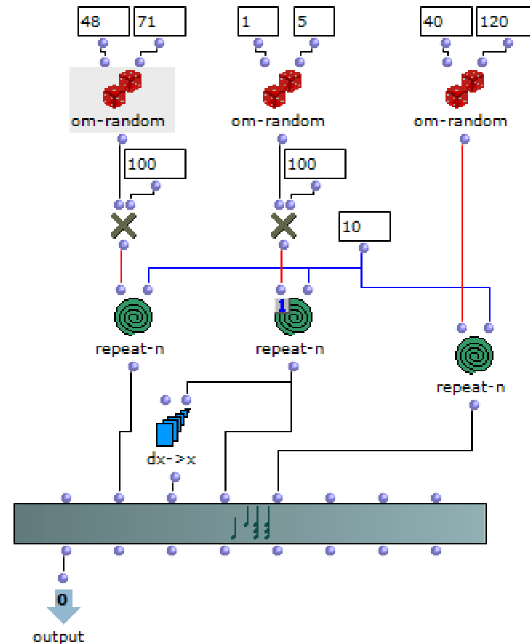


50 – instancias 3

Figura 129.

Parámetros al azar

La solución sería incluir otra clase clase `chord-seq` dentro de la abstracción, de la cual extraeríamos toda la información de su salida *self*.



50 – instancias 3

Figura 130. Parámetros al azar. Contenido

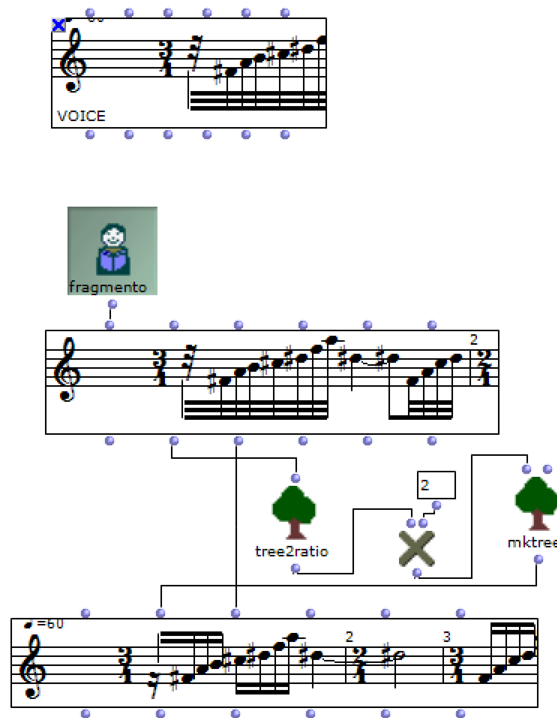
Variables

Las instancias materializadas de una clase pueden almacenarse como variables globales, para ser utilizadas desde cualquier *patch* de un mismo *workspace*. Cada vez que una variable sea evaluada remitirá a la misma instancia almacenada, y los cambios que sobre ella se realicen desde un editor impactarán en todas las aplicaciones que hagan uso de ella.

Para almacenar una instancia materializada como una variable global nos dirigimos al menú *Windows/Library*. Una vez allí, abrimos la ventana *Globals* con doble *click*, y arrastramos la instancia hasta ese lugar. Recordemos que materializamos una instancia haciendo *Shift* + *Control* + *click* sobre el primer *outlet* (*self*) del objeto. Una vez ubicada en la carpeta, podremos cambiarle el nombre, e incluirla en cualquier otro programa, simplemente arrastrándola con el mouse hasta el *patch*.

Para el ejemplo que sigue, partimos de un objeto `voice`, utilizado con anterioridad, y guardamos en la carpeta *globals* una instancia materializada suya, con el nombre

`fragmento`. Posteriormente, lo recuperamos arrastrándolo hasta nuestro *patch*. Una vez allí, realizamos una aumentación rítmica.



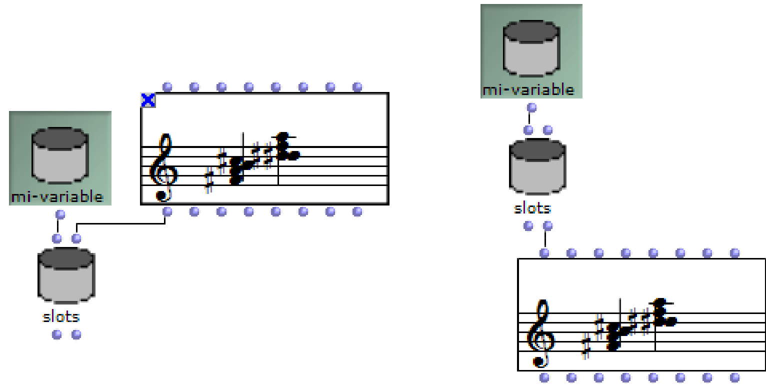
51 – variables 1

Figura 131. Utilización de una variable global

De este modo, reutilizamos un material preexistente, como si se tratara de un elemento temático.

Otro modo de crear variables globales es utilizando expresamente la clase `store`. Para ello, nos dirigimos nuevamente a *Windows/Library*, y una vez allí abrimos la ventana *Globals* con doble *click*. En el menú de esa ventana elegimos *File/New variable*, y a la caja creada le cambiamos el nombre; en nuestro caso la llamaremos `mi-variable`. Después de arrastrarla hasta nuestro *patch*, hacemos doble *click* sobre el fondo de la ventana y escribimos `store` (como si quisiéramos invocar a la clase), pero luego, en lugar de hacer *click* afuera del editor para crearla, hacemos *Shift* + *click*, que según ya vimos, sirve para generar *slots*.

El *patch* siguiente muestra cómo almacenar información (en este caso una secuencia de acordes) y cómo recuperarla, utilizando el método descripto.



52 – variables 2

Figura 132. Uso de la clase `store`

Mensajes y secuencias MIDI

A fin de integrar el contenido de un archivo MIDI a un *patch*, empleamos la clase `midifile`, a la cual se accede desde el menú *Classes/Midi*. Para elegir un archivo almacenado en la computadora disponemos de tres opciones:

a) Desde el explorador de archivos del sistema, arrastrando el ícono o el nombre del archivo a la ventana del *patch*. En este caso, el objeto `midifile` se crea automáticamente.

b) Colocando el objeto de la clase `midifile` en el *patch* y evaluándolo. Inmediatamente, el explorador del sistema se abre para que podamos buscar el archivo deseado. Una vez realizado esto es importante bloquear el objeto `midifile` –seleccionándolo y presionando la tecla *b*– para evitar que en futuras evaluaciones el programa intente buscar un nuevo archivo.

c) Conectando el objeto `infile` a `midifile`. En el *inlet* de `infile` deberemos especificar el nombre del archivo buscado, entre comillas. A fin de que OM lo encuentre, el archivo deberá estar guardado en la carpeta *in-file*, que se encuentra dentro del directorio de nuestro *workspace*.

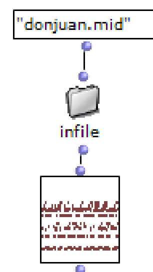


Figura 133.

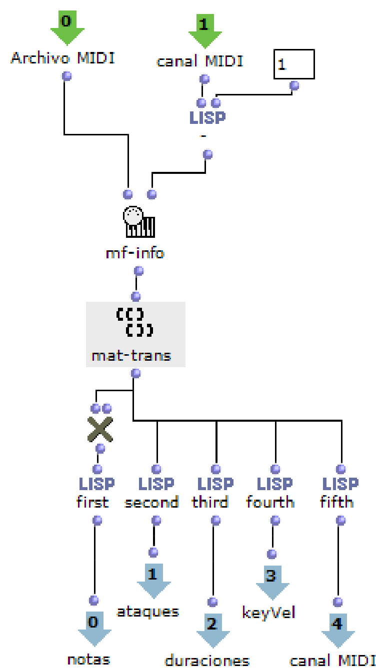
Carga de archivo MIDI

Una vez localizado el archivo, es posible ver su contenido en formato "*piano-roll*" haciendo doble *click* sobre la caja de `midifile`.

La extracción de las notas almacenados en un archivo *.mid* puede realizarse mediante `mf-info`. Esta función devuelve la información de cada evento en sublistas, que especifican el número de nota MIDI, el tiempo de inicio del evento (en milisegundos), la duración, su *key velocity* (intensidad del sonido) y el canal MIDI asignado.

La abstracción que se ilustra a continuación permite extraer la información de un canal particular de un archivo MIDI. Con el propósito de reordenar los datos, de manera tal que queden agrupados por tipo (números de nota por un lado, tiempos de ataque por otro, duraciones, etc.) utilizamos la función `mat-trans`, que se encuentra en *Functions/Basic Tools/List Processing*. Esta función transforma una lista agrupando el primer elemento de todas las sublistas, luego el segundo, y así sucesivamente. Si consideramos que las sublistas de la lista dada son las filas de una matriz, `mat-trans` nos devuelve las columnas:

```
> (mat-trans '((a1 a2 a3) (b1 b2 b3) (c1 c2 c3))
((a1 b1 c1) (a2 b2 c2) (a3 b3 c))
```

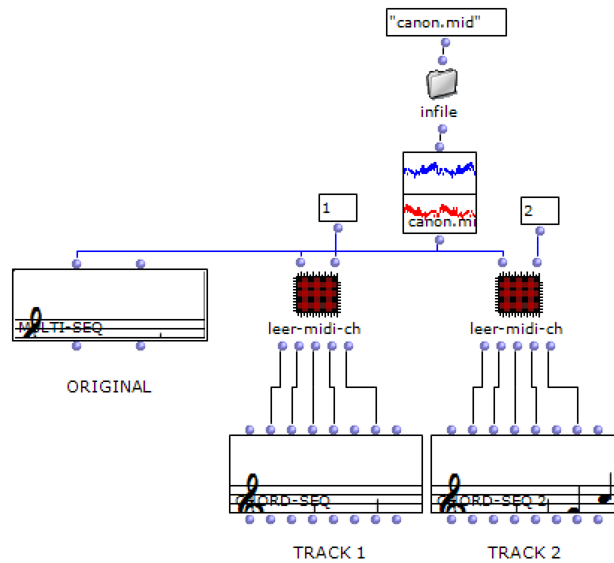


53 – midi 1

Figura 134.

Abstracción `leer-midi-ch`

En el *patch* 53 utilizamos la abstracción creada para extraer las pistas de una secuencia MIDI multipista (MIDI *format* 1). Cada pista extraída es conectada a una clase `chord-seq` para visualizar su contenido. Nótese que es posible conectar directamente al objeto `midifile` con la clase `chord-seq`, utilizando su primer *inlet* (*self*), o con `multi-chord-seq`. En el primer caso veremos todas las pistas en un solo pentagrama, mientras que la segunda opción nos permitirá visualizar las pistas separadas.

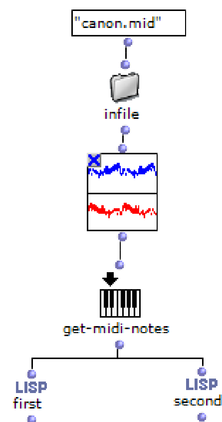


53 – midi 1

Figura 135. Extracción de pistas

El haber podido separar los parámetros de una nota (número de nota MIDI, tiempo de ataque, duración, *key velocity* y canal MIDI) nos permitirá luego procesar esa información, si deseamos realizar transformaciones en la secuencia MIDI.

Pero veamos antes otro modo de obtener esa información, utilizando la función `get-midinotes`. El objeto devuelve una lista cuyo primer nivel de sublistas corresponde a cada una de las pistas de la secuencia. Dentro de esas sublistas hallamos los parámetros antes detallados. En el ejemplo siguiente, evaluando `first` y `second` veremos en el *listener* las sublistas de parámetros de notas correspondientes a la primera y segunda pistas de la secuencia.



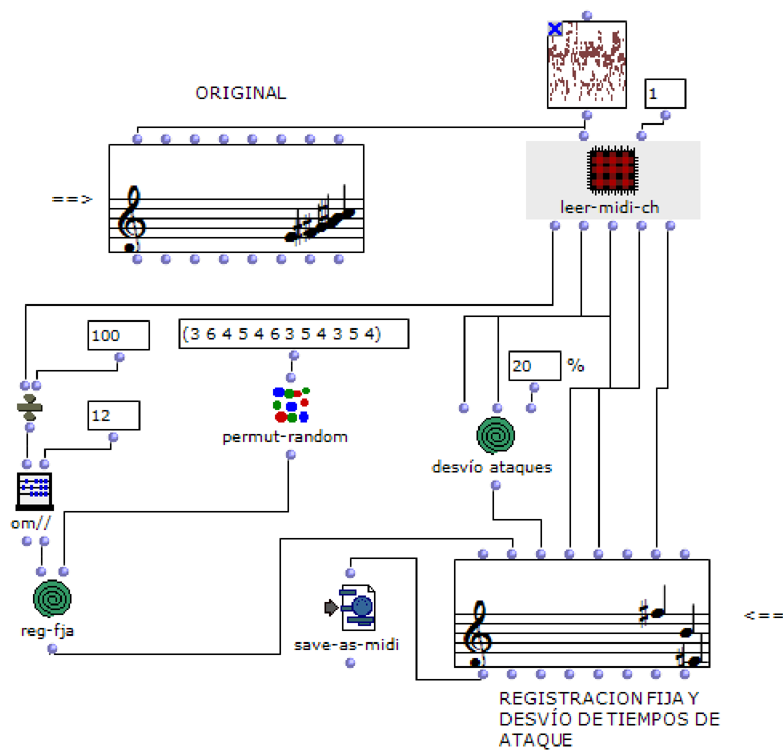
54 – midi 2

Figura 136.

La función `get-midi-notes`

En el *patch* 55 partimos de la lectura de un archivo MIDI[38] al cual le modificamos sus notas aplicando registración fija, y los tiempos de ataque, asignándoles un porcentaje de desvío. Los números de octava establecidos para cada grado cromático se obtienen de una lista de 12 números, entre 3 y 6 (recordemos que en MIDI la octava central es 5).

Por cada evaluación de la clase `chord-seq`, ubicada en la de la parte inferior del *patch*, la lista de octavas es permutada aleatoriamente con `permut-random`. Dado que las notas extraídas por `mf-info` son devueltas en *midicents*, procedemos a dividirlas por 100 para restaurarlas a su versión original. Luego, aplicando módulo 12 con la función `om//` las convertimos a números entre 0 y 11, que representan a los grados cromáticos.



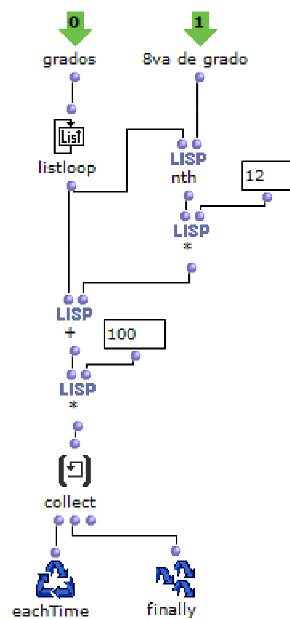
55 – midi 3

Figura 137. Transformación de una secuencia

[38] Los archivos utilizados para estos ejemplos son el tango Don Juan, de Ernesto Ponzio, y un canon a dos voces por movimiento retrógrado de la Ofrenda Musical de J. S. Bach http://www.tangoargentino.com/miditeca/

http://www.jsbach.net/midi/midi_musicaloffering.html

El *subpatch* `omloop` de registración fija se muestra a continuación. Según se aprecia en el gráfico, los grados cromáticos se extraen de la lista con `listloop` y se les suma la octava correspondiente multiplicada por 12. Esto convierte al grado en nota MIDI. La asociación entre grado y octava se resuelve mediante `nth`, que devuelve el enésimo elemento de la lista de octavas. Al grado 0 (*do*) le corresponde el primer elemento de la lista de octavas, al grado 1 (*do#*) el segundo, y así sucesivamente. La multiplicación por 100 que sigue, transforma a la nota MIDI en *midicents*. Por último, los datos son recolectados para ser devueltos al finalizar el bucle a través de `finally`.



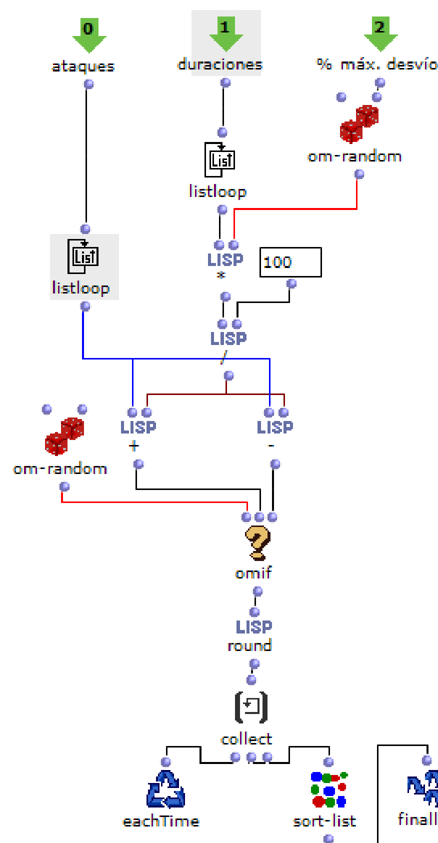
55 – midi 3

Figura 138.

Registración fija

En el caso particular de los tiempos de ataque, el porcentaje de desviación se establece aleatoriamente con `om-random`. Luego de calcular el porcentaje de la duración de cada evento, se modifica el tiempo de ataque sumándole o restándole el valor obtenido. La operación a aplicar, suma o resta, también se determina de forma aleatoria, utilizando el condicional `omif`. Finalmente, la nueva lista de tiempos de ataque de las notas es reordenada de menor a mayor, dado que al haber aplicado sumas y restas podría no quedar en orden creciente.

Los resultados que surgen de la ejecución del *patch* pueden ser almacenados en un nuevo archivo MIDI. Para ello, basta con evaluar el objeto `save-as-midi`.



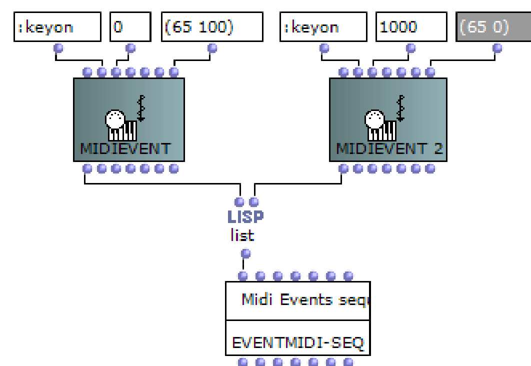
55 – midi 3

Figura 139.

