Facultad de Artes y Ciencias Musicales

Composición Asistida por Computadora con PWGL

Pablo Cetta

Seminario de Composición Asistida
y Procesamiento en Tiempo Real de Sonido y Música

Doctorado en Música – Área Composición

2010

ÍNDICE

INTRODUCCIÓN

Este escrito trata la utilización de la computadora como herramienta asistente en la composición musical. A fin de ejemplificar la formalización de los procedimientos a analizar vamos a utilizar el lenguaje PWGL.

PWGL es un lenguaje visual orientado al desarrollo de aplicaciones de composición asistida y síntesis del sonido. Fue creado en la Sibelius Academy de Finlandia por Mikael Laurson, Mika Kuuskankare, Vesa Norilo y Kilian Sprotte, y está basado en Patchwork[1], Common LISP y OpenGL[2]. Tanto el programa, como su documentación, pueden descargarse de la siguiente URL:

http://www2.siba.fi/pwgl/downloads.html

Las versiones disponibles son para los sistemas operativos Macintosh OS X 10.4-10.6 Intel, Macintosh OS X 10.4-10.5 PPC y Windows XP (SP2). No obstante, por tratarse de un programa con código abierto, puede compilarse para otros sistemas.

Utilizaremos, además, objetos de la librería FLAT, que se encuentra disponible en:

http://www.umanitoba.ca/studio-flat/download.php

Se adjunta al presente escrito un archivo con todos los ejemplos citados, y con copia de las descargas.

Dado que PWGL es un entorno de programación creado en LISP, comenzaremos analizando algunos conceptos básicos de este último lenguaje.

Creo oportuno aclarar que un mejor aprovechamiento de las posibilidades de PWGL requiere profundizar el conocimiento de LISP a través de bibliografía especializada.

[1] http://www2.siba.fi/soundingscore/PWHomePage/patchwork.html
[2] http://www.opengl.org/

CONCEPTOS BÁSICOS DE 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. Se trata de un lenguaje de alto nivel, cuya sintaxis se basa en listas (secuencias de elementos contenidas entre paréntesis, separados por espacios). La versión de mayor difusión en esa época fue *LISP 1.5*, publicada por el mismo McCarthy [3].

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 se denomina *ANSI Common LISP*, que fue diseñada originalmente por Guy Steele[4].