Desvío de ataques

Hasta aquí, los objetos utilizados han extraído información sobre las notas de una secuencia y la han devuelto en un formato propio, distinto al de los mensajes definidos en el protocolo MIDI. Los mensajes MIDI están formados por un comando y uno o dos parámetros asociados. Un mensaje MIDI recibido, de *Note On* por ejemplo, establece simplemente que una nota ha de ejecutarse y a través de qué canal (comando), de qué nota se trata y, cuál es su intensidad (parámetros). El instante en que se recibe el mensaje es el tiempo de inicio de la nota, y su duración está determinada entre ese instante y el momento en que se recibe la orden de detener la ejecución de la nota (mensaje de *Note Off*). Del mismo modo, encontramos otros mensajes relacionados con la asignación de valores de controladores (*Continuous controller*, tales como el volumen o el paneo), con el cambio de instrumento (*Patch Change*), o con la modificación del *aftertouch* o del *pitch bend*.

La clase de OM que representa a un evento MIDI es `midievent`. Los *slots* de `midievent` son *self* (el objeto mismo), *type* (tipo de evento), *date* (instante en el que se envía), *track* (pista MIDI), *port* (puerto MIDI, usualmente 0), *channel* (canal MIDI en el que se transmite) y *fields* (parámetros del mensaje). En el ejemplo siguiente se crean dos eventos, uno de *Note On* y el otro de *Note Off* alternativo[39]. Ambos mensaje son puestos en una lista y enviados al objeto `eventmidi-seq`, capaz de construir la secuencia MIDI. Seleccionando este último objeto y presionando la barra espaciadora, es posible ejecutar los mensajes y escuchar la nota *fa* de la octava central (nota MIDI 65) con un *key velocity* igual a 100 durante 1.000 milisegundos.



56 – midi 4

Figura 140. La clase `midievent`

[39] El mensaje de *Note Off* alternativo cumple la misma función que el de *Note Off* normal y se construye asignándole una intensidad (*key velocity*) igual a 0 al mensaje de *Note On*. En la comunicación MIDI el primer byte representa al comando y luego siguen los bytes de datos. Si en el siguiente mensaje el comando es el mismo que el del mensaje anterior, no es necesario reenviarlo. Esa técnica se denomina *running status* y disminuye la cantidad de información a transmitir. El mensaje alternativo de *Note Off* posee el mismo comando que el de *Note On*, por lo cual resulta más efectivo.

Los tipos de eventos más usuales son los que se muestran en la tabla que sigue[40].

Descripción	**Tipo de evento**
KeyOn	`:keyon`
KeyOff	`:keyoff`
CtrlChange	`:ctrlchange`
ProgChange	`:progchange`
ChanPress	`:chanpress`
KeyPress	`:keypress`
PitchBend	`:pitchbend`
Lyric	`:lyric`
Tempo	`:tempo`
TimeSign	`:timesign`
KeySign	`:keysign`
InstrName	`:instrname`
SeqName	`:seqname`
Copyright	`:copyright`

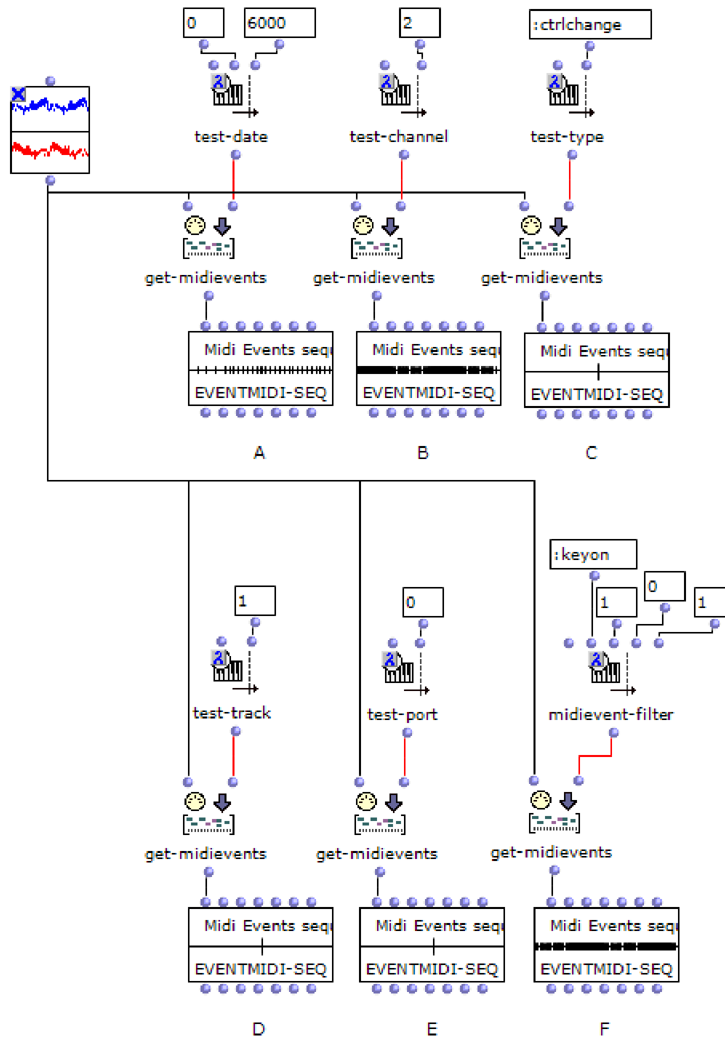
Filtros de eventos

Los eventos MIDI no sólo pueden ser creados, sino también obtenidos de una secuencia contenida en un archivo o de objetos tales como `note`, `chord`, `chord-seq`, `voice` o `eventmidi-seq`, entre otros. Mediante la función `get-midievents` podemos extraemos esos eventos, incluso aplicando filtros, lo cual resulta útil para modificar los datos obtenidos y generar así nuevas secuencias.

En el *patch* 57, representado en la figura que sigue, vemos una secuencia MIDI conectada a varios objetos `get-midievents`. En cada rama del *patch* aplicamos un filtro distinto. El primero (`test-date`, en *A)* extrae todos los eventos cuyos tiempos de aparición se encuentran comprendidos entre 0 y 6000 ms, vale decir, todo lo que

[40] La totalidad de los tipos de evento reconocidos se encuentran en la variable global `*midi-event-types*` de OM.

ocurre en los seis primeros segundos de la secuencia. En la parte *B*, en cambio, extraemos la totalidad de los eventos del canal 2 con `test-channel`.



57 – midi 5

Figura 141. Filtros de eventos

En *C*, obtenemos los eventos relacionados con controladores. Si evaluamos la segunda salida de `eventmidi-seq`, presionando la tecla *Control* y haciendo *click* sobre el círculo, veremos impreso en el *listener* los tipos de eventos que resultaron del filtrado:

```
OM => (:ctrlchange :ctrlchange :ctrlchange :ctrlchange)
```

Esto expresa que se encontraron 4 eventos del tipo *Control Change*. Al evaluar la tercera (*date*) veremos en qué momento ocurrieron. En nuestro ejemplo, los cuatro al inicio de la secuencia, en el milisegundo 0.

```
OM => (0 0 0 0)
```

Al evaluar la anteúltima salida (*channel*) notaremos los canales MIDI por los cuales se transmitieron los mensajes. Los dos primeros por el canal 2 y los restantes por el 1.

```
OM => (2 2 1 1)
```

Al evaluar la última salida (*fields*) veremos los datos de los cuatro eventos.

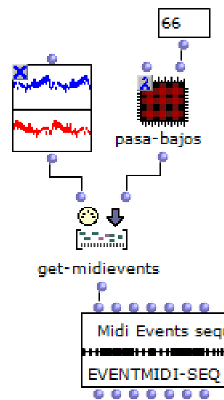
```
OM => ((10 32) (7 100) (10 96) (7 100))
```

Lo cual muestra que se envió por el canal 2 un mensaje para que el controlador 10 (paneo) recibiera el valor 32, luego para que el controlador 7 (volumen) recibiera el valor 100. Posteriormente, por el canal 1 (según se vio al evaluar *channel*) otro mensaje de paneo con valor 96, y otro de volumen con valor 100.

Del mismo modo, en los ejemplos *D* y *E* filtramos los eventos que no pertenecen a la pista y a al puerto MIDI especificados, respectivamente, empleando los filtros `test-track` y `test-port`. Y finalmente, en el ejemplo *F* utilizamos el filtro `midievent-filter`, de uso más general, especificando el tipo de evento MIDI a obtener, en qué pista, en qué puerto y el canal de pertenencia.

En todos los casos, debe tenerse en cuenta que las funciones que actúan como filtros de `get-midievents` deben encontrarse en modo *lambda*, dado que son funciones que actúan como parámetros.

Estas funciones predefinidas de OM no son las únicas factibles de ser usadas. Podemos crear nuestras propias funciones a fin de establecer criterios propios de selección de eventos. En el *patch* que sigue, usamos una abstracción en modo *lambda* para elegir todos los *Note On* y *Note Off* cuyo número de nota MIDI sea menor al especificado en la entrada derecha del objeto.

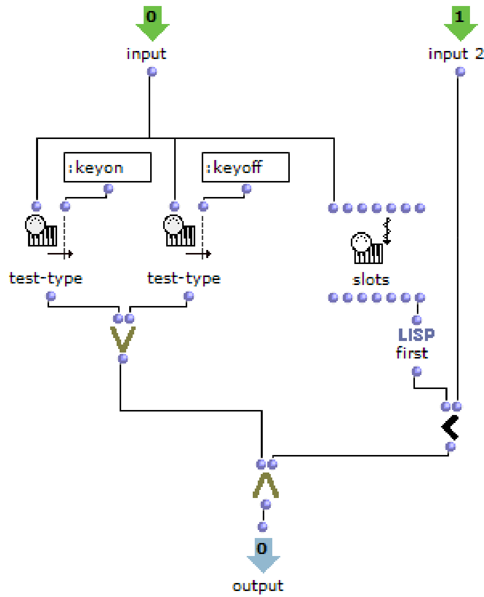


58 – midi 6

Figura 142.

Filtro en modo *lambda*

La programación de la abstracción se muestra a continuación. El evento ingresado debe ser del tipo *Note On* o *Note Off*, según lo requieren los filtros `test-type`. Nótese el objeto `omor` que los vincula. Por otra parte, el número de nota se obtiene a partir de una instancia creada de la clase `midievent`, tomando el dato del último *slot*. Esa nota se compara con la especificada y si es menor, el objeto `om<` devuelve verdadero. Si la salida del `omor` y la del `om<` son ambas verdaderas, el objeto `omand` del final devuelve también verdadero, lo cual indica que el evento cumple con las condiciones requeridas para su aceptación.



58 – midi 6

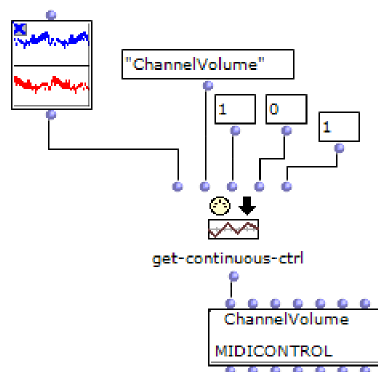
Figura 143. Contenido del filtro

Otros tipos de funciones que nos permiten extraer información MIDI de un archivo o una secuencia son `get-continuous-ctrl` (información de controladores

MIDI), `get-tempomap` (cambios de tempo y de compás) y `get-mf-lyrics` (texto de una canción).

La función `get-continuous-ctrl` se emplea, según su nombre lo indica, para obtener información de los controladores MIDI que actúan en una secuencia determinada. Los parámetros requeridos son *self* (secuencia o archivo MIDI), *ctrlname* (tipo de controlador), *ref* (número de pista), *port* (puerto MIDI) y *channel* (canal MIDI). El tipo de controlador se especifica a través de una cadena de caracteres. De los controladores más usuales, las cadenas que los representan son: "*KeyPress*", "*ChanPress*", "*PitchBend*", "*PitchWheel*", "*BankSelect*", "*ModulationWheel*", "*BreathController*", "*FootController*", "*ChannelVolume*", "*Balance*", "*Pan*", "*ExpressionController*", "*DamperPedal*", "*Portamento*", "*SoftPedal*". Una lista exhaustiva puede obtenerse observando el código fuente de la función `get-continuous-ctrl`. Para ello, seleccionamos el objeto y presionamos la tecla "e". Al elegir *method get-continuous-ctrl* del cuadro de diálogo que se despliega, accedemos al código *LISP*.

La información obtenida por `get-continuous-ctrl` puede ser visualizada a través de la clase `midicontrol`. Una vez ubicado el objeto en la ventana del *patch*, al hacer doble *click* sobre el ícono, se abre un editor que nos permite observar los valores del controlador requerido. En el ejemplo siguiente, podemos apreciar su uso. Según se aprecia en el editor, al abrir el *patch*, la secuencia analizada muestra un único valor de volumen, que es especificado al inicio.

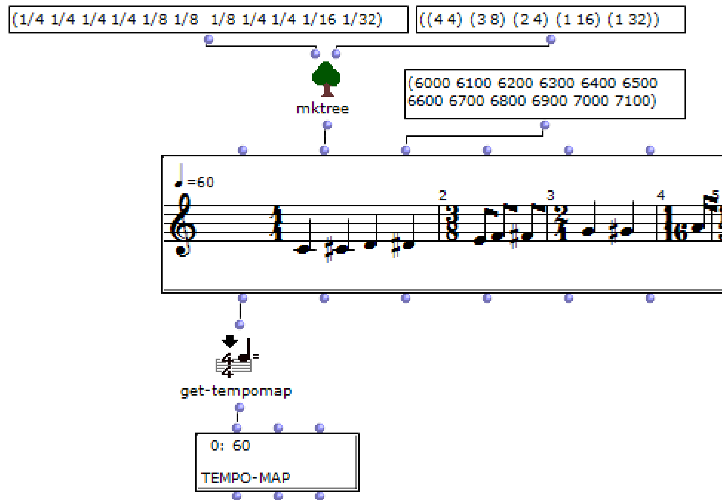


59 – midi 7

Figura 144.

Controladores

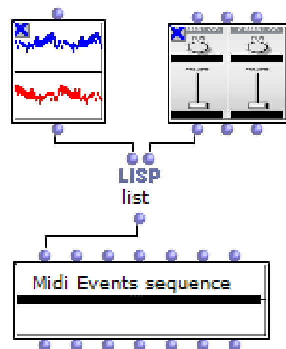
Del mismo modo, podemos extraer los *tempi* y los cambios de compás de una secuencia con la función `get-tempomap`. En combinación con la clase `tempo-map`, podemos visualizar los instantes en los que se producen los cambios de indicación metronómica, y evaluando el tercer *outlet* (*timesign-evts*) podemos observar los cambios de compás y cuándo ocurren.



60 – midi 8

Figura 145. Función `get-tempomap`

En el ejemplo del *patch* 61 hacemos uso de la clase `midi-mix-console` para modificar una secuencia. Haciendo doble *click* sobre el objeto se despliega la ventana del mezclador, y allí podremos modificar el instrumento, el "paneo" (lateralización de las fuentes sonoras), la afinación (en cents) y los valores de cualquier controlador disponible en el instrumento MIDI.



61 – midi 9

Figura 146.

`midi-mix-console`

Si escuchamos el ejemplo, habiendo seleccionado `eventmidi-seq` y presionando la barra espaciadora, notaremos que se cambiaron los instrumentos, que una de las voces fue desafinada aproximadamente un cuarto de tono y que ambas pistas se lateralizaron a izquierda y derecha, respectivamente. Si modificamos los parámetros del mezclador, será necesario evaluar nuevamente `eventmidi-seq` para que los cambios se hagan efectivos.

Código *LISP* en OM

La interfaz principal con el lenguaje *LISP*, que subyace en OM, es la ventana del *OM Listener*. Normalmente leemos allí el resultado de nuestras operaciones sobre los *patches*, pero también podemos escribir código habilitando esta función desde el menú *Preferences/General*, y tildando la opción *Enable Listener input*.

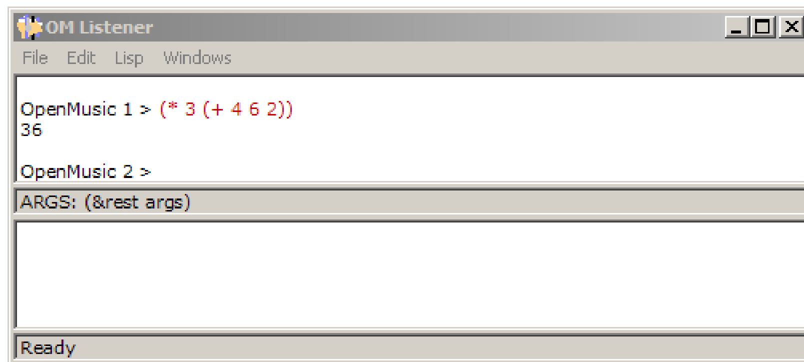
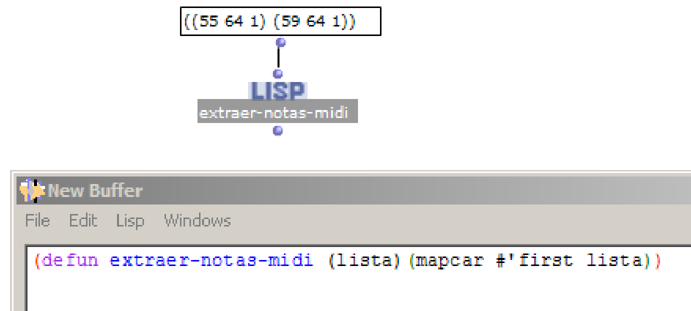


Figura 147. Ventana del *listener*

El menú *LISP* del *listener* contiene opciones que pueden resultar útiles. Por ejemplo *Abort*, que permite finalizar la ejecución de un proceso, particularmente cuando ocurren errores de programación, como bucles infinitos. O *Find Definition*, que abre una ventana para examinar el código fuente de OM, al escribir el nombre de una clase o función y luego elegir esta opción. Y también *Load File…*, que ofrece la posibilidad de cargar un archivo *.LISP* con funciones e integrarlo al entorno de programación.

Pero una de las herramientas más empleadas para incorporar código *LISP* en OM es el *Editor LISP*. Accedemos a él a través del menú *Windows/LISP Editor*, y al observarlo notamos que se trata de un simple editor de texto, con herramientas de copiado y pegado, capaz de interpretar código *LISP*. Desde el menú *LISP* observamos la posibilidad de evaluar todo el código o sólo una parte, hallar una definición –al igual que en el *listener*– o cargar un archivo *.LISP* preexistente.

En el ejemplo que sigue, copiamos una función que tratamos al estudiar *LISP*, que extrae las notas MIDI de ternas Nota MIDI - *Key Vel* - Canal MIDI almacenadas en sublistas.

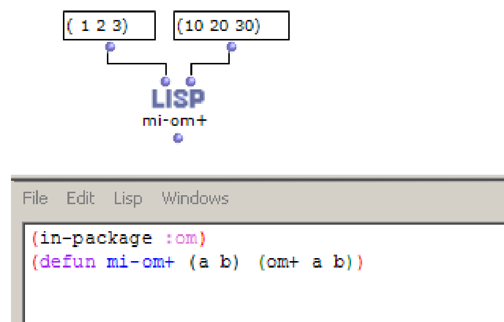


62 – código *LISP* Figura 148. Editor *LISP*

Luego de escribir la función la evaluamos desde el menú, y posteriormente creamos un *patch* para probarla. El resultado de evaluar la caja de la función arroja el siguiente resultado:

```
OM => (55 59)
```

Si deseamos incorporar nuestras funciones a las de OM, y hacer uso de ellas dentro de nuestro código, debemos agregar la sentencia `(in-package :om)` al principio. En el ejemplo que sigue la vemos, y luego el uso de la función `om+` dentro de la nuestra.



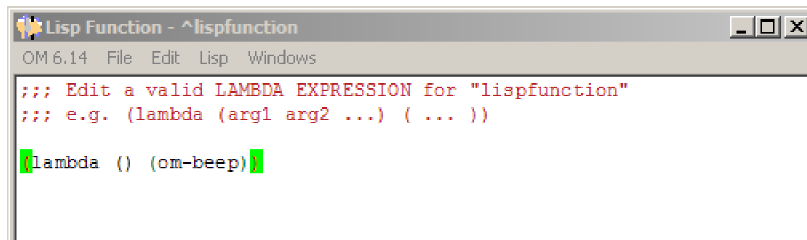
62 – código *LISP* Figura 149. `in-package :om`

A fin de evitar la carga manual de los archivos *.LISP* cada vez que deseamos utilizar las funciones programadas en ellos, podemos almacenarlos en el *Workspace*, en la carpeta denominada *user*. Cada vez que iniciemos el espacio de trabajo los archivos con sus funciones se cargarán automáticamente.

Otra forma de agregar funciones escritas en *LISP* a OM es a través de las cajas de funciones *LISP*, que son abstracciones definidas en código simbólico, no visual como el de OM.

Para crear una caja de código vamos al menú contextual haciendo *click* derecho sobre el fondo de la ventana de programación, y de allí a *Internal/LISP Function*. Siguiendo estos pasos creamos una abstracción interna (su caja es roja, como las abstracciones internas programadas en OM). Si deseamos, en cambio, una caja almacenada en el *Workspace*, accesible por todos los patches, vamos al menú *File* del *Workspace* y elegimos *New LISP Function*.

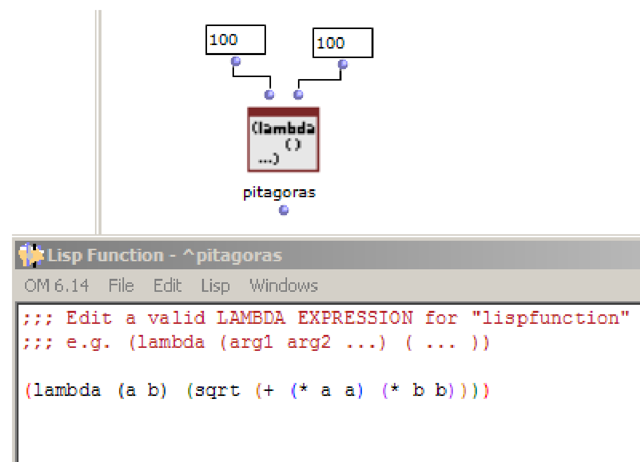
Cualquier sea el método que elijamos, al abrir el ícono de la función *LISP* observaremos lo siguiente.



62 – código

Figura 150. Funciones *lambda*

Según se ve, se trata de una función *lambda* cuyo contenido puede ser modificado. Por ser una función anónima no posee ningún identificador, a excepción del nombre que editemos debajo de la caja que la representa. Además, una función de este tipo sólo cuenta con un único *outlet*. Veamos un ejemplo donde codificamos la solución al teorema de Pitágoras.



62 – código

Figura 151. Caja de funciones *lambda*

Una cuarta instancia que permite crear código *LISP* en el entorno de OM es la programación de librerías especializadas. El *Forumnet* del IRCAM[41] ofrece varias de ellas, destinadas a extender los alcances de *OpenMusic*. Dado que no todas las librerías existentes se encuentran alojadas allí, puede ser de interés para el usuario buscar otras a través de la web. Dado que OM es un lenguaje *open source*, contamos con la ventaja de poder analizar el código de las librerías existentes, y de ese modo adquirir mayores conocimientos que nos permitan mejorar nuestras aplicaciones.

[41] http://forumnet.ircam.fr/product/openmusic-libraries-en/

CAPÍTULO 3

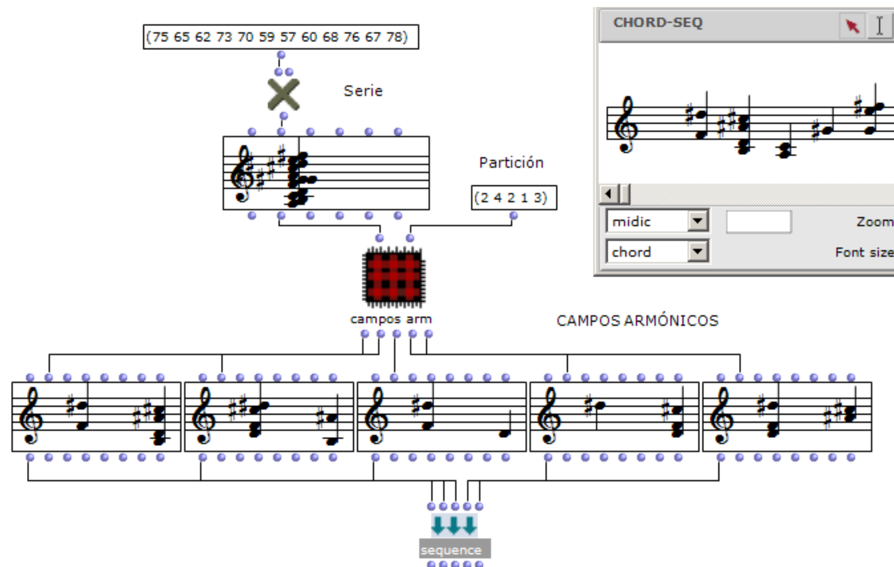
Aplicaciones musicales

La práctica de la programación

A fin de ampliar y afianzar los alcances de OM y *LISP*, procederemos a formalizar algunos procedimientos compositivos. Mediante el análisis de los *patches* veremos algoritmos que complementan a los ya estudiados, y que sin duda servirán de punto de partida para nuevos desarrollos.

Campos armónicos

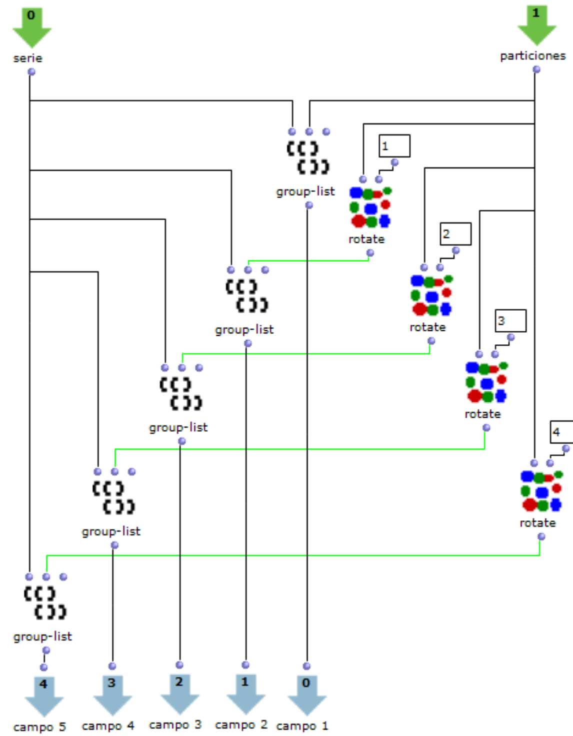
Pierre Boulez crea para *Le marteau sans maître* (1954) un plan de alturas basado en una serie dodecafónica registrada. A partir de la secuencia numérica 2, 4, 2, 1, 3, agrupa las notas de la serie en cinco complejos armónicos (acordes), y posteriormente realiza permutaciones circulares de esa secuencia con el propósito de obtener distintas subdivisiones de la serie original, y un mayor repertorio de complejos. A cada grupo, de los cinco obtenidos, lo denomina campo armónico. Veamos el *patch* que los calcula.



63 - campos armónicos

Figura 152. Campos armónicos en *Le marteau*…

Al abrir la abstracción observamos que la función `group-list` resulta clave para realizar las subdivisiones de la serie según cantidad de notas. Por otra parte, las permutaciones circulares las obtenemos con `rotate`.



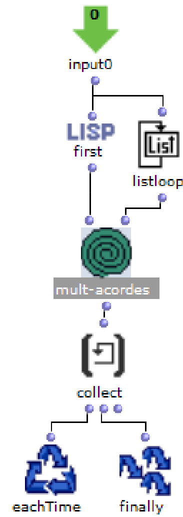
63 - campos armónicos

Figura 153. Contenido de la abstracción

Multiplicación de acordes

Una vez obtenidos los campos armónicos, Boulez amplía el material mediante un procedimiento denominado multiplicación de complejos armónicos, que consiste en la transposición del primer acorde sobre cada una de las notas del segundo acorde.

La figura siguiente muestra el bucle que realiza esta tarea. Tomamos el primer acorde (*a*) del campo armónico dato, y lo vamos multiplicar por cada uno de los complejos de ese campo (*a* x *a*, *a* x *b*, *a* x *c*, *a* x *d* y *a* x *e*).

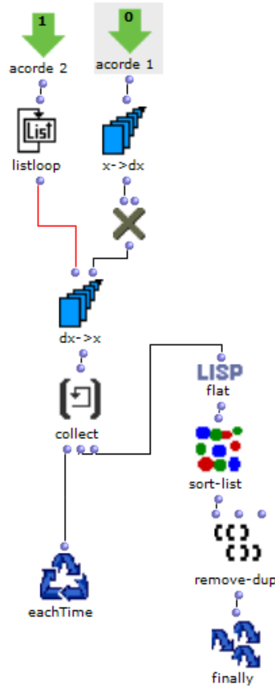


64 – mult acordes

Figura 154.

Bucle de multiplicación de acordes

El bucle interno que realiza la multiplicación propiamente dicha es el que sigue. Se calculan los intervalos del primer acorde y se superponen sobre cada una de las alturas del segundo acorde. Al finalizar el bucle se eliminan los paréntesis internos, se ordena la lista y se remueven las duplicaciones que pudieran existir.

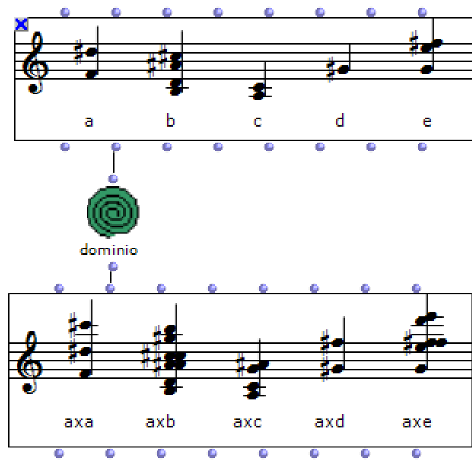


64 – mult acordes

Figura 155.

Bucle interno

Los resultados se muestran a continuación. Cuando multiplicamos todos los acordes de un campo entre sí obtenemos un *dominio*. De los cinco campos armónicos surgen cinco dominios diferentes.



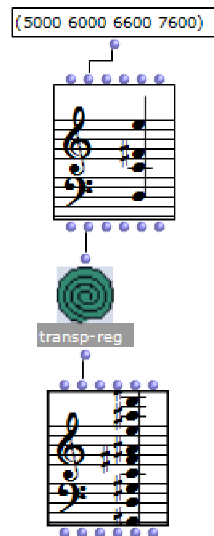
64 – mult acordes

Figura 156.

Resultados de la multiplicación del primer complejo de un campo armónico

Transposiciones limitadas registradas

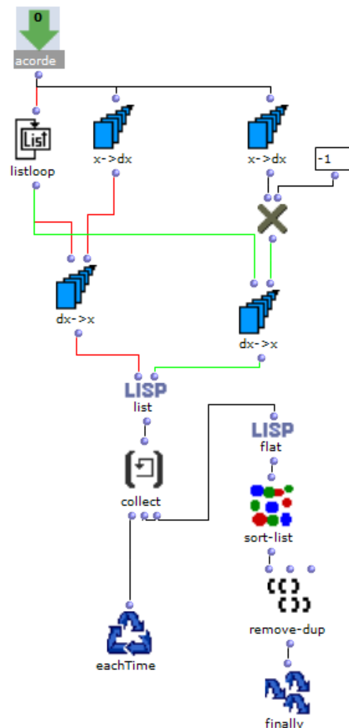
Un procedimiento similar, basado en la multiplicación de acordes, es la transposición de la estructura interválica de un acorde sobre sus propias notas, conservando la registración que resulta de tal operación. La diferencia con la técnica de Boulez es que la superposición no sólo parte de la nota inferior de la estructura, sino también de la superior. Por ejemplo, un acorde de *do* mayor transpuesto sobre sí mismo da, del grave al agudo, *do-mi-sol*; *mi-sol#-si* y *sol-si-re*. Y del agudo al grave, partiendo de la nota superior de la estructura: *sol-mi-do*; *mi-do#-la* y *do-la-fa*.



65 – transp registradas

Figura 157.

Transposiciones limitadas registradas



65 – transp registradas

Figura 158.

Contenido del bucle

Mediante este procedimiento, si mantenemos la registración fija, es posible lograr un alto grado de coherencia en la altura, y un timbre particular. La registración creada puede sostenerse en el tiempo, o bien modular a otras más o menos consonantes.

Conjuntos de grados cromáticos

En relación con los conjuntos de grados cromáticos (*Pitch Class Sets*), OM contiene diversas funciones, disponibles desde el menú *Functions/MathTools*.

Para comenzar, vamos a crear una abstracción que genere la forma prima a partir del nombre del PCS, y la transforme aleatoriamente. La programación puede verse en el gráfico siguiente. La función `pc-set` recibe el nombre del conjunto y devuelve su forma prima. Luego procedemos a permutarlo al azar, a invertirlo o no a través de un condicional, y a transportarlo.

La función `pc-set,` que se encuentra en *Functions/MathTools/Groups/Dn,* puede devolver la información de tres modos distintos: *integer* (forma prima del PCS, con números del 0 al 11), *vector* (vector interválico) o *pitch* (forma prima con los nombres de las notas). Nótese, además, los caracteres que deben rodear al nombre (`|3-3|`).

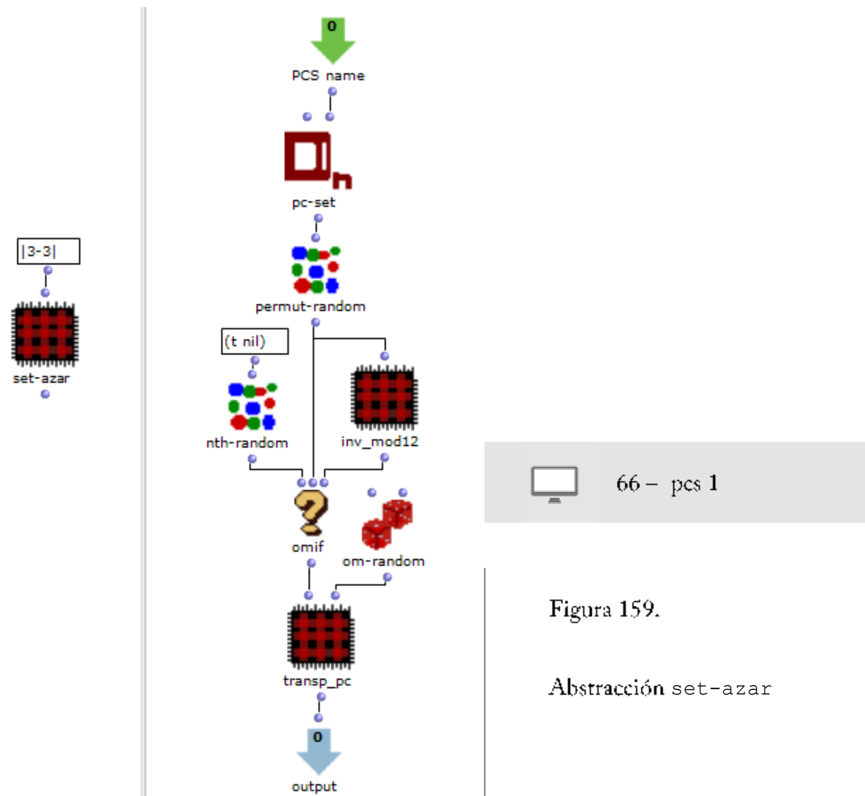


Figura 159.

Abstracción `set-azar`

Ahora realizaremos una función en *LISP* que genere una cadena de PCS, enlazados por una nota en común, expresando los grados cromáticos mediante números del 0 al 11. En el primer *inlet* ubicaremos el nombre del PCS a encadenar, y en el segundo la cantidad de conjuntos a enlazar.

Figura 160. Función *LISP* para generar una cadena de PCS

A continuación se muestra el código. Además de la función `cadena`, creamos otras dos. Una que transporta un conjunto ordenado sobre un grado cromático dado, y la otra que invierte módulo 12.