Listas

El término LISP es acrónimo de *List Processing*, con lo cual podemos inferir la importancia que tienen las listas en este lenguaje. Una lista está formada por elementos, y a la vez puede contener a otras listas anidadas.

```
(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.

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)`.

Eval

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. A fin de comprobar nuestros ejemplos podemos emplear, entre otros, el entorno de programación *LispWorks*[5]

La forma de escribir expresiones en LISP se denomina notación LISP, o notación EVAL. Una expresión típica es `(+ 5 6)`, en la cual vemos una lista con tres elementos. El primero hace referencia a la función suma, y los otros dos representan los números a sumar. Si evaluamos esta lista, obtendremos un 11 como resultado.

Otro ejemplo de notación LISP es `(* 3 (+ 5 6))`. Aquí vemos una lista dentro de otra, que indica que primero se debe resolver la operación de suma, y luego la multiplicación, en función del resultado de la suma. Al evaluar, obtenemos un 33 como resultado.

---

[3] McCarthy, John, Abrahams, Paul W., Edwards, Daniel J., Hart, Timothy P., and Levin, Michael I., *Lisp1.5 Programmer's Guid*e, 2nd ed., MIT Press, Cambridge, MA, 1965.
[4] Steele, Guy et al. *Common Lisp, the Language, 2nd Edition*. Digital Press, Bedford, MA. 1990.
[5] *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.

Tipos de datos

Algunos tipos de datos son:

Números: los tipos de datos numéricos son *integer* (entero), *ratio* (fracciones), *floating-point* (decimales) y *complex* (complejos).
14 es entero
-67.32 es decimal
3/8 es fraccionario
#c(2 5) es complejo, con parte real (2) e imaginaria (5).

Caracteres: se especifican con la forma #c\a (carácter *a*).

Símbolos: son secuencias de letras, dígitos y caracteres especiales. Los siguientes son caracteres especiales válidos: +, -, *, /, @, $, ^, &, \, <, >, . La palabra RUEDA, por ejemplo, es un símbolo. LISP traduce todas las palabras en minúsculas a mayúsculas, pues no hace distinción entre ambos tipos.

Listas: pueden contener números, símbolos, u otras listas.

Otros tipos de datos son ***Arrays*** (matrices), **Vectores** (matrices unidimensionales), ***Strings*** (cadenas de caracteres), ***hash tables*** (estructuras preparadas para la búsqueda rápida de información).

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 asignar, en algún lugar de nuestro programa, el nombre de la variable (el símbolo) a un contenido específico. De otro modo, el intérprete nos dará un mensaje de error por desconocer a esa variable.

Existe un modo de evitar la evaluación de una lista, que consiste en colocar un apóstrofe (ALT + 39) delante de ella. Si escribimos `'(A B C)`, el intérprete devuelve `(A B C)`, es decir, la lista sin evaluar. Con esto evitamos que el primer elemento de la lista sea considerado como el nombre de una función, y los demás elementos como sus argumentos. Si, en cambio, escribimos directamente en el intérprete `(A B C)`, éste nos devolverá un mensaje de error diciendo que A no es una función válida.

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 ENTER, el intérprete indicará un error, pues no conoce a la función UNO. Para obtener el segundo elemento de la lista es necesario evitar su evaluación, y que sea considerada como argumento de la función *second*. Debemos colocar, entonces, el apóstrofe delante de ella, y escribir `(second '(uno dos tres))`. De este modo, obtendremos DOS como respuesta.

Funciones

El intérprete LISP trae incorporada una gran cantidad de funciones (*built-in*). Entre ellas, para realizar operaciones aritméticas, funciones trigonométricas, operaciones de control del flujo de un programa, condicionales, operaciones con listas de datos, operaciones de entrada y salida de datos, etc.

`(max 3 5 2 1 7 10)`, por ejemplo, devuelve 10, que es el valor numérico máximo que se encuentra en la lista.

`(list a b c d)` crea una lista a partir de los cuatro elementos establecidos como argumentos, y devuelve `(A B C D)`.

Los **predicados** son funciones que comprueban la veracidad o falsedad de un enunciado. Si queremos saber si un número es positivo, podemos preguntarlo escribiendo `(plusp -4.3)`; la respuesta es, obviamente, *NIL*. Otros predicados son *evenp* (para saber si un número es par), *zerop* (si el número es cero), o los símbolos >, <, =, /= (este último símbolo significa distinto).

Una **macro** es un tipo de sentencia que al expandirse se transforma en función. En las macros los argumentos no se evalúan. Un caso típico de macro es *defun*, que nos permite crear nuestras propias funciones. Si escribimos `(defun cuadrado (n) (* n n))`, el intérprete responde con el nombre de la nueva función creada: CUADRADO. La letra *n* refiere a una variable. Luego, si escribimos `(cuadrado 5)`, habremos asignado a *n* el valor 5, y obtendremos 25 como resultado.

Pero qué sucede si queremos aplicar la nueva función *cuadrado* a todos los elementos de una lista. Si escribimos `(square '(1 2 3 4 5))` el interprete nos dará error, pues nuestra función sólo acepta un número como argumento. En ese caso debemos recurrir a una nueva función que aplique *cuadrado* a todos los elementos de la lista, y esa función es *mapcar*.

`(mapcar #'cuadrado '(1 2 3 4 5))` devuelve `(1 4 9 16 25)`

Observamos una combinación de símbolos `#'` antes del nombre de la función a aplicar, lo cual tomaremos como una exigencia de la sintaxis del lenguaje.

Condicionales

Los condicionales son macros que evalúan una serie de sentencias, hasta encontrar una que sea verdadera. Si la encuentra, evalúa a las demás sentencias que la acompañan a esa hallada. En el siguiente ejemplo, vemos la definición de una función que devuelve el nombre del país al que pertenece una ciudad. La función *equal* devuelve *T* si dos símbolos son iguales. Si desconoce la ciudad responde con un DESCONOZCO.

```
(defun donde-queda (x)
      (cond ((equal x 'paris) 'francia)
            ((equal x 'londres) 'inglaterra)
            ((equal x 'beijing) 'china)
             (t 'desconozco)))
```

El intérprete devuelve DONDE-QUEDA, informando que ya reconoce a nuestra función. Si luego escribimos

`(donde-queda 'paris)`, el programa informa FRANCIA.

Se observa en la función que la última opción es **t** (*true*). Mediante este artilugio, si la función no encontrara en sus opciones la ciudad ingresada como dato, se fuerza una respuesta verdadera (en este caso DESCONOZCO).

Otra manera de establecer condiciones es 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)` devuelve IMPAR.

Variables

Las variables se clasifican en **globales** y **locales,** y ambas permiten el almacenamiento de datos y su posterior modificación. Es posible acceder al contenido de una variable global desde cualquier lugar de un programa. Las variables locales, en cambio, sólo son accesibles en el ámbito en el que fueron creadas.

En el siguiente ejemplo definimos una variable global con el nombre *frecuencias*, y le asignamos una lista con tres elementos en su interior. Para ello usamos la función *defparameter*, o también *defvar*. Los nombres de variables globales suelen encerrarse entre asteriscos, con el propósito de diferenciarlas claramente de las variables locales.

```
(defparameter *frecuencias* '( 100 200 300))
```

A continuación, definimos dos variables locales (*nota* e *intervalo*) con *let*. El acceso a su contenido sólo es posible dentro del bloque de paréntesis en el que son creadas (se observa su empleo dentro de los paréntesis que contienen a *let*). El programa devuelve 8 como resultado.