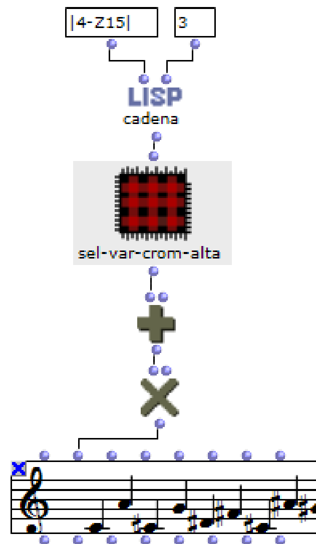
```
(in-package :om)

(defun cadena (pcs cant)
   (let ((temp nil) (result nil) (ind 0))
      (loop for x from 1 to cant do
         (setf temp (permut-random (pc-set :integer pcs)))
         (if (nth-random '(t nil)) (setf temp (inv-12 temp)))
         (push (setf temp ( cdr (transp-pc temp ind))) result)
         (setf ind (last-elem temp)))
      (flat (list 0 (reverse result)))))

(defun inv-12 (pc-list)
  (mapcar #' (lambda (i)(mod (- 12 i) 12)) pc-list))

(defun transp-pc (pc-list ind)
  (mapcar #' (lambda (i)
    (mod (+ (- ind (first pc-list)) i) 12)) pc-list))
```

Esta función es bastante simple, y no tiene en cuenta la variedad cromática. Por esta razón, vamos a realizar una abstracción que calcule varias cadenas y elija la que involucre el mayor número de grados distintos.



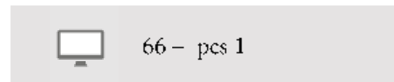
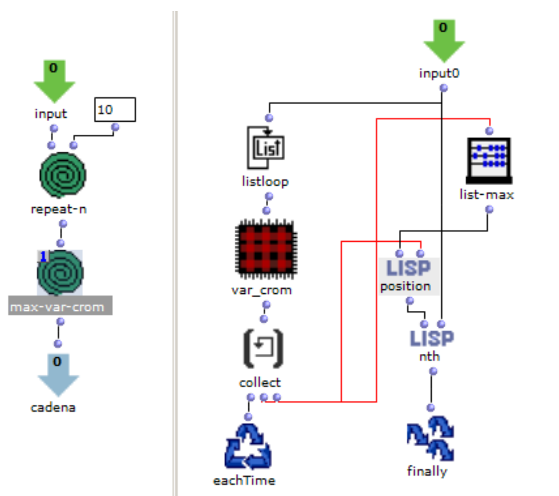


Figura 161. Abstracción para elegir una cadena con variedad cromática alta

Para ello, calculamos dentro de un bucle la densidad cromática de cada una de las cadenas generadas y recolectamos esos valores en una lista. Luego averiguamos el valor máximo dentro de la lista con `list-max` y su posición con `position`. Finalmente, devolvemos la cadena que se encuentra en esa misma posición, con `nth`. Los grados cromáticos de la cadena son multiplicados por 100 y se les suma 6000 para convertirlos a notas MIDI. Por último, el objeto `chord-seq` muestra los resultados en notación musical.

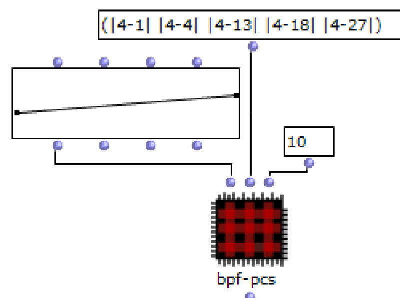


66 – pcs 1

Figura 162. Interior de la abstracción

Sería interesante que pudiéramos extender la función que genera las cadenas, para que actúe sobre una lista de varios PCS distintos. Incluso, que pudiéramos recorrer la lista por medios gráficos, con una envolvente, para controlar el grado de consonancia o disonancia de la cadena en el tiempo.

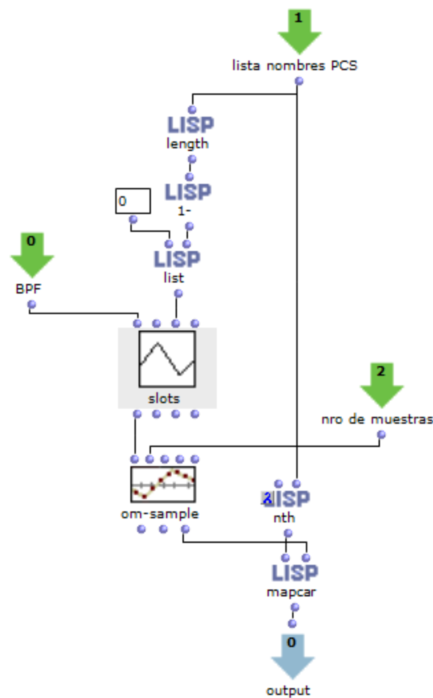
La figura siguiente muestra una abstracción que recibe una instancia de la clase `bpf`, una lista ordenada de nombres de PCS, y el número total de conjuntos a enlazar por medio de sonido común. La misma devuelve una lista de nombres de PCS en relación con el contorno de la curva.



67 – pcs 2

Figura 163. Control de la consonancia

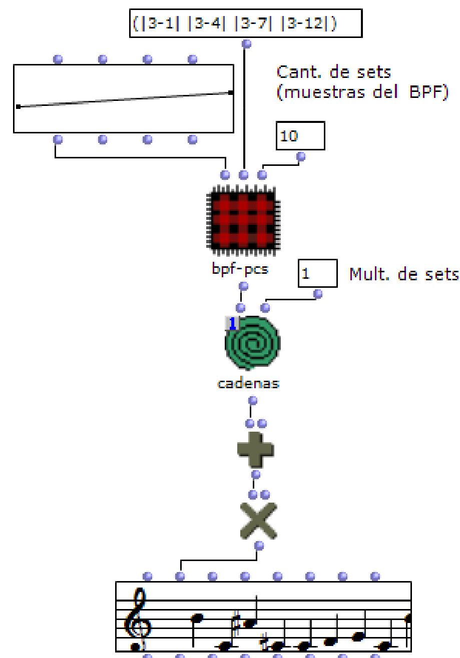
En el interior de la abstracción podemos apreciar el uso de *slots* para modificar los datos de la instancia `bpf` que ingresa por el primer *inlet*. También el uso de `nth` en modo *lambda* con `mapcar`, para asociar números de posición con los nombres de los PCS.



67 – pcs 2

Figura 164. Contenido de la abstracción `bpf-pcs`

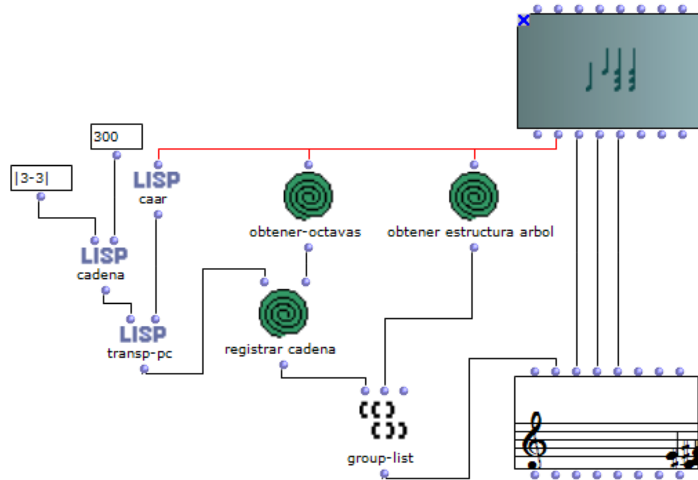
El *patch* completo se muestra en la figura siguiente. Cuenta con la abstracción antes comentada y con un bucle para procesar las llamadas a la función `cadena`. La unión entre cadena y cadena se realiza por medio de un intervalo al azar.



67 – pcs 2

Figura 165. Cadenas de PCS con consonancia – disonancia controlada por envolventes

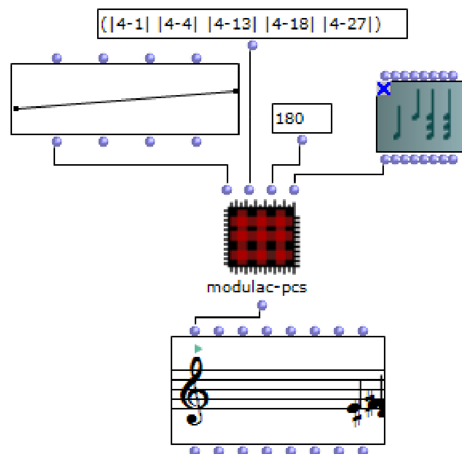
El siguiente programa permite cambiar el contenido interválico de una secuencia almacenada en una clase `chord-seq`. Un bucle (`obtener estructura árbol`) obtiene la longitud de las sublistas de notas MIDI, a fin de determinar la densidad polifónica de los eventos, es decir, cuándo se trata de acordes y cuándo de notas sueltas. Otro bucle obtiene la octava de las notas originales (`obtener-octava`), y el tercero registra las notas de la cadena de acuerdo a esa información (`registrar cadena`).



68 – pcs 3

Figura 166. Cambio del contenido interválico de una secuencia

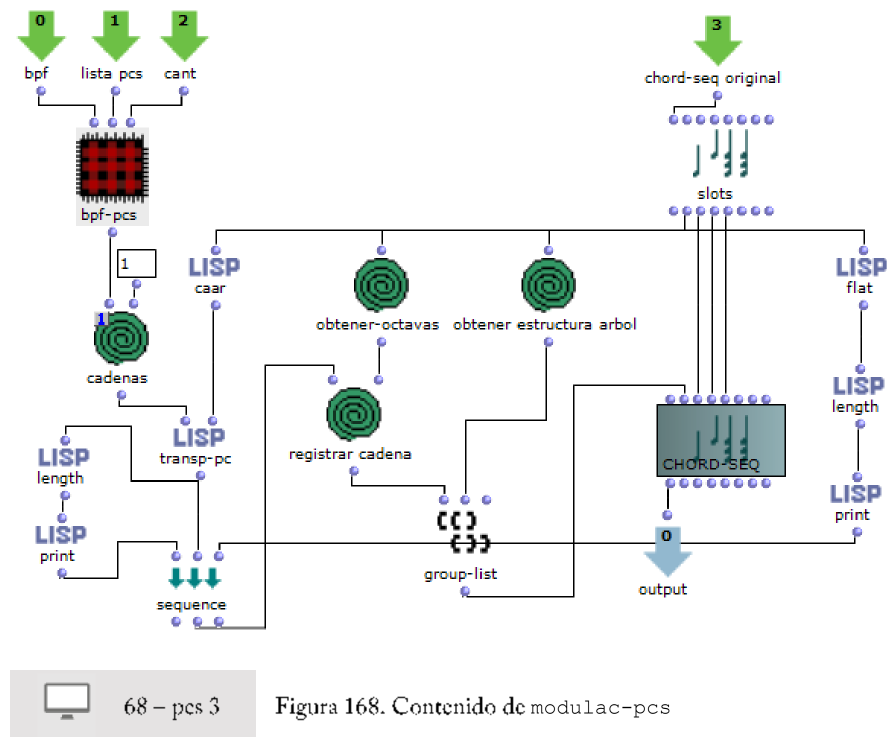
En el *patch* que sigue combinamos la posibilidad de cambiar el contenido interválico de una secuencia preexistente con la abstracción que permite elegir los PCS mediante una envolvente (`bpf-pcs`).



68 – pcs 3

Figura 167. Cambio de interválica con envolventes

Nuevamente, es importante observar el uso de *slots* para modificar los datos de la secuencia original almacenada en el objeto `chord-seq`, y de `sequence`, para evaluar en paralelo las distintas ramas del programa.



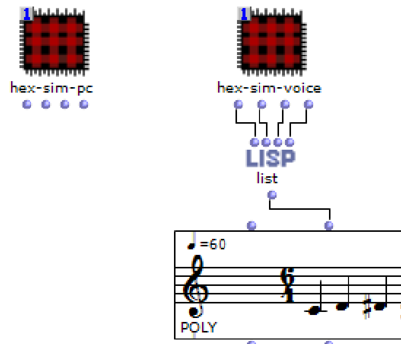
68 – pcs 3

Figura 168. Contenido de `modulac-pcs`

Series con hexacordios simétricos

Las series dodecafónicas simétricas son aquellas en las que el segundo hexacordio es una transposición del primero, con las notas en idéntico orden, o una retrogradación transpuesta, o una inversión transpuesta, o una retrogradación de la inversión transpuesta.

El siguiente *patch* crea el primer hexacordio al azar y calcula un segundo hexacordio para cada una de las variantes expuestas anteriormente. Vemos dos variantes: uno que devuelve el resultado en números de grados cromáticos (0 a 11) y el otro en un formato comprensible por la clase `poly`.

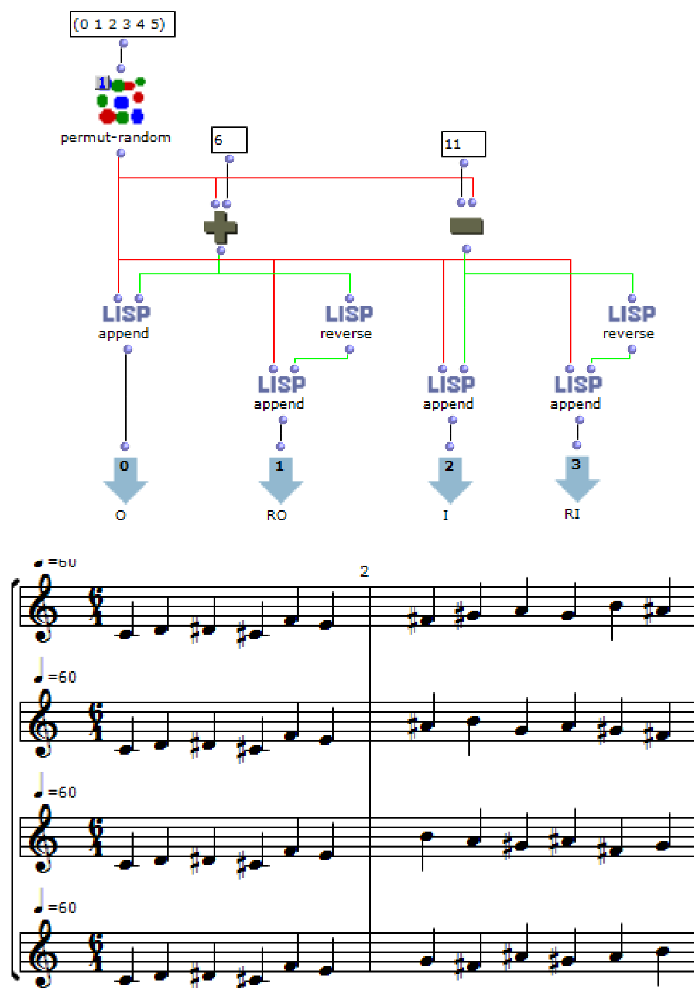


69 – simetría

Figura 169.

Series simétricas

El contenido de la abstracción es el que sigue a continuación. Se observa, luego, uno de los resultados posibles.

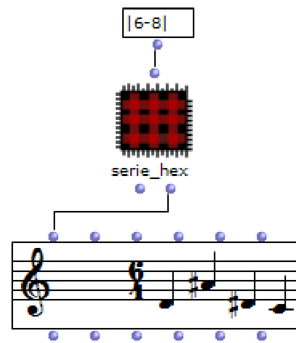


69 – simetría

Figura 170. Contenido de `hex-sim-pc` y resultados

Combinatoriedad

El siguiente *patch* genera aleatoriamente una serie dodecafónica, especificando el PCS del hexacordio.

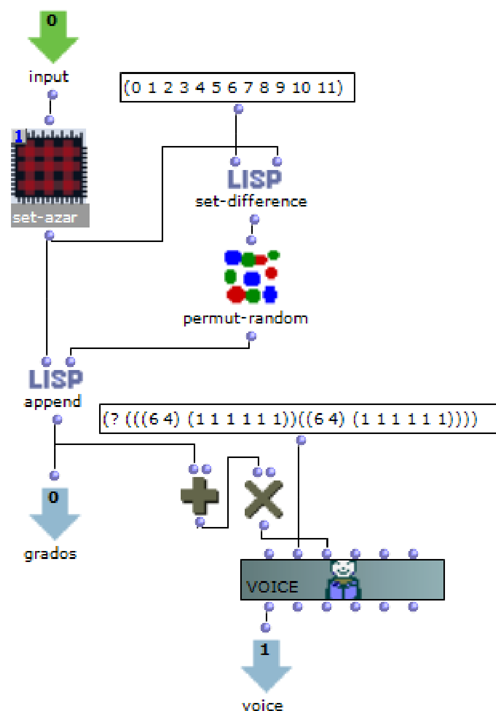


70 – combinatoriedad 1

Figura 171.

Serie a partir de PCS de 6

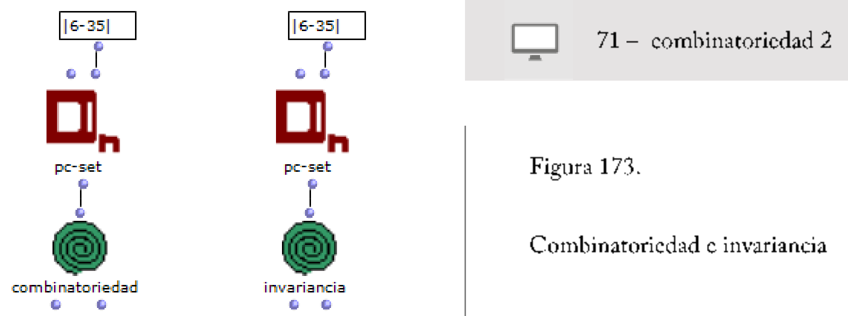
Con la abstracción set-azar generamos un conjunto transformado respecto a la forma prima, calculamos su complemento con `set-difference`, lo permutamos al azar y lo preparamos para ser representado en una clase `voice`.



70 – combinatoriedad 1

Figura 172. Contenido de `serie-hex`

Los dos programas que siguen calculan las transformaciones *TnI* que generan combinatoriedad e invariancia en un PCS de 6 elementos. Existe combinatoriedad cuando transformando el hexacordio por *Tni* (transposición o inversión seguida de transposición) obtenemos el complemento (las notas que faltan para completar el total cromático). La invariancia, por el contrario, se produce cuando la transformación arroja las mismas notas, pero cambiadas de orden.



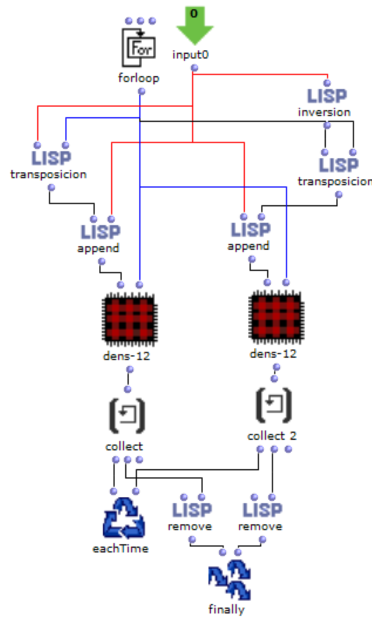
71 – combinatoriedad 2

Figura 173.

Combinatoriedad e invariancia

El PCS 6-35, por ejemplo, produce combinatoriedad para *T1, T3, T5, T7, T9, T11, T1I, T3I, T5I, T7I, T9I* y *T11I.* E invariancia para todas las transformaciones de índice par (*T0, T2, T4…, T0I, T2I…*). Los índices de transposición se obtienen por salidas distintas del bucle: `[(1 3 5 7 9 11) (1 3 5 7 9 11)]` para combinatoriedad y `[(0 2 4 6 8 10) (0 2 4 6 8 10)]` para invariancia; la primera lista (*oulet* izquierdo) corresponde a los índices de *Tn* y la segunda (*oulet* derecho) a los índices de *TnI*.

Para obtener los resultados calculamos todas las transformaciones y comparamos cada una con el hexacordio original, para saber si ambos dan el total cromático (variedad cromática igual a 12) para la combinatoriedad, o si dan una variedad de 6, para el caso de invariancia.



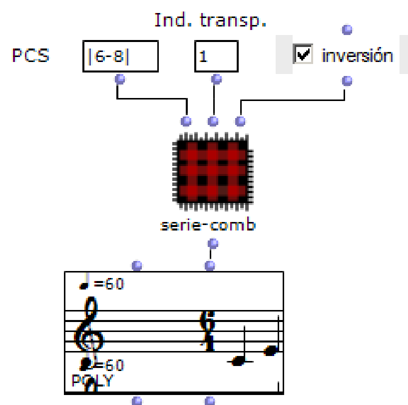
71 – combinatoriedad 2

Figura 174.

Contenido del bucle de combinatoriedad

Finalmente, vamos a generar series dodecafónicas combinatorias, en las cuales una transformación por *Tn* o *TnI* de la serie original produzca combinatoriedad. Es decir, que las notas del primer hexacordio aparezcan en el segundo hexacordio de la serie transformada, cambiadas de orden, y que las del segundo hexacordio aparezcan en las del primero. El objetivo de esta técnica es poder escribir a dos voces manteniendo la variedad cromática al máximo nivel posible, ya que la superposición del primer hexacordio de la serie original y el primero de la transformada dan el total cromático. Lo mismo ocurre con los segundos hexacordios de ambas series.

Nótese en el *patch* el uso de un *checkbox* para forzar o no la inversión de las serie. Se trata de una clase disponible en *Classes/Kernel/Interface Boxes*.

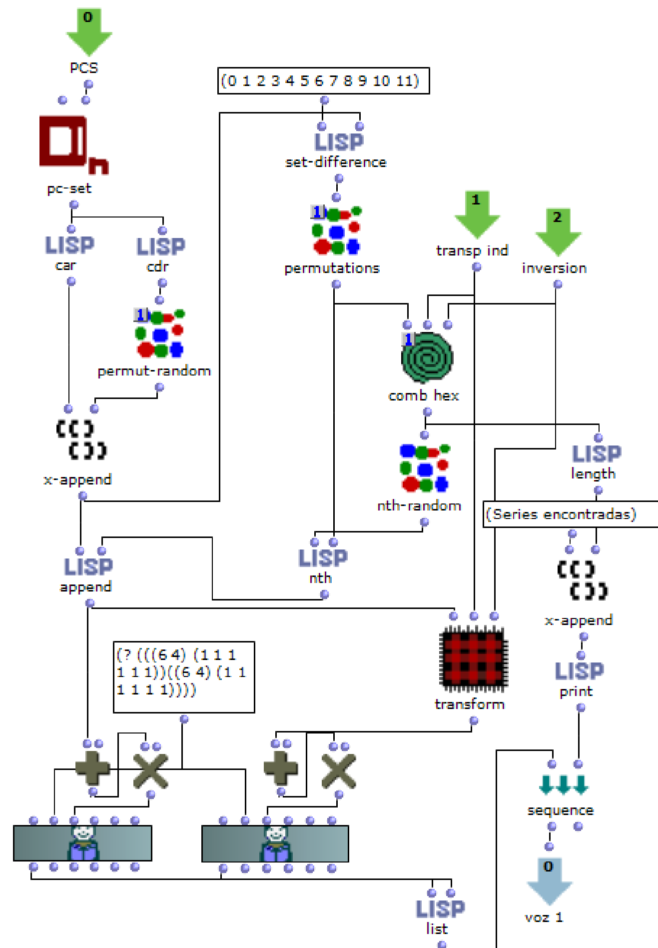


72 – combinatoriedad 3

Figura 175.

Series combinatorias

A continuación se observa el contenido de la abstracción. Para `transform` se crearon dos funciones *LISP*, de transposición e inversión. Seleccionándolas y presionando *e* es posible ver el código fuente.



72 – combinatoriedad 3

Figura 176. Contenido de `serie-comb`

Y uno de los resultados posibles es:

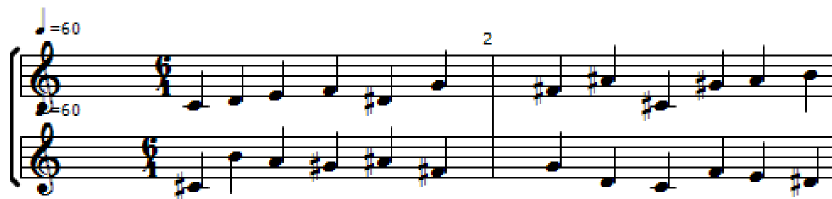
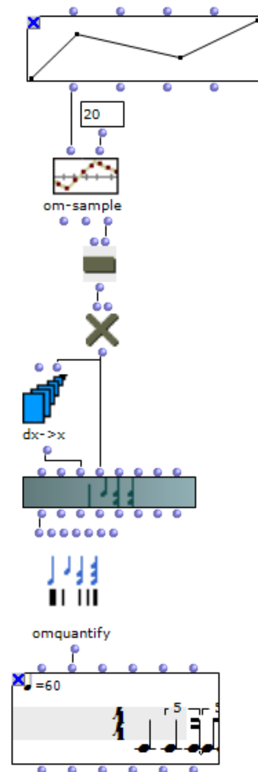


Figura 177. Resultado de la combinatoriedad serial

Gestualidad rítmica

Mediante esta aplicación generaremos una secuencia rítmica cuyos tiempos de ataque y duraciones sean proporcionales a los valores de amplitud de una curva diseñada con `bpf`.



73 – gesto rítmico

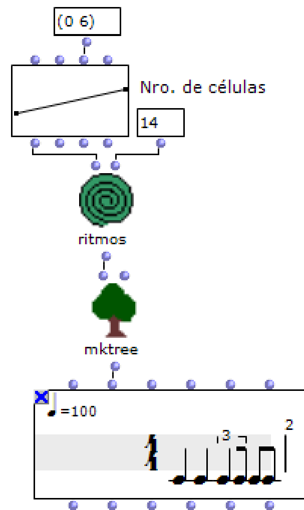
Figura 178.

Rítmica a partir de envolvente de duraciones

La curva es muestreada con `om-sample` y luego escalada. Las muestras de amplitud son interpretadas como valores de duración, mientras que los tiempos de ataque son calculados a partir de esas duraciones, por acumulación, con dx->x e ingresados a la clase `chord-seq`. La conversión de notación proporcional a mensurada se realiza mediante una cuantización, y el resultado final es representado con `voice`.

Densidad cronométrica

Planteemos ahora un programa que genere una secuencia rítmica de manera aleatoria, pero cuya densidad cronométrica esté determinada de acuerdo a una envolvente. El gráfico siguiente ilustra su realización.



73 – gesto rítmico

Figura 179.

Rítmica a partir de envolvente de densidad cronométrica

Para una recta ascendente, dos resultados posibles podrían ser:

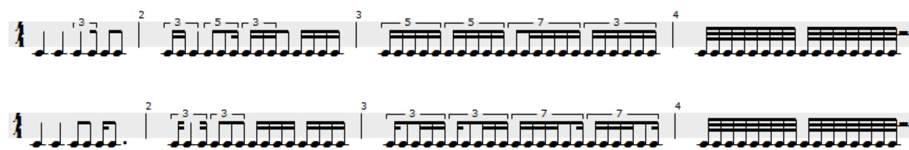
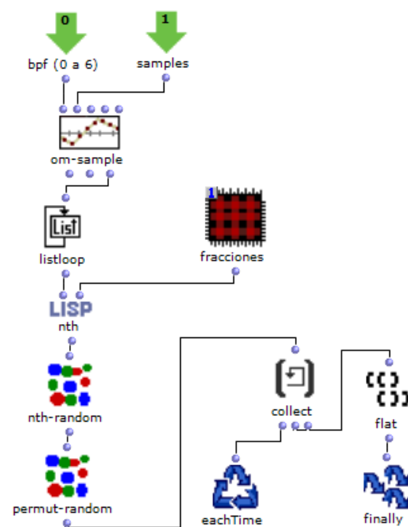


Figura 180. Resultado del control de la densidad cronométrica

El contenido del bucle es el siguiente:



73 – gesto rítmico

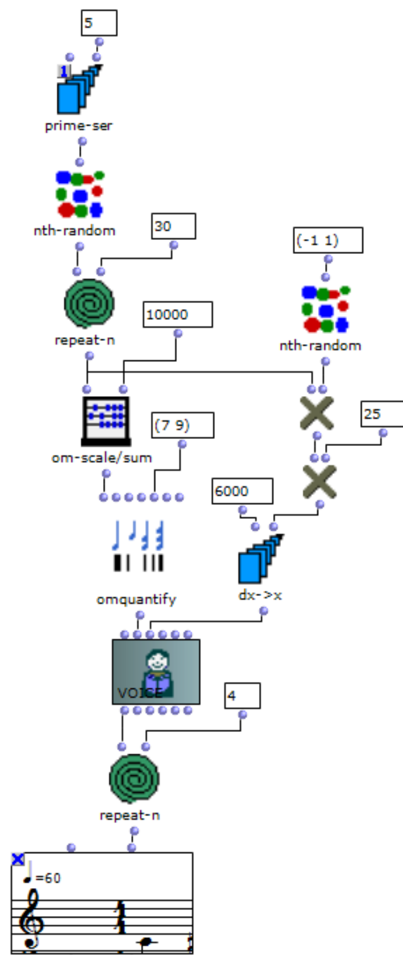
Figura 181.

Contenido del bucle de densidad cronométrica

La abstracción `fracciones` contiene una lista de células rítmicas ordenadas por densidad, en la cual cada evento está representado por un número fraccionario (la negra es 1/4). De acuerdo al valor de la curva, se elige al azar una célula rítmica que cumpla con la densidad requerida.

Números primos aplicados a la altura y al ritmo

De una sucesión de números primos (1, 2, 3, 5, 7, 11) elegimos 30 en orden aleatorio y los escalamos de modo tal que su suma de 10.000. Estos números son interpretados como duraciones a cuantizar, con una suma equivalente a 10 segundos.



74 – primos

Figura 182.

Empleo de números primos para determinar altura y ritmo

Por otra parte, la misma secuencia de números primos va a determinar la cantidad de cuartos de tono -ascendentes o descendentes- centrados alrededor del *do* 4. Finalmente, repetimos el proceso 4 veces a fin de obtener una polifonía. Un

resultado musical, de los posibles debido al uso del azar, es el que se muestra a continuación

Figura 183. Resultado de la utilización de números primos

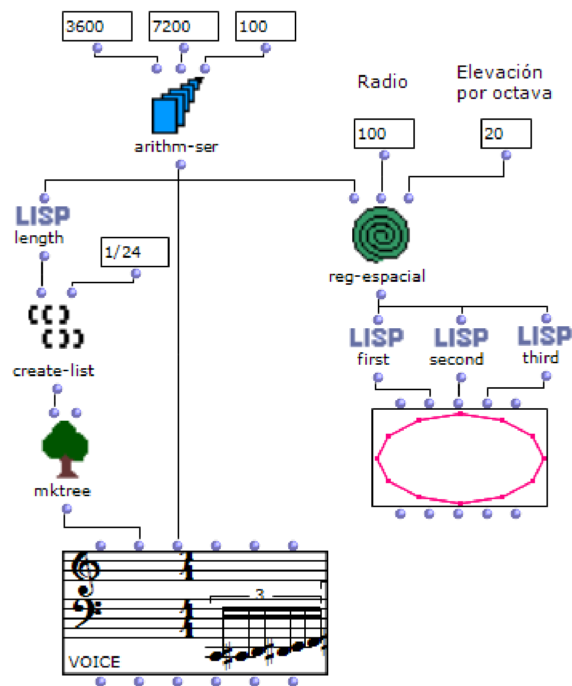
Registración espacial de la altura

En relación con la localización espacial del sonido, nos proponemos realizar ahora una aplicación que asigne una posición fija a cada nota, en un espacio tridimensional simulado. Para ello vamos a seguir un modelo utilizado en psicoacústica, que ubica a cada grado de una escala cromática en una espiral ascendente, de modo tal que las notas de igual nombre caen sobre la misma vertical, y en el que cada espira representa a una octava.

Aplicando este método, una melodía cualquiera se desarrollaría siguiendo una trayectoria determinada por sus notas y por su registración. Si ejecutáramos una simple escala cromática ascendente con duraciones idénticas, percibiríamos un ascenso gradual por la espiral, a una velocidad determinada. Pero si ejecutáramos, en cambio, una escala por tonos, notaríamos que la velocidad se duplica. Esto significa que para duraciones iguales de cada nota, la velocidad de desplazamiento del sonido queda supeditada a la interválica. En una escritura contrapuntística, las voces se entrecruzarían en el espacio, asociándose su direccionalidad e interválica con las trayectorias que cada una realiza, integrándose de este modo la localización al comportamiento de la altura.

No vamos a tratar aquí ninguna técnica de espacialización del sonido[42], pues no es la finalidad de nuestro estudio. Nos limitaremos, simplemente, a calcular las coordenadas del espacio que surgen de la aplicación de este procedimiento.

Con `arithm-ser` obtenemos los semitonos ascendentes entre las notas 3600 y 7200. Por un lado las representamos en notación musical, y por otro, las ingresamos en el bucle que calcula la registración espacial. Posteriormente, representamos la trayectoria con la clase `3dc`.



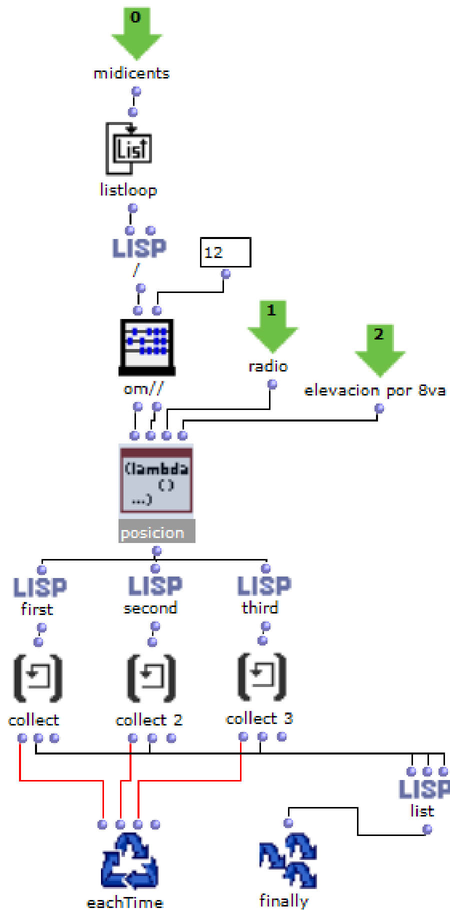
75 – espacio 1

Figura 184. Registración espacial de la altura

Vemos, a continuación, el contenido del bucle `reg-espacial`. En él separamos grado de octava con om// e ingresamos esos datos y las dimensiones de la espiral virtual en una función *LISP*.

---

[42] Para ello puede consultarse Cetta, P. *Un modelo para la simulación del Espacio en Música*. Buenos Aires. EDUCA. 2004.



75 – espacio 1

Figura 185. Contenido de `reg-espacial`

A través de los argumentos aportados, la función calcula las coordenadas a asignar a cada nota.

Lisp Function - ^posicion

OM 6.14 File Edit Lisp Windows

```
(lambda (oct grado radio elev8)
  (let ((x (* radio (sin (* grado 0.5233))))
        (y (* radio (cos (* grado 0.5233))))
        (z (+ (* (- oct 3) elev8) (* (+ grado 1) (/ elev8 12.0)))))
  (list x y z)))
```

75 – espacio 1

Figura 186. Código *LISP*. Obtención de coordenadas

La trayectoria que se obtiene en este caso es la misma espiral, dado que ingresamos al programa una escala cromática ascendente.

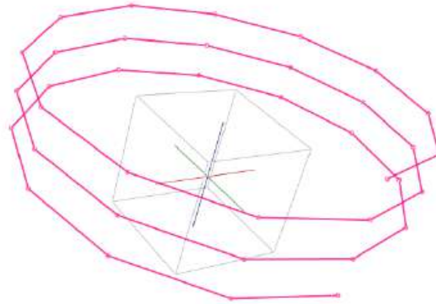


Figura 187. Editor 3dc. Espiral ascendente de tres octavas

Ritmos a partir de las primeras reflexiones

Continuando con las aplicaciones referidas a la espacialización del sonido y la música, vamos a intentar convertir las primeras reflexiones, que ocurren en un ambiente virtual, en ritmos aplicables a la escritura instrumental.

Cuando percibimos un evento sonoro en un ambiente cerrado, escuchamos en primer término el sonido que procede de la fuente, y luego los que resultan de las primeras reflexiones, contra las paredes y el techo, los que rebotan dos o más veces contra estas superficies, hasta percibir algo que nos rodea y que denominamos reverberación. Considerando el sonido directo y las primeras reflexiones (6), es posible pensar que al efectuar un *zoom* considerable sobre el espacio de audición escucharíamos esos ataques espaciados en el tiempo, y que podrían constituirse en células rítmicas asociadas a la posición de la fuente.