```
(let ((nota 3)
      (intervalo 5))
     (+ nota intervalo))
```

Funciones lambda

Se denomina *lambda* a un tipo de función que no tiene nombre propio, y que es creada para un uso temporario, o como argumento de otra función.

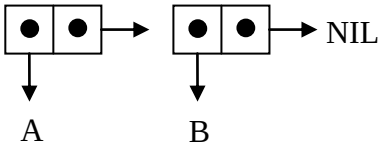
```
(mapcar #'(lambda (n) (* n 2)) '(1 2 3 4 5))
```

que devuelve `(2 4 6 8 10)`.

CAR y CDR

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 (*Contents of Adress portion of Register*) y la segunda CDR (*Contents of Decrement portion of Register*), y ambas toman sus nombres de los registros de una primitiva computadora donde corría LISP en sus orígenes, la IBM 704. 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)` posee dos células *cons*, dispuestas de la siguiente manera:



Además de tratarse de punteros, CAR y CDR son funciones primitivas 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)`, obtenemos nuevamente `(UNO DOS TRES)`.

EL ENTORNO PWGL

En el libro *PWGL Book*, de Mikael Laurson y Mika Kuuskankare, que se adjunta a la documentación de PWGL, encontraremos información detallada sobre las características de este programa (menús, configuración, *shortcuts*, preferencias, instalación de librerías, etc.) y un *tutorial* con ejemplos de aplicación interesantes.

Realizaremos, no obstante, una breve reseña de los aspectos más salientes, que nos permita comenzar a trabajar en la programación.

Al iniciar el programa PWGL observamos dos ventanas. La que lleva por título *PWGL 1*, es la ventana de edición, donde programaremos a través de objetos interconectados mediante cables virtuales. En la otra ventana, *PWGL Output*, recibiremos los resultados de evaluar las funciones de nuestra aplicación.

Para ubicar objetos en la ventana de edición basta con hacer un *click* con el botón derecho del *mouse* sobre el fondo de la ventana. Veremos que aparece un menú con distintas opciones, y dentro de cada opción hallaremos a los objetos.

Para evaluar un objeto lo seleccionamos con el *mouse*, y luego presionamos la tecla "v". Como dijimos antes, el resultado de esa evaluación aparece en la ventana *PWGL Output*. Si se trata, en cambio, de una cadena de objetos y queremos observar el resultado final, simplemente seleccionamos el último objeto de esa cadena y presionamos "v".

Podemos agrandar o achicar la apariencia de nuestra aplicación haciendo girar la rueda del *mouse* en un sentido o en otro. Esto sirve también para los editores gráficos (2D *editor*, *chord editor*, *score editor*). Un doble *click* sobre cualquier editor lo abre para apreciar o modificar su contenido.

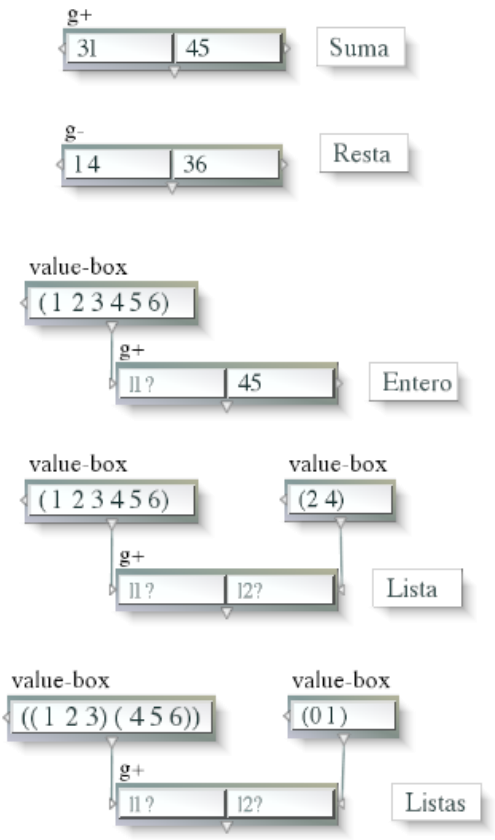
La alineación de los objetos en pantalla se consigue seleccionándolos con el *mouse*, y luego presionando "x" o "y", ya sea para una alineación vertical u horizontal, respectivamente. Para conducir los cables, hacemos *click* con el botón derecho sobre uno de ellos y elegimos una opción en el menú que se despliega. Las opciones son *line* (cable recto), *5point* (doblado en ángulos rectos) o *bezier* (doblado en curvas que pueden modificarse).

PROGRAMACIÓN EN PWGL

Operaciones aritméticas

Para comenzar con el uso de los objetos, veremos algunas operaciones con datos numéricos. Para acceder a la lista de objetos presionamos el botón derecho del mouse sobre el fondo de la ventana de edición. Los objetos a emplear se encuentran en la opción *Arithmetic*.

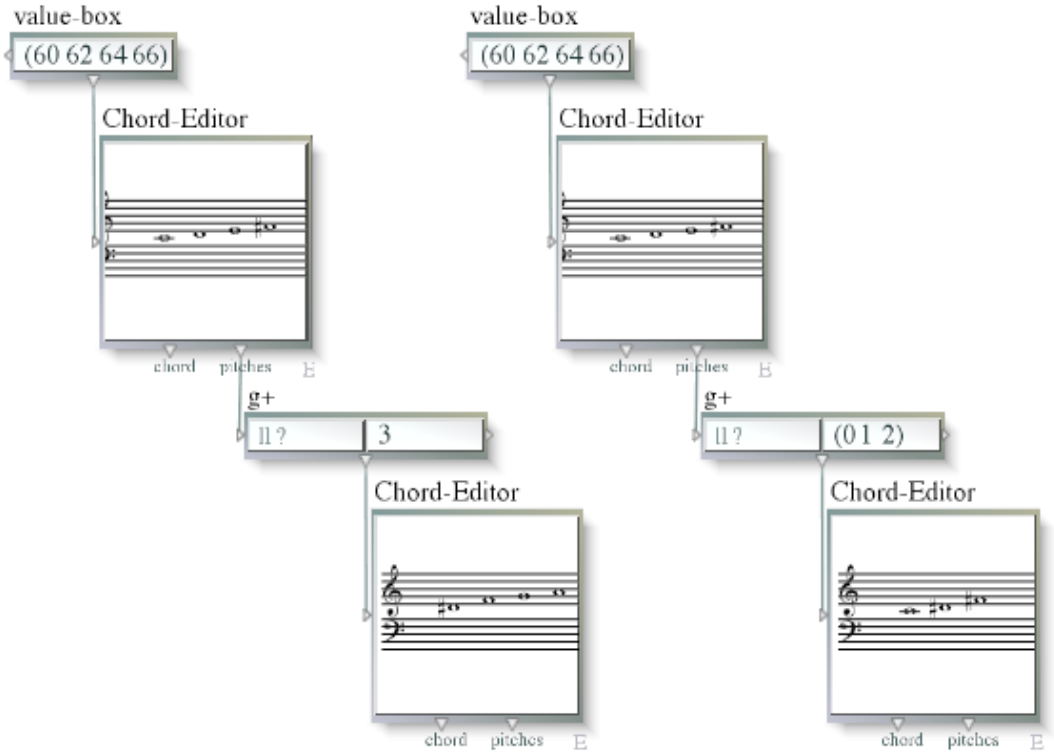
En el primer ejemplo (figura 01) utilizamos los objetos *g+* y *g-* para obtener respectivamente la suma y la resta de dos números, la suma de una lista de números con un entero, la suma de dos listas, y la suma de una lista de listas con otra lista. Podemos evaluar las operaciones seleccionando cada objeto *g+* o *g-*, y presionando luego la tecla "*v*". Según vimos antes, los resultados se observan en la ventana *PWGL output*. Vemos, además, que los datos pueden escribirse directamente en el objeto, o externamente, valiéndonos de un *value-box* (que se encuentra en la opción *Data*).



01-Operaciones aritméticas

Notamos también que las listas son conjuntos de datos, en estos casos numéricos, encerradas entre paréntesis. Una lista puede contener a otras listas en su interior, por ejemplo