A fin de calcular la dirección de cada reflexión, la distancia que recorre, la intensidad con la que llega y el retardo con el que arriba, podemos recurrir al método de las fuentes fantasma, que consiste en rebatir el espacio de audición para una posición determinada de la fuente y del oyente. La figura siguiente ilustra este procedimiento. Una vez comprendido, los cálculos se resumen en sumas, restas y la aplicación del teorema de Pitágoras.

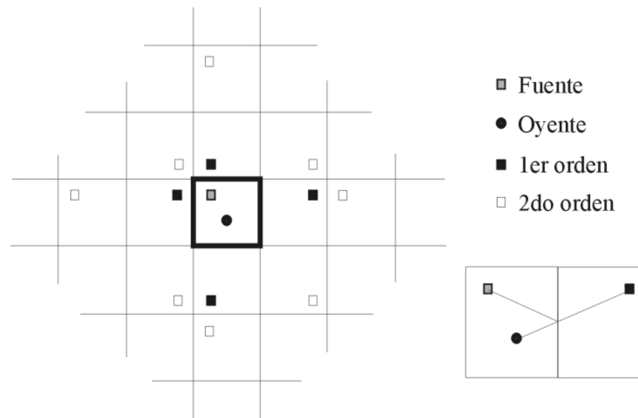


Figura 188. Método de las fuentes fantasma para calcular la distancia de las reflexiones

Una vez establecidas las dimensiones del ambiente virtual, y las posiciones de la fuente y del oyente (0, 0, 0), podemos calcular las distancias recorridas por las reflexiones a través de una función *LISP*.

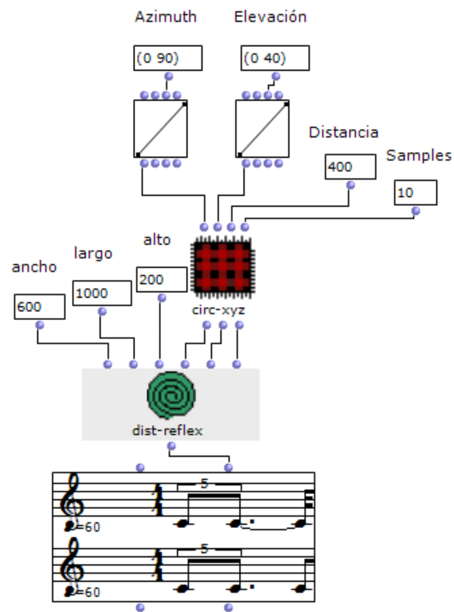
Lisp Function - ^dist-reflex

OM 6.14 File Edit Lisp Windows

```
(lambda (ancho largo alto x y z)
  (let* ((xr (- ancho x))
         (xl (- (* -1 ancho) x))
         (yf (- largo y))
         (yb (- (* -1 largo) y))
         (zu (- alto z))
         (zd (- (* -1 alto) z))
         (dist  (sqrt (+ (* x x) (* y y) (* z z))))
         (distr (sqrt (+ (* xr xr) (* y y) (* z z))))
         (distl (sqrt (+ (* xl xl) (* y y) (* z z))))
         (distf (sqrt (+ (* x x) (* yf yf) (* z z))))
         (distb (sqrt (+ (* x x) (* yb yb) (* z z))))
         (distu (sqrt (+ (* x x) (* y y) (* zu zu))))
         (distd (sqrt (+ (* x x) (* y y) (* zd zd)))))
    (list dist distr distl distf distb distu distd)))
```

Figura 189. Función que calcula las distancias de las reflexiones

A continuación vemos el *patch* que convierte a ritmos la posición de una fuente virtual en movimiento, y los resultados cuando recorre de 0 a 90 grados de azimut y de 0 a 40 grados de elevación, simultáneamente.



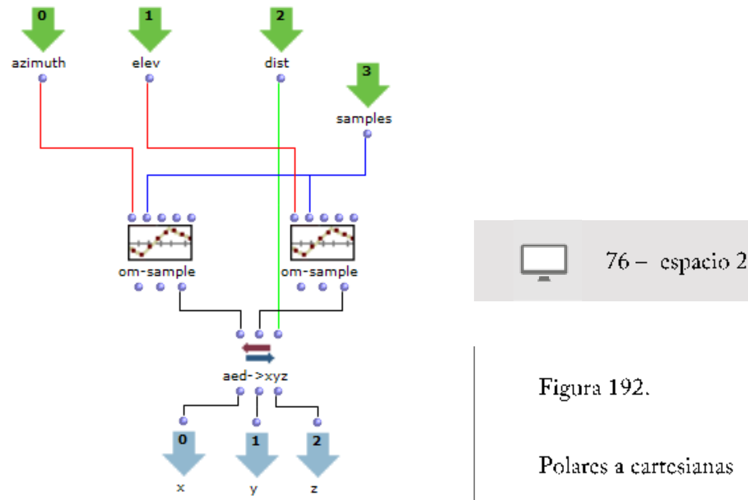
76 – espacio 2

Figura 190.

Reflexiones a ritmos

Figura 191. Ritmos generados por el movimiento de la fuente

La abstracción `circ-xyz` convierte las coordenadas esféricas en cartesianas, empleando la función `aed->xyz`, que se encuentra en el menú *Functions/Basic Tools/Geometry*.

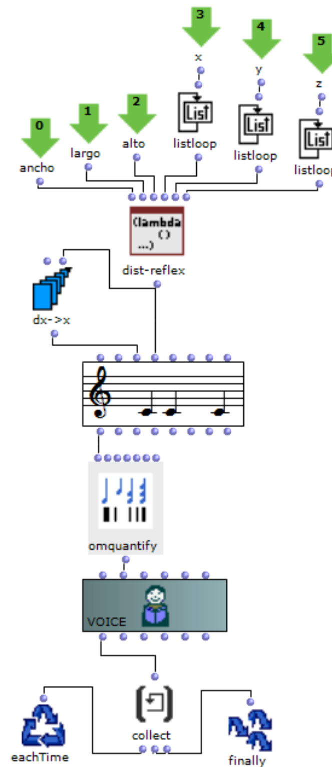


76 – espacio 2

Figura 192.

Polares a cartesianas

El bucle principal, `dist-reflex`, es el que sigue.

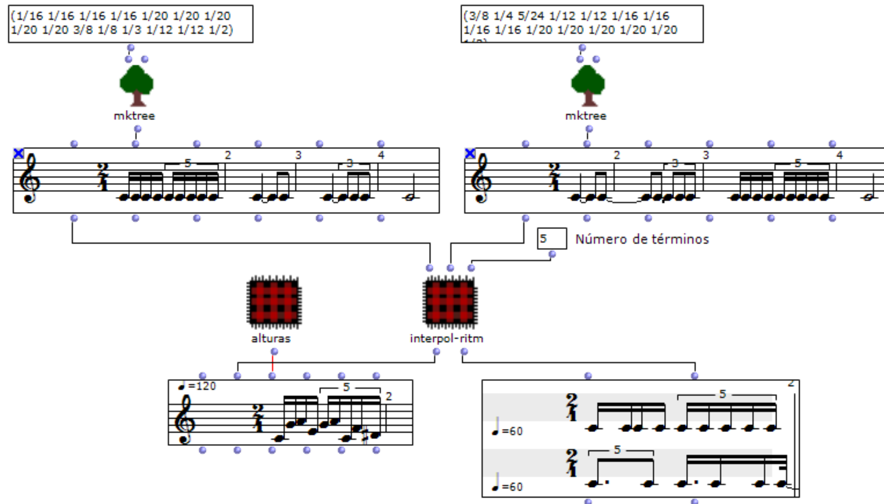


76 – espacio 2

Figura 193. Contenido del bucle principal

Interpolación rítmica

Partiendo de una secuencia rítmica, y considerando otra de llegada, vamos a generar secuencias intermedias que modulen gradualmente de principio a fin.

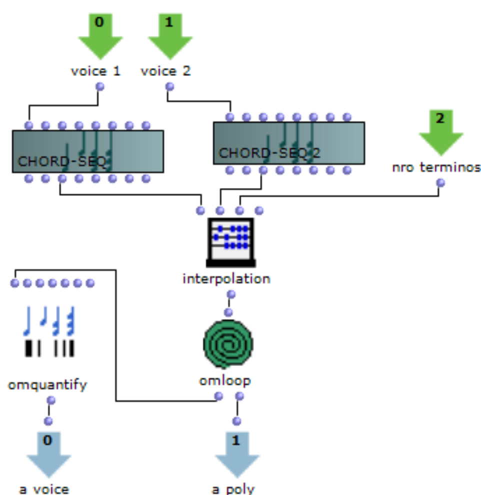


77 – interpol ritmos

Figura 194. Interpolación rítmica

Para ello, la abstracción `interpol-ritm` convierte los objetos `voice` a `chord-seq`, para que las duraciones queden expresadas en milisegundos y no como árboles rítmicos. Mediante la función `interpolation` obtenemos los términos intermedios a partir de las listas numéricas de los extremos. Luego, mediante un bucle, generamos tantas instancias `voice` como términos posee la interpolación, a fin de representarlas superpuestas con `poly`, y así poder compararlas.

El programa brinda dos salidas. Por un lado muestra los resultados a través de `voice`, ubicando los términos calculados uno a continuación del otro. Por otro lado, crea una polifonía, no con el objetivo de reproducir los términos de ese modo, sino para que sea más fácil compararlos visualmente. En el primer caso utilizamos una abstracción para brindar alturas a la secuencia, las cuales son generadas con la función `cadena`, antes vista.



77 – interpol ritmos

Figura 195.

Contenido de la abstracción

Los resultados en notación musical son los siguientes:

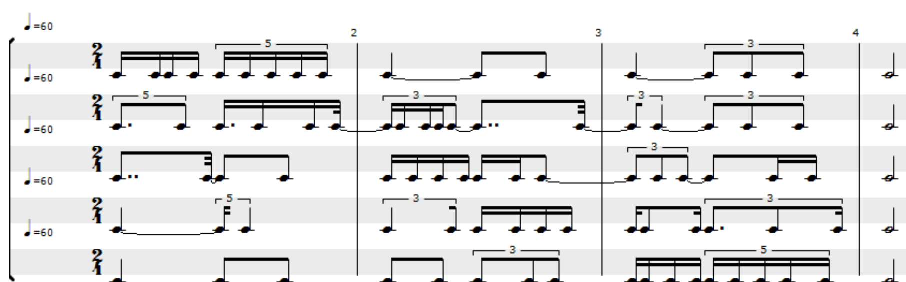


Figura 196. Resultados de una interpolación rítmica

Resíntesis instrumental

La resíntesis instrumental se basa en el análisis espectral de un sonido, y en su reproducción por medio de instrumentos tradicionales, de modo tal que la frecuencia y la amplitud de cada componente es ejecutada por un instrumento diferente. Para que la interpretación sea posible, es necesario redondear las frecuencias obtenidas a las del sistema temperado, o realizar una aproximación microtonal por cuartos, sextos u octavos de tono. Para el siguiente ejemplo partimos de un sonido electrónico generado por frecuencia modulada. El análisis espectral lo efectuamos con *Spear*[43], un software que nos permite grabar el resultado del análisis en un archivo del tipo

[43] http://www.klingbeil.com/spear/

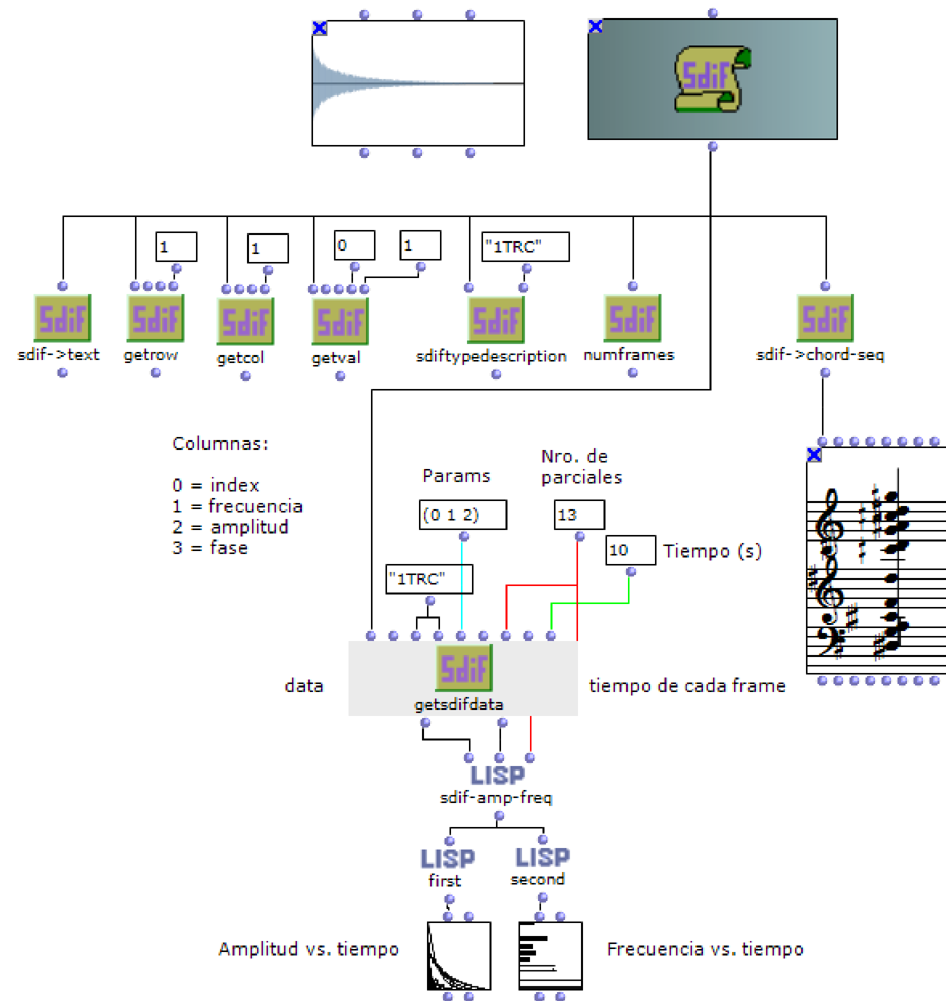
SDIF[44]. El formato empleado en este caso es el denominado “1TRC”, en el cual la información es estructurada en *frames*, ventanas temporales cuya duración es especificada por el mismo usuario. Cada *frame* alberga a las distintas componentes que tienen lugar en un instante determinado, y para cada uno de esos parciales se detalla su valor de frecuencia, de amplitud y de fase instantánea. La tabla siguiente muestra el contenido de los dos primeros *frames* del archivo SDIF. El análisis detectó 13 componentes, cuya información es la siguiente.

1TRC	1	0	0.0
1TRC	0x0004	13	4
Índice	**Frecuencia**	**Amplitud**	**Fase**
1	625.281	0.0475071	1.19961
2	1063.88	0.0804973	-1.57044
3	1299.86	0.137434	0.371725
4	1564.46	0.0527404	-1.801
5	2174.16	0.063414	-3.04091
6	2516.2	0.051151	2.12195
7	3267.29	0.0716478	-0.967817
8	1865.15	0.106242	-0.771784
9	157.855	0.0958051	-1.1172
10	192.145	0.0948067	1.28172
11	232.4	0.07602	-0.428978
12	278.855	0.148731	-1.77892
13	349.89	0.0467168	0.626311
1TRC	1	0	0.1
1TRC	0x0004	13	4
Índice	**Frecuencia**	**Amplitud**	**Fase**
0	159.186	0.0745236	-1.98298
1	628	0.0536272	-0.38122
2	1061.85	0.0511354	-2.1937
3	1298.99	0.117593	-0.49121
4	1563.27	0.0154642	-0.06835
5	2175.2	0.0415955	-0.40733
6	2517.36	0.0454587	-0.172631
7	3266.81	0.0591745	2.4682
8	1864.27	0.0605166	1.89654
10	197.238	0.0936982	-0.707634
11	236.413	0.0897065	3.04512
12	282.731	0.158854	-0.519338
13	352.487	0.0349092	2.27888

[44] http://sdif.sourceforge.net/

El primer *frame* corresponde al instante 0.0, mientras que el segundo al 0.1, o sea una décima de segundo más tarde.

La clase `sdiffile` de OM nos permite importar el archivo obtenido, y una vez cargado (evaluando la instancia de la clase), podemos accedemos a su contenido a través de las funciones que se encuentran en el menú *Functions/SDIF*.



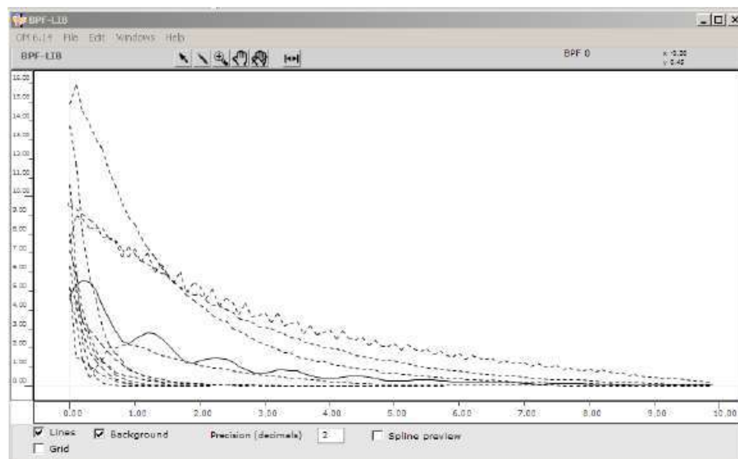
78 – resíntesis

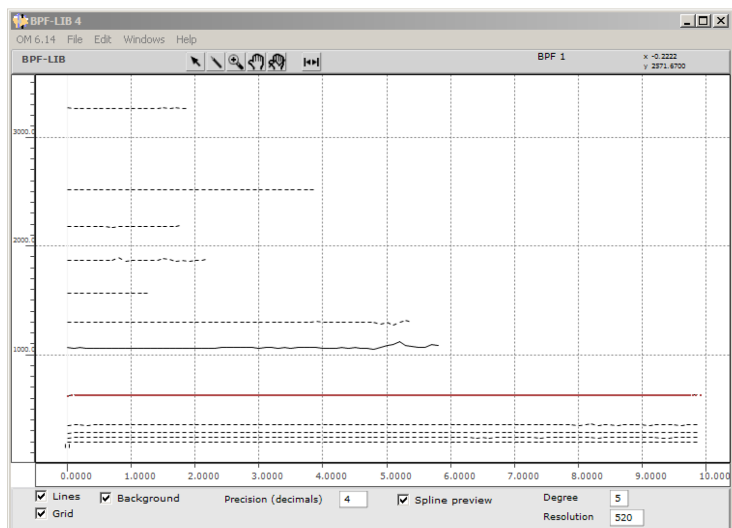
Figura 197. Resíntesis instrumental

Para ver la documentación de cada función podemos seleccionar al objeto y presionar la tecla *d*. Las funciones más importantes para nuestros fines actuales son `sdif-`

>chord-seq, que convierte la información del archivo al formato de la clase chord-seq, y getsdifdata, que nos devuelve el contenido del archivo.