`(3 2 1 (5 8) (0 4 (1)))`, que forma una estructura de árbol.

Dentro de la opción *Editors* encontramos un objeto denominado *Chord-Editor*. Mediante un *value-box* crearemos una lista de notas MIDI conectada al editor. Al evaluar el *Chord-Editor* es posible observar esa lista representada en notación musical. También podemos editar el contenido si hacemos doble clic sobre el objeto de edición; entre las opciones de edición, elegir si el acorde se muestra arpegiado o no. En el ejemplo siguiente (02), aplicamos *g+* para obtener la transposición del acorde, primero a distancia de una tercera menor (3), y luego para transportar las primeras tres notas según la lista `(0 1 2)`.

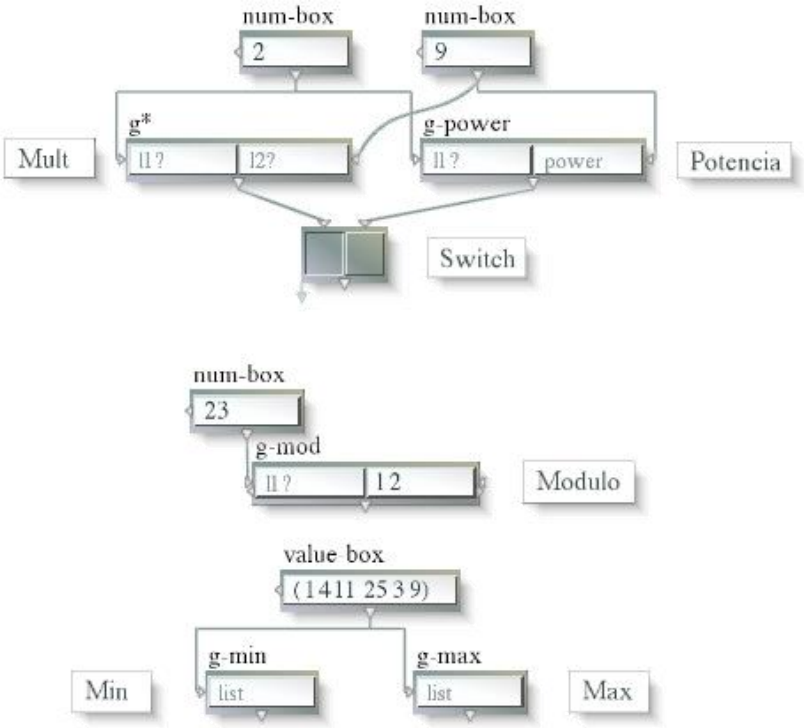


02-Chord Editor

En el primer *patch* de los que siguen (figura 03), observamos los objetos *g** (multiplicación), *g-power* (potenciación) y *pwgl-switch* (que se halla en la opción *Control*). Este último, permite seleccionar el flujo de datos de alguna de sus entradas mediante botones, y es un objeto redimensionable. Para aumentar la cantidad de botones lo seleccionamos y presionamos la tecla "+". Observamos, en general, que los objetos redimensionables muestran una flecha en la parte inferior de su representación gráfica.

Para el segundo ejemplo, utilizamos el objeto *g-mod*, que devuelve el módulo *n* del número o lista en su entrada. Una aplicación típica es calcular el módulo 12 de una lista de notas MIDI, a fin de obtener los grados cromáticos (de 0 a 11) de esas notas.

En el tercer ejemplo obtenemos el valor mínimo y el máximo de una lista de enteros, con los objetos *g-min* y *g-max*.



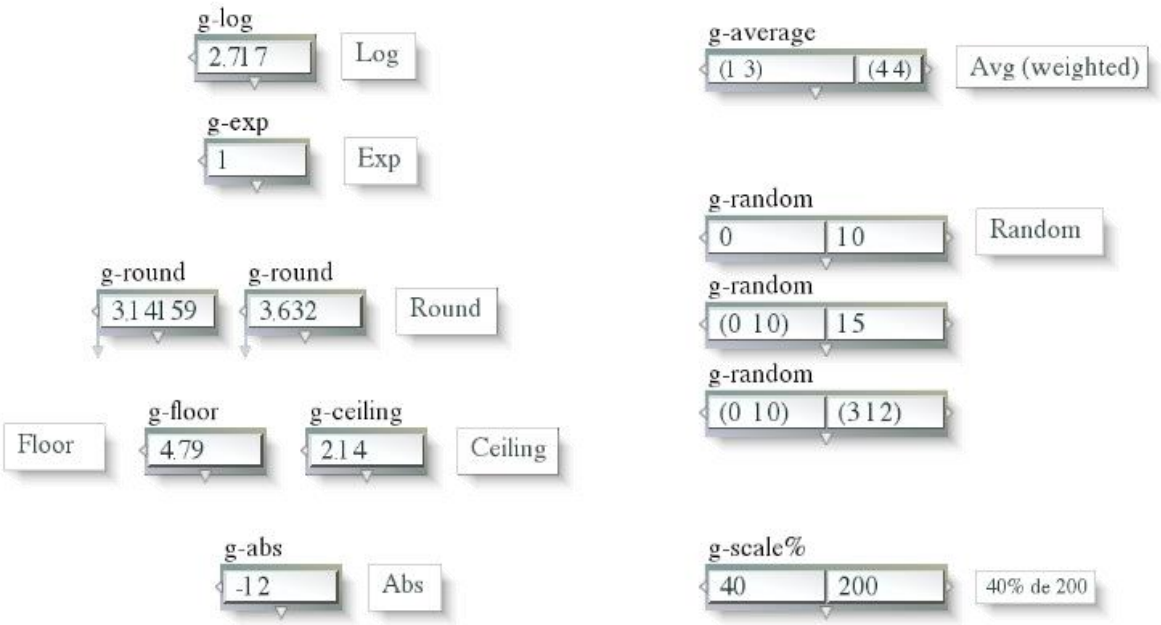
03-Otras operaciones

Otros objetos, representados en la figura 04, son *g-log* (logaritmos en base *e*), *g-exp* (*e* elevado a un exponente), *g-round* (redondeo de un número con decimales al entero más próximo), *g-floor* (redondeo al entero más bajo), *g-ceiling* (redondeo al entero mayor), *g-abs* (valor absoluto) y *g-scale%* (porcentaje de un valor dado).

g-average calcula el promedio ponderado de un conjunto de datos (x_1, x_2, ..., x_n). Este promedio se determina a través de la siguiente fórmula, en la cual *w* es el peso de cada elemento:

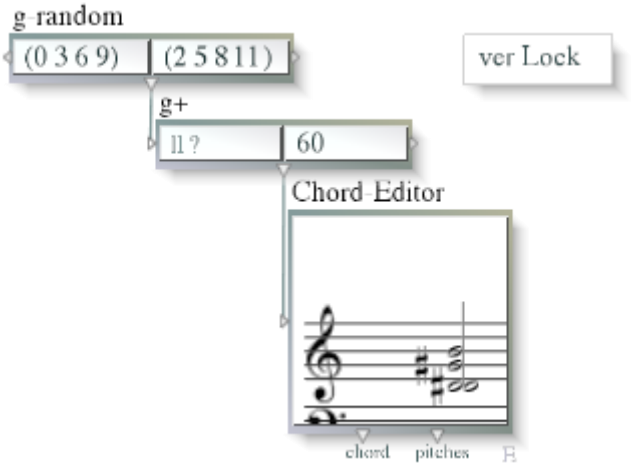
$$\bar{x} = \frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i},$$

Con *g-random* generamos números al azar. Este objeto arroja un número comprendido entre un mínimo y un máximo, o bien una lista de números al azar cuyos mínimos y máximos se especifican también en forma de lista.



04-Más operaciones

A continuación (ejemplo 05) utilizamos el objeto *g-random* para crear un acorde a partir de sus grados cromáticos, y luego sumamos 60 a cada grado (*do* central en MIDI) para representarlo en notación musical. Si seleccionamos *g-random*, y presionamos el botón derecho sobre el objeto, se despliega un menú. Allí podemos elegir la opción *toggle lock on/off*, con lo cual bloqueamos al objeto para que siempre devuelva los mismos valores. Cuando bloqueamos un objeto aparece una pequeña llave en su representación gráfica.



05-Random

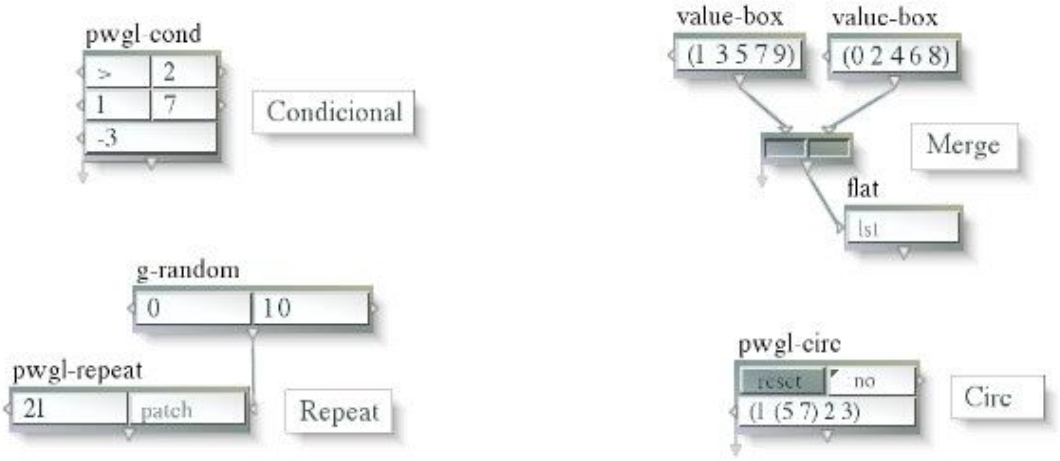
Operaciones de control

El objeto *pwgl-cond* determina si se cumple o no una condición determinada. Por ejemplo, si un número es mayor que otro, menor, igual o distinto. Si la afirmación es verdadera devuelve un valor, caso contrario devuelve otro. En el ejemplo que sigue, decimos que 2 es mayor que 1; como la afirmación es verdadera el objeto devuelve 7, pero si esa afirmación fuera falsa, el objeto devolvería -3. Este objeto es redimensionable y permite hacer comparaciones múltiples, y si no se cumple ninguna condición, devuelve el último valor.

Utilizamos el objeto *pwgl-merge* para unir dos o más listas. En nuestro ejemplo, unimos la lista `(1 3 5 7 9)` a la lista `(0 2 4 6 8)`. El resultado es `((1 3 5 7 9) (0 2 4 6 8))`, o sea otra lista que incluye a las dos anteriores. El objeto *flat* (que se encuentra en la opción *List*) sirve para eliminar los paréntesis internos, vale decir, devuelve una única lista sin sublistas.

El objeto *pwgl-repeat* repite la evaluación de la cadena de objetos unidos a él una cantidad especificada de veces. En el ejemplo, produce una lista con 21 números al azar comprendidos entre 0 y 10.