Para su interpretación, y a los efectos de poder graficar tanto la amplitud como la frecuencia de los parciales en función del tiempo, creamos una función en *LISP* denominada sdif-amp-freq, que resulta útil a estos fines. El formato de salida de la función permite conectar sus *outlets* directamente a las clases bpf-lib.

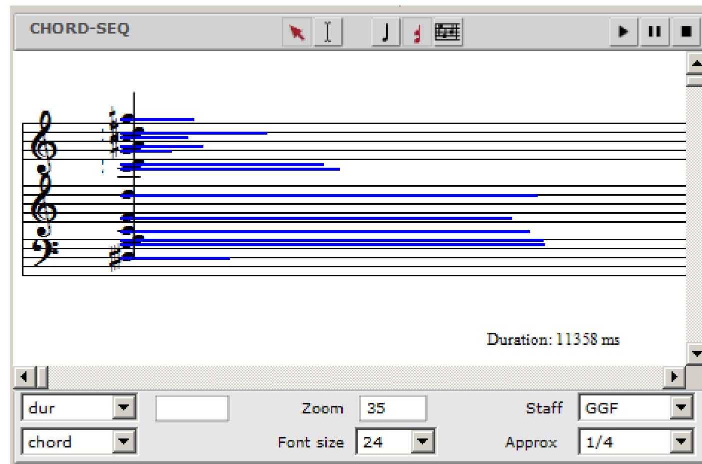




78 – resíntesis

Figura 198. Representación gráfica de los parciales

Otro modo de observar los resultados es a través del editor de la clase `chord-seq`. Si seleccionamos la opción *dur* del editor podremos ver, de manera rápida y concisa las relaciones de duración entre las componentes.



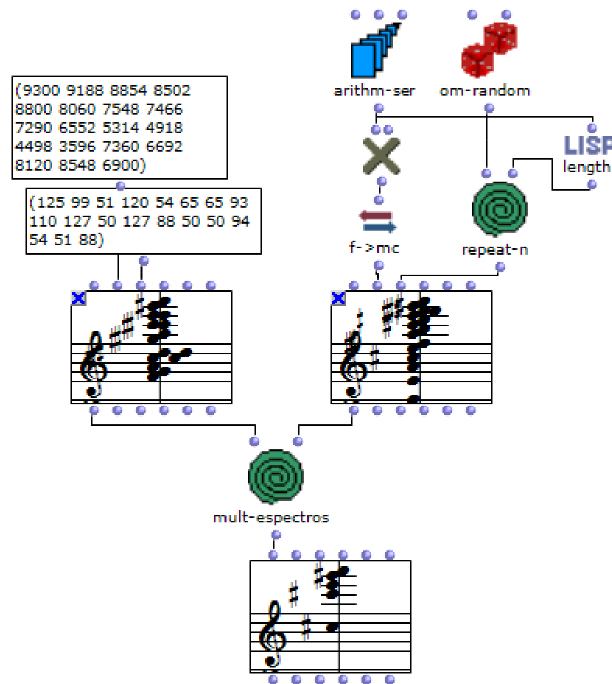
78 – resíntesis

Figura 199. Editor de `chord-seq`

Multiplicación de espectros

Una técnica frecuentemente empleada en el procesamiento digital de señales es la convolución de dos formas de onda, que se obtiene –dicho en términos muy generales- multiplicando sus respectivos espectros. Usada frecuentemente como medio para aportar reverberación natural a un sonido, la convolución suele utilizarse, además, como una técnica de síntesis, a la que se denomina "síntesis cruzada". Los resultados del análisis de dos sonidos destinados a la resíntesis, una vez aproximados a la resolución deseada, también son susceptibles de ser multiplicados en pos de generar un espectro nuevo. La multiplicación que se practica es la de las amplitudes entre componentes de igual frecuencia. Por lo tanto, las componentes que resultan son únicamente aquellas que se encuentran en ambos espectros, considerando que cero por cualquier número es cero.

En el ejemplo que sigue multiplicamos un acorde-espectro por los primeros 16 armónicos de una fundamental de 110 Hz, cuyas amplitudes son generadas aleatoriamente.



79 – mult espectros

Figura 200. Multiplicación de espectros

Dentro del bucle, se extraen los valores de amplitud y frecuencia de los objetos `chord` conectados a él. Luego de comparar si las frecuencias consideradas se hallan presentes en ambos espectros, sus valores de *key-velocity* son transformados en valores de amplitud, mediante una abstracción que realiza el siguiente cálculo[45]:

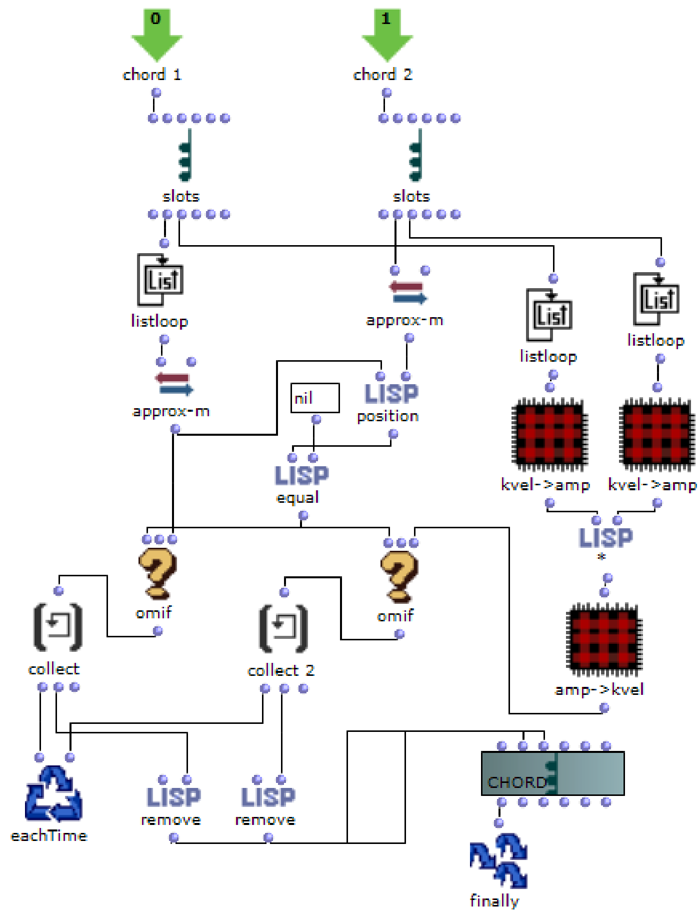
$$A = keyvel * 10^{\frac{0.30103}{127}} - 1$$

donde 0.30103 es igual al logaritmo en base 10 de 2.

Una vez multiplicadas las amplitudes se realiza el cálculo inverso, para convertir a esas amplitudes en valores de *key velocity*.

$$keyvel = \frac{127}{0.30103} * \log(A + 1)$$

[45] La norma MIDI no especifica el modo en que los valores de *key velocity* se convierten en amplitud, por lo cual cada fabricante de hardware implementa su propia forma de realizarlo.



79 – mult espectros

Figura 201. Contenido del bucle

CAPÍTULO 4

La librería *OMMatrix*

Matrices combinatorias

En este capítulo se describen las propiedades de una librería especialmente creada para el entorno de composición asistida OM, destinada a la generación y transformación de matrices combinatorias y a la realización de operaciones con conjuntos de grados cromáticos.

Las matrices combinatorias no seriales sirven para garantizar cierta coherencia interválica entre el aspecto horizontal y el vertical de una configuración de alturas determinada. En general, están formadas por un mismo conjunto de grados cromáticos (PCS, por *Pitch Class Sets*) tanto en sus filas como en sus columnas, y presentan homogeneidad interválica en ambas dimensiones.

A fin de caracterizar los distintos tipos de matrices y sus modos de construcción vamos a basarnos en la terminología empleada por Robert Morris en su artículo "Combinatoriality without the aggregate" (Morris, 1983). Una descripción detallada de sus modos de construcción, con ejemplos musicales, puede hallarse en el libro *Elementos de Contrapunto Atonal*, de Pablo Di Liscia y Pablo Cetta (2009), y también su aplicación a través de una librería especialmente programada para el entorno de procesamiento en tiempo real *Pure Data*.

Una primera clasificación de las matrices resulta de la división en simples y complejas. En el primer caso, la construcción de la matriz no depende de las propiedades del PCS elegido como norma generatriz, y el resultado tiende a ser una mera permutación de elementos, o bien una yuxtaposición de estructuras acórdicas. En esta categoría se inscriben las matrices conocidas como "cuadrado romano", tipo "IA", tipo "IB" y tipo "II".

Las matrices complejas, por su parte, demuestran amplias posibilidades como materiales para la composición, y se dividen en matrices generadas a partir de cadenas de PCS, y matrices creadas a partir de ciclos de un mismo operador.

Matrices simples

El "cuadrado romano" se construye a partir de las permutaciones circulares de los elementos de un conjunto cualquiera, ubicando cada permutación en una fila distinta. En el ejemplo siguiente, los grados del PCS elegido –numerados del 0 al 9 y con las letras A y B– se ubican de forma arbitraria en las posiciones de la primera fila de una matriz de 4 x 4. Las permutaciones circulares de las posiciones dan el siguiente resultado.

1B5	39		82
39		82	1B5
	82	1B5	39
82	1B5	39	

Según se observa, todas las filas y columnas tienen el mismo contenido, en distinto orden, por supuesto.

Las matrices tipo "IA", "IB" y "II" constan de un único elemento por posición. La matriz tipo "IA" se obtiene ubicando el PCS elegido en la primera fila y en la primera columna. Las filas o columnas siguientes se calculan transponiendo el PCS sobre sus propios grados. En el ejemplo que sigue, el PCS (8, 0, 1, 7, 6) se transporta sobre 0, 1, 7 y 6.

8	0	1	7	6
0	4	5	B	A
1	5	6	0	B
7	B	0	6	5
6	A	B	5	4

Como se aprecia, se trata de una yuxtaposición de acordes de igual estructura, que se reconoce tanto armónica como melódicamente.

Las matrices "IB", por su parte, contienen el PCS original en las filas, y su inversión en las columnas, como ocurre en el caso siguiente. La interválica es la misma en todas las filas y columnas de la matriz (PCS 5-7).

0	4	5	B	A
8	0	1	7	6
7	B	0	6	5
1	5	6	0	B
2	6	7	1	0

Las matrices tipo "II", en cambio, se forman con dos PCS diferentes, uno para las filas y otro para las columnas. Por esta razón, dos PCS pueden dar como resultado matrices rectangulares, si elegimos a cada uno con distinto número de elementos, como en el ejemplo.

A	2	3	9	8
B	3	4	A	9
7	B	0	6	5

Matrices complejas

Las matrices generadas mediante cadenas de PCS enlazados resultan de interés debido a sus características especiales. Una cadena se construye a partir de las particiones de un conjunto dado. Para la realización del ejemplo siguiente tomamos el PCS 5-15 = (0 1 2 6 8) y sus particiones A: 01|268 con subconjuntos 2-1|3-8 y B: 16|028 con subconjuntos 2-5|3-8. El subconjunto común a ambas (3-8) es lo que nos permite construir la cadena, aplicando transposiciones, inversiones y retrogradaciones a las particiones elegidas:

01	268	A
268	07	$T_6R(B)$
07	15B	$T_1I(B)$
15B	67	$T_7IR(A)$
67	028	$T_6(A)$
028	16	$T_0R(B)$
16	57B	$T_7I(B)$
57B	01	$T_1I(A)$

A partir de esta disposición estamos en condiciones de crear la cadena, que es cerrada y apta para generar una matriz, dado que el primer elemento y el último son iguales:

01|268|07|15B|67|028|16|57B|01

De la mera observación del procedimiento se desprende la necesidad de contar con una aplicación que nos asista en la realización de estas tareas. Una vez obtenida la cadena, la construcción de la matriz es relativamente simple.

01	268		
	07	15B	
		67	028
57B			16

Las matrices generadas a partir de ciclos de un mismo operador también revelan amplias posibilidades al utilizarlas en la composición. Se construyen ubicando el PCS utilizado como norma en la primera posición, y las transformaciones sucesivas a través de un mismo operador (*TnI*) a lo largo de la diagonal. Las dimensiones de la matriz dependen del operador elegido; en la tabla siguiente se describen estas relaciones.

Operador	*Dimensiones de la matriz*
T_6 - T_nI	2 x 2
T_4 – T_8	3 x 3
T_3 – T_9	4 x 4
T_2 – T_A	6 x 6

Para el siguiente ejemplo utilizaremos la técnica mencionada. El conjunto elegido es PCS 4-3 = {2, 1, 5, 4}, y el operador de transposición es *T3*.

2154			
	4578		
		78AB	
			12AB

En principio, puede parecer que tal configuración no tiene posibilidades de aplicación, debido al escaso nivel de distribución de las notas en las posiciones de la matriz. Para mejorar esta situación podemos recurrir a un procedimiento de intercambio gradual de elementos entre posiciones (*swapping*). En las matrices siguientes, si se observan detenidamente las operaciones realizadas, podrá notarse que luego de cada intercambio, todas las filas y columnas conservan el mismo PCS, en este caso el 4-3.

154			2
	4578		
		78AB	
2			1AB

15	4		2
4	578		
		78AB	
2			1AB

1	45		2
45	78		
		78AB	
2			1AB

1	45		2
45	8	7	
	7	8AB	
2			1AB

1	45		2
45	8	7	
	7	8B	A
2		A	1B

1	45		2
45	8	7	
	7	8	AB
2		AB	1

La utilización en orden sucesivo de las matrices anteriores puede emplearse para la composición de un proceso gradual, como el que se muestra a continuación.

Figura 202. Aplicación musical de *swapping* de una matriz

OMMatrix

Una vez instalada la librería deberíamos verla en el menú de funciones, de forma similar al que se muestra en la figura.

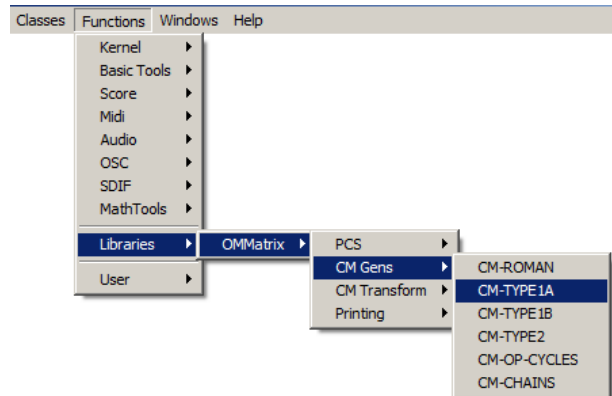
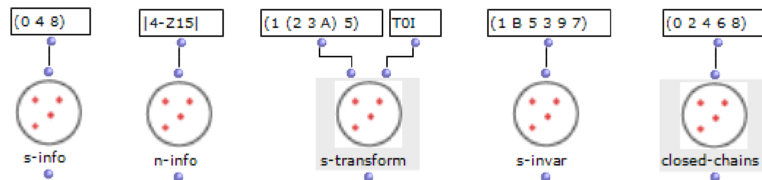


Figura 203. Menú de la librería *OMMatrix*

Operaciones con PCS

A la librería de matrices combinatorias hemos agregado algunos objetos relacionados con pitch class sets, a fin de integrarlos al entorno. A continuación describiremos sus características principales.

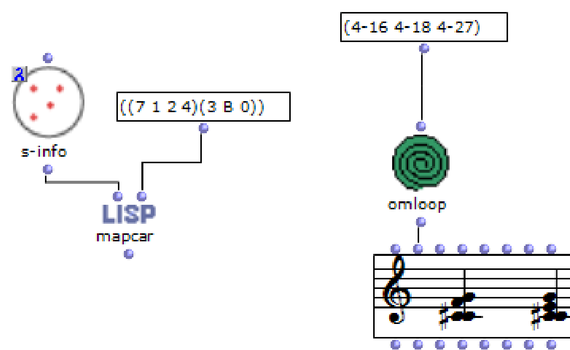


80 – ommatrix 1

Figura 204. Operaciones con PCS

s-info: dado un conjunto cualquiera en forma de lista, representado por números (0 a 11) o de forma alfanumérica (0 a 9, A y B), devuelve su nombre, su forma prima y su vector interválico.

n-info: brinda la misma información que el objeto anterior, pero a partir de especificar el nombre del conjunto. En el ejemplo de la figura siguiente, al ingresar el texto 4-z15 y al evaluar, la función devuelve la lista `(|4-Z15| (0 1 4 6) (1 1 1 1 1 1))` en la ventana de salida.



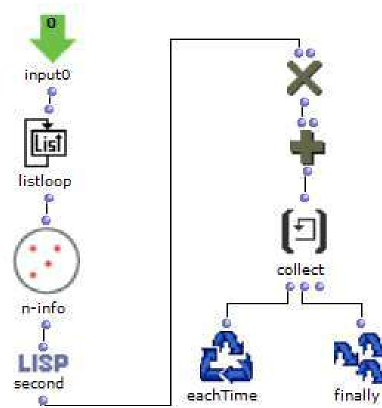
81 – ommatrix 2

Figura 205. `s-info` y `n-info`

El gráfico anterior, a la izquierda, muestra el uso de `s-info` aplicado a una lista de conjuntos, mediante `mapcar`. Obsérvese que la nota *si* está representada por *B* en lugar de un 11. Como es de rigor en estos casos, la función debe hallarse en modo *lambda*. El resultado expresa el nombre, la forma prima y el vector interválico de cada PCS:

```
OM => ((|4-13| (0 1 3 6) (1 1 2 0 1 1)) (|3-3| (0 1 4) (1 0 1
1 0 0)))
```

En el *patch* de la derecha obtenemos la forma prima de tres PCS definidos por sus nombres. En el interior del bucle recolectamos solamente la segunda sublista, que es la que contiene el dato buscado.