Por último, el objeto *pwgl-circ* devuelve, a cada evaluación, las permutaciones circulares de los elementos o sublistas de una lista.



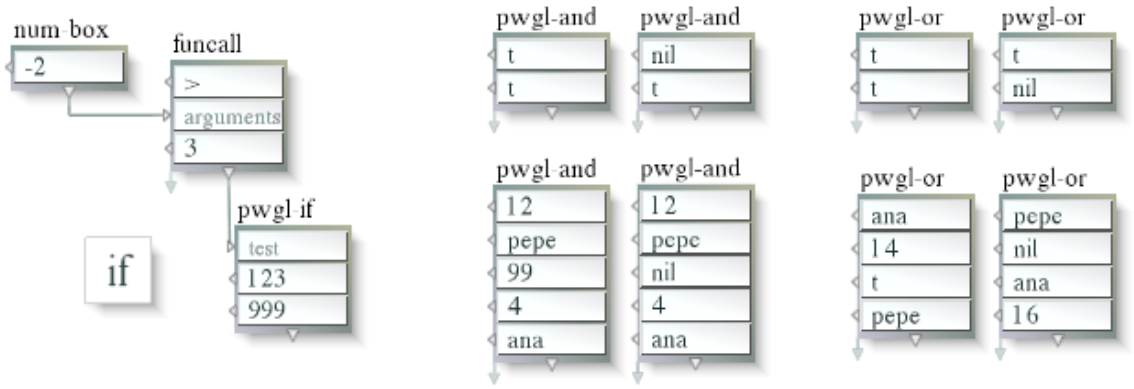
06-Control

En el *patch* siguiente vemos el empleo de *pwgl-if*, que es un condicional. En su primera entrada ingresa el resultado de una evaluación, la cual puede ser verdadera (*T*) o falsa (*nil*). Si ingresó *T*, el objeto *if* evalúa lo conectado a su segunda entrada, y si ingresó *nil* evalúa su tercera entrada. En nuestro caso, queremos saber si el número -2 es mayor que 3, y para esto recurrimos al objeto *funcall*, que llama a la función ">" de LISP. Dado que la evaluación da *nil*, el objeto *if* devuelve el valor 999 (ver figura 07).

El objeto *pwgl-if* no sólo devuelve resultados numéricos, sino que también permite controlar el flujo de información de un programa en un sentido o en otro. En estos casos, en lugar de ingresar números en su segunda y tercera entradas, podemos conectar *patches*. Lo mismo ocurre con *pwgl-cond*, que vimos anteriormente.

El objeto *pwgl-and* evalúa cada una de sus entradas siguiendo el orden de arriba hacia abajo. 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.

El objeto *pwgl-or* evalúa también 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*.



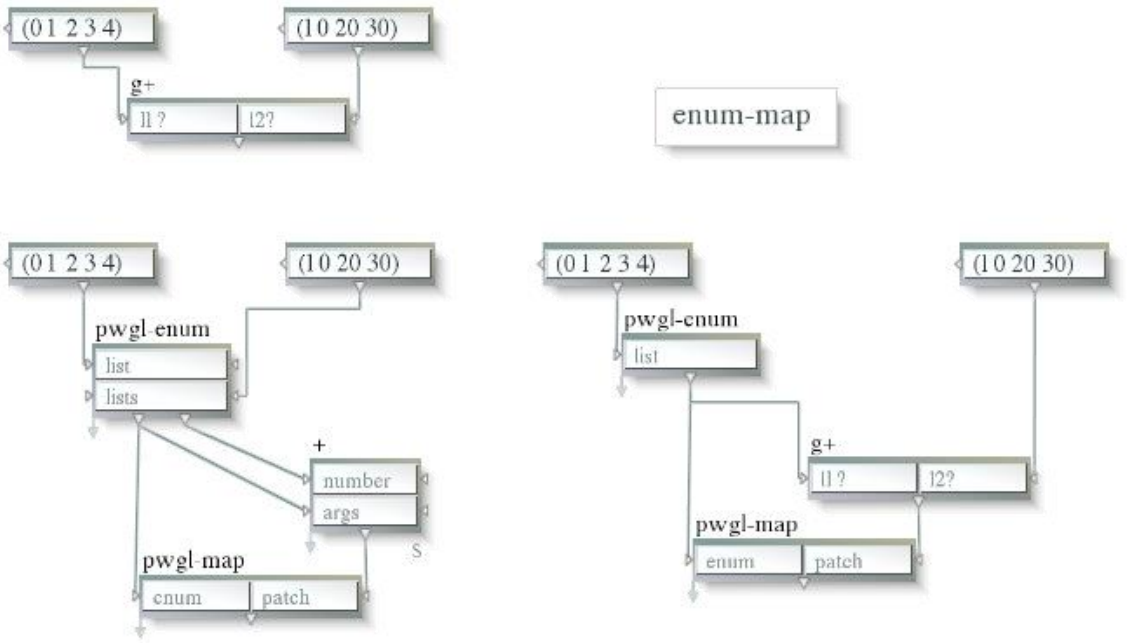
07-Control

Vimos, al tratar el objeto g+, que éste puede sumar dos números, un número a una lista, o dos listas. Esto difiere de la simple función "+" de LISP, que sólo suma una cantidad dada de números. Las funciones propias de LISP pueden incorporarse a PWGL, haciendo doble clic sobre el fondo de la ventana y escribiendo el nombre de la función que deseamos utilizar en el cuadro de diálogo que aparece.

En el ejemplo siguiente (figura 08) vamos a utilizar la función LISP "+" en un programa que nos permita sumar dos listas, al igual que el objeto *g+*. Para esto es necesario contar con un objeto que extraiga uno por uno los elementos que conforman cada lista, para que podamos sumarlos por separado, y coleccionarlos en una nueva lista que presente el resultado. En realidad se trata de dos objetos, que se llaman respectivamente *pwgl-enum* y *pwgl-map*.