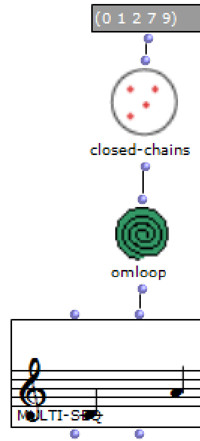
81 – ommatrix 2

Figura 206. Interior del bucle `omloop`

`n-transform`: aplica *Tn* o *TnI* a un conjunto. La lista de grados puede tener hasta un nivel de sublistas.

`s-invar`: devuelve la lista de invariantes por transposición e inversión de un conjunto dado. El conjunto `(1 B 5 3 9 7)` del ejemplo, exhibe invariancia para los operadores *T2, T4, T6, T8, TA, T0I, T2I, T4I, T6I, T8I, y TAI.*

`closed-chains`: calcula una lista de cadenas cerradas, producidas por medio de las particiones de un conjunto dado. Los conjuntos dato (norma de la cadena) deben tener 5 o 6 elementos, dado que son los más empleados en la construcción de matrices combinatorias mediante esta técnica. El PCS (0 1 2 7 9), por ejemplo, genera la siguiente lista resultado: `(((0 9) (1 2 7) (0 5) (3 4 10) (5 8) (3 9 10) (2 9) (0 1 7) (2 5) (0 6 7) (5 10) (3 8 9) (0 9)) ((1 2) (0 7 9) (2 8) (4 7 9) (2 3) (1 8 10) (1 7) (3 6 8) (1 2)))`. Según se observa, la lista comprende dos sublistas (dos cadenas generadas), y a su vez cada cadena está formada por sublistas que albergan a las particiones enlazadas. Obsérvese, además, que cada cadena comienza y termina con la misma partición –(0 9) en la primera cadena y (1 2) en la segunda– lo cual pone de relieve la igualdad entre comienzo y fin, propia de las cadenas cerradas.



82 – ommatrix 3

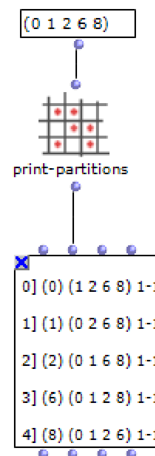
Figura 207.

Cadenas cerradas de PCS

Es importante aclarar que en este ejemplo mostramos los resultados con `multi-seq`, en forma de polifonía, a los efectos de comparar una cadena con otra, no para que suenen en conjunto. El contenido de `omloop` sirve simplemente para dar formato a la información obtenida, para poder representarla en notación musical.

Figura 208. Cadenas resultantes

`print-partitions`: imprime en una clase `textfile` las particiones de un PCS de 5 o 6 elementos.



82 – ommatrix 3

Figura 209.

Particiones de un PCS

Operaciones con matrices combinatorias

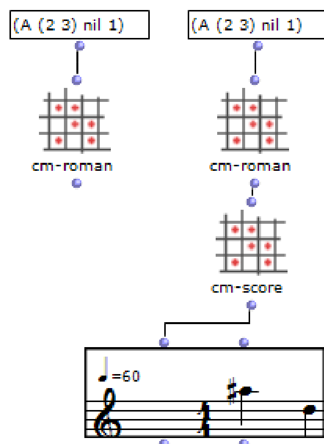
Los objetos vinculados con matrices se dividen de acuerdo a su función: generación, transformación e impresión.

Generación de matrices

cm-roman: construye una matriz del tipo cuadrado romano. El conjunto de partida puede tener solo un nivel de sublistas, ya que varios grados cromáticos pueden compartir la misma posición en la matriz. También puede contener el símbolo `nil`, que va a determinar una posición vacía. El conjunto `(6 (2 4) nil 8)`, por ejemplo, genera una matriz en la cual la primera fila está formada por el grado 6 en la primera columna, los grados 2 y 4 en la segunda columna, `nil` en la tercera y el grado 8 en la cuarta. El resto de la matriz, según ya sabemos, se genera a partir de permutaciones circulares de la primera fila. En el ejemplo que sigue generamos un cuadrado romano con el conjunto `(A (2 3) nil 1)`. Al evaluar `cm-roman` obtenemos:

```
OM => ((10 (2 3) nil 1) ((2 3) nil 1 10) (nil 1 10 (2 3)) (1
10 (2 3) nil))
```

Y mediante el uso de `cm-score` podemos representar el resultado en notación musical sobre la clase `poly`. Obsérvese que la notación rítmica obedece simplemente a la distribución de grados por posición, y no a la forma que deba tener en la composición.



83 – ommatrix 4

Figura 210.

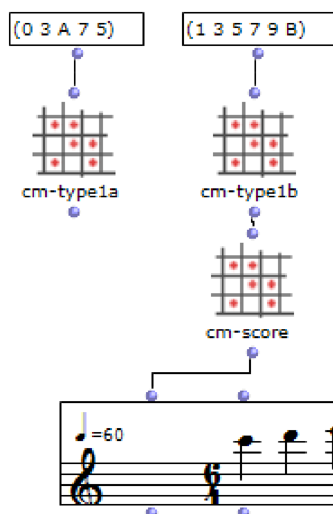
Cuadrado romano

Figura 211. Matriz en notación musical

cm-type1a: genera una matriz del tipo *1a*. La matriz debe tener un solo elemento por posición, por lo cual el conjunto de partida no debe contener sublistas. Ejemplo `(0 3 A 7 5)`, que da por resultado:

```
OM => ((0 3 10 7 5) (3 6 1 10 8) (10 1 8 5 3) (7 10 5 2 0) (5
8 3 0 10))
```

cm-type1b: genera una matriz del tipo *1b* a partir de un conjunto dado. Aquí también, la lista que especifica el conjunto dato no debe contener sublistas.

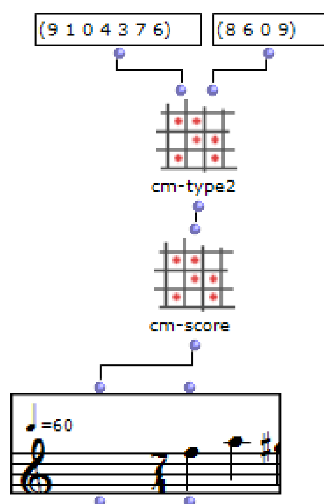


83 – ommatrix 4

Figura 212.

Matrices tipo *1a* y *1b*

cm-type2: genera una matriz del tipo 2 a partir de dos conjuntos, ambos especificados por medio de listas sin sublistas. El primer conjunto (1 9 B, en el ejemplo de la figura) determina la norma horizontal, mientras que el segundo conjunto (2 A 4 5) establece la norma vertical de la matriz.



83 – ommatrix 4

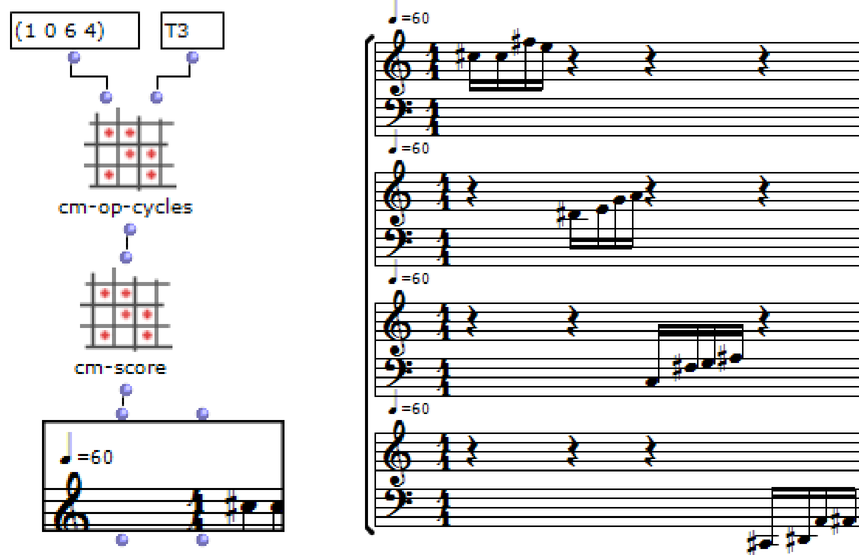
Figura 213.

Matriz tipo *2*

Figura 214. Matriz rectangular en notación musical

cm-op-cycles: genera una matriz por ciclos de un mismo operador. Para ello, utilizamos un conjunto de partida cuyos elementos ocupan la primera posición de la matriz (fila 1, columna1). Este conjunto de partida se especifica con doble paréntesis, dado que **todos sus grados cromáticos pertenecen a la misma posición.**

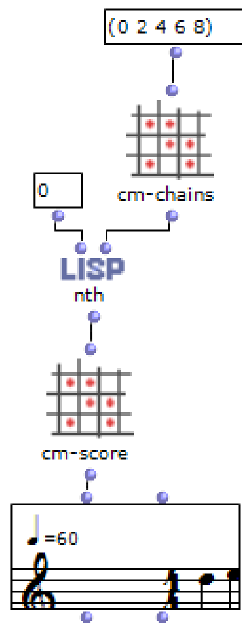
Ejemplo: `((1 7 A B))`. Como segundo argumento se debe determinar el operador de transformación que genera la matriz. Los operadores *T6* y *TnI* producen una matriz de 2x2; los operadores *T4* y *T8* una matriz de 3x3; los operadores *T3* y *T9* una matriz de 4x4 y los operadores *T2* y *TA*, una matriz de 6x6. En la figura que sigue se observa el empleo del operador *T9*, y la matriz cuyas dimensiones son 4x4.



84 – ommatrix 5

Figura 215. Matriz por ciclos de un operador

`cm-chains`: construye una serie de matrices por cadenas. El conjunto dato es el que servirá de norma de las particiones a generar, y debe declararse a través de una lista simple, por ejemplo `(0 1 2 7 9)`. Dado que el programa emplea todas las particiones útiles del conjunto especificado, las matrices generadas se ubican dentro una superlista que las contiene. Para acceder a cada una de ellas se puede recurrir al objeto `nth`, que devuelve la enésima sublista (matriz) deseada. La figura siguiente lo ilustra.



84 – ommatrix 5

Figura 216. Matriz por cadenas

La norma empleada en el ejemplo produce dos matrices distintas. Utilizando el argumento 0 en `nth` obtenemos la primera de ellas, mientras que con 1 obtenemos la segunda matriz. La cantidad total de matrices generadas por el objeto `cm-chains` se informa a través del *listener*.

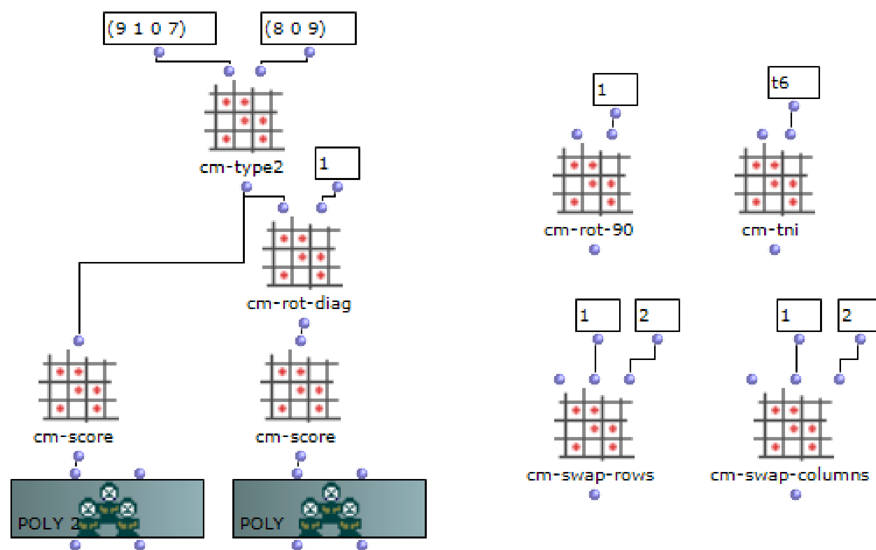
Transformación de matrices

Los siguientes objetos sirven para realizar transformaciones sobre las matrices generadas. Los procesos comprenden rotaciones, intercambio de elementos entre posiciones, transposiciones, inversiones, y conservación de elementos por invariancia.

`cm-rot-diag`: rota la matriz por sus diagonales. Se puede especificar la diagonal deseada a través de las opciones `1` o `2`.

`cm-rot-90`: rota la matriz 90 grados a la izquierda o a la derecha.

`cm-tni`: Transforma la matriz por *Tn* ó *TnI*.



85 – ommatrix 6

Figura 217. Operaciones de transformación

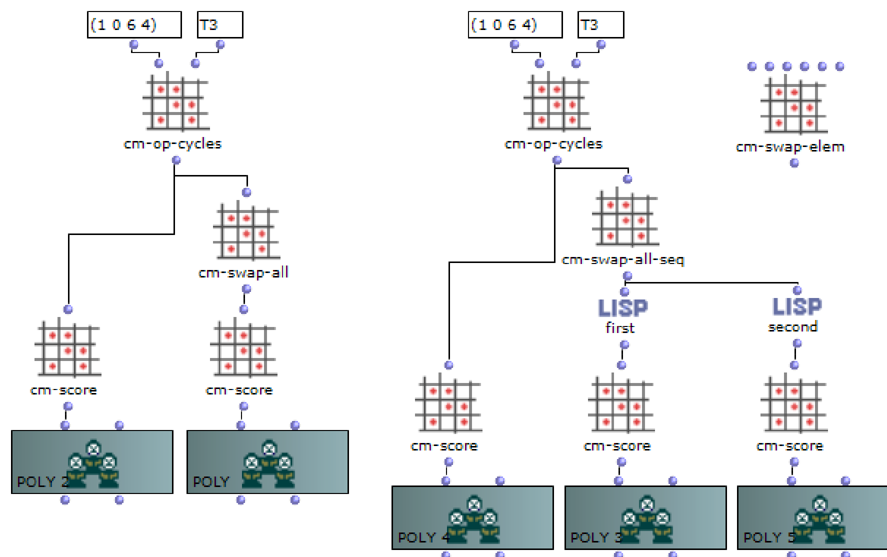
Figura 218. Filas por columnas en una matriz rectangular

cm-swap-rows: intercambia las filas de una matriz. Los argumentos 1 y 3, por ejemplo, determinan que la primera fila ocupe el lugar de la tercera, y viceversa.

cm-swap-columns: intercambia las columnas de una matriz.

cm-swap-elem: intercambia elementos iguales entre posiciones. Aquí se debe determinar el elemento (grado cromático a intercambiar), la fila y la columna en la que se encuentra el primer elemento, y la fila y la columna del segundo elemento.

cm-swap-all: intercambia elementos entre posiciones hasta obtener una distribución lo más homogénea posible en la matriz. En la figura siguiente se observa una matriz construida mediante ciclos de un operador, en la cual los grados cromáticos se ubican en la diagonal. Luego de la transformación puede verse la distribución alcanzada.



86 – ommatrix 7

Figura 219. Operaciones de *swapping*

cm-swap-all-seq: intercambia elementos entre posiciones hasta obtener una distribución homogénea en la matriz, pero en este caso, guardando las matrices generadas a lo largo del proceso de *swapping*. El resultado es una lista de matrices, por lo cual, para acceder a cada una de ellas se deben utilizar los objetos `first`, `second`, `third`, etcétera, o bien `nth`.

Figura 220. Matriz antes y después del *swapping* con `cm-swap-all`

`cm-invar-position`: genera una serie de matrices transpuestas en las cuales sólo una posición permanece invariante. La posición elegida se determina especificando en el objeto el número de fila y el número de columna. Para acceder a cada una de las matrices de la serie debemos usar `first`, `second`, u otros objetos de selección.

`cm-invar-row`: genera una serie de matrices transpuestas en las cuales sólo la fila especificada permanece invariante.

`cm-invar-column`: genera una serie de matrices transpuestas en las cuales sólo la columna especificada permanece invariante.

Impresión de matrices

Los siguientes objetos sirven a la representación gráfica de las matrices combinatorias.

Las funciones **`cm-score`** y cm-print-partitions ya fueron utilizadas en los ejemplos anteriores. Por último, la función **`cm-norm-print`** imprime en el *listener* los PCS que conforman la norma horizontal y vertical de la matriz combinatoria.

Bibliografía

Ariza, C. Two Pioneering Projects from the Early History of Computer-Aided Algorithmic Composition. Computer Music Journal 35 (3), pp. 40-56. 2011. Consultado en http://hdl.handle.net/1721.1/68626.

Assayag, G., Rueda, C., Laurson, M., Agon, C., y Delerue, O. Computer-Assisted Composition at IRCAM: From PatchWork to OpenMusic, Computer Music Journal, vol.23 (3). 1999.

Cetta, Pablo. *Captura y proceamiento de sonido.* Bernal. Universidad Virtual de Quilmes. 2014.

Cetta, Pablo y Di Liscia, Oscar. *Elementos de Contrapunto atonal.* Buenos Aires. Educa. 2009.

Gabriel, R., White, J. L., y Bobrow, D. G. CLOS: Integration Object-oriented and Functional Programming. Communications of the ACM, 34 (9). 1991.

Hiller, L. and Isaacson, L. Musical composition with a high-speed digital computer. Journal of Audio Engineering Society 6, pp. 154-160. 1954.

Kligbeil, M. Software for spectral analysis, editing, and synthesis. Proceedings of the International Computer Music Conference 2005. Barcelona. pp. 107-110. 2005.

Laurson, M. y Duthen, J. Patchwork, a Graphic Language in PreForm. En Proceedings of the International Computer Music Conference, Ohio State University, USA. 1989.

Lovelace, Ada Augusta. Notes by A.A.L. [Augusta Ada Lovelace]. *Taylor's Scientific Memoirs,* Londres, vol. III, 1843, pp. 666-731.

Morris, Robert. Combinatoriality without the aggregate. Perspectives of New Music, Vol. 21 No. 1-2, pp. 432-486. 1983.

Puckette, Miller. Prefacio. En C. Agon, G. Assayag, and J. Bresson, editores, *The* OM *Composer's Book 1.* Paris. Delatour/IRCAM. 2006.

En 1955 comienzan las primeras experiencias sobre la utilización de las computadoras en música. Ya desde los inicios, se perfilan dos campos de aplicación diferenciados: el de la composición asistida, por un lado, y el de la síntesis y el procesamiento de señales de audio, por otro. En ambos casos, las necesidades creativas no solamente exigen el desarrollo de aplicaciones informáticas, sino también, de lenguajes especialmente diseñados para estos usos. Este libro propone el estudio y el análisis de la formalización de procedimientos aplicables a la composición musical y su codificación a través de la programación con LISP y OpenMusic.