Pwgl-enum es redimensionable y puede extraer los elementos de la cantidad de listas que deseemos. La cantidad de elementos que extrae es igual a la cantidad de elementos de la lista de menor longitud. En nuestro ejemplo, la lista más corta posee 3 elementos `(10 20 30)`, que se van a sumar uno a uno con los primeros tres de la otra lista `(0 1 2)`. *Pwgl-map* recolecta los resultados de cada suma y los ubica en una nueva lista `(10 21 32)`.

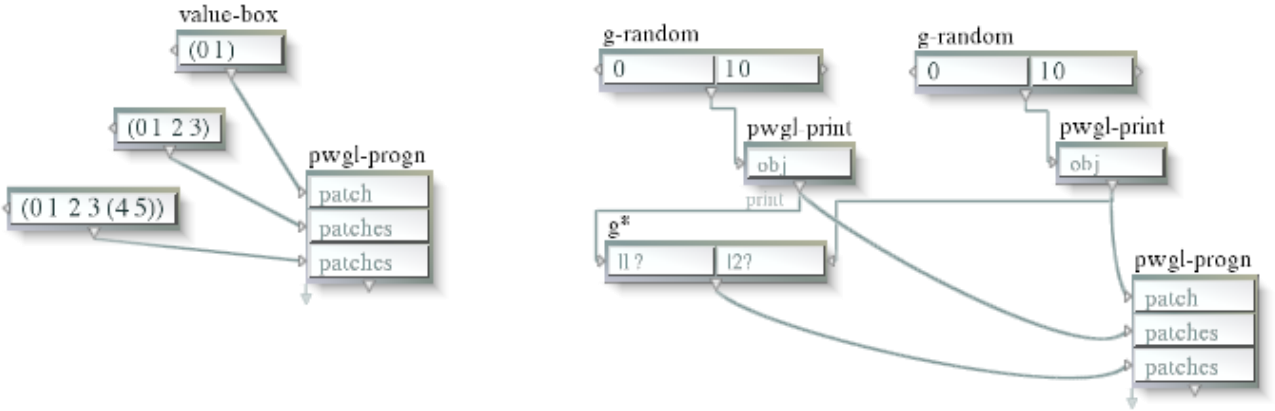
En el *patch* de la derecha de nuestro ejemplo sucede algo parecido, pero en este caso enumeramos los elementos de la primera lista y los sumamos a la segunda lista completa (no enumerada) con *g+*. El resultado en este caso es `((10 20 30) (11 21 31) (12 22 32) (13 23 33) (14 24 34))`.



08-Control

Otro objeto de control es *pwgl-progn*, que evalúa en orden cada una de sus entradas y devuelve el resultado de evaluar la última de ellas. En el ejemplo 9, a la izquierda, evaluamos tres listas, y obtenemos como resultado la última lista `(0 1 2 3(4 5))`.

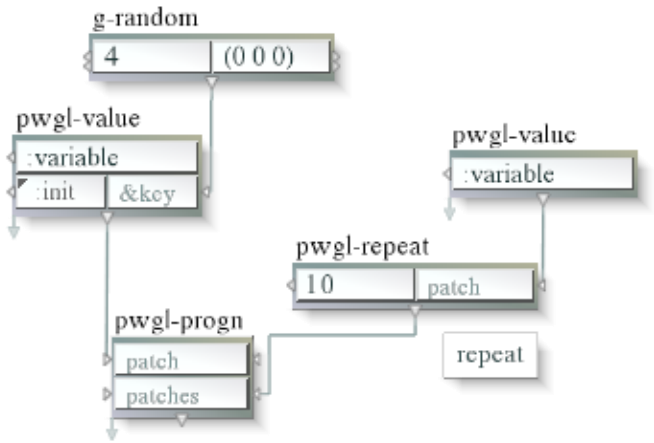
En el *patch* de la derecha, evaluamos la impresión de un número al azar comprendido entre 0 y 10, luego la de otro número al azar (que también se imprime en la ventana *PWGL output* con *pwgl-print*), y finalmente el producto de ambos números. Pero qué es lo que realmente sucede. Cuando seleccionamos *pwgl-progn* y presionamos la tecla "*v*", aparecen 4 números impresos y luego el resultado de la multiplicación de los dos últimos. Según se observa, de los objetos *pwgl-print* salen dos cables, lo cual significa que ese objeto y todo lo que está por encima de ellos (en este caso *g-random*) es evaluado dos veces, una por *g** y la otra por *pwgl-progn.* El objeto que está encima de *pwgl-print* genera números al azar cada vez que se lo evalúa, y eso explica el por qué de la diferencia que encontramos impresa. Debemos prestar especial atención cuando salen dos o más cables de un mismo objeto, a fin de evitar errores de programación.



09-Control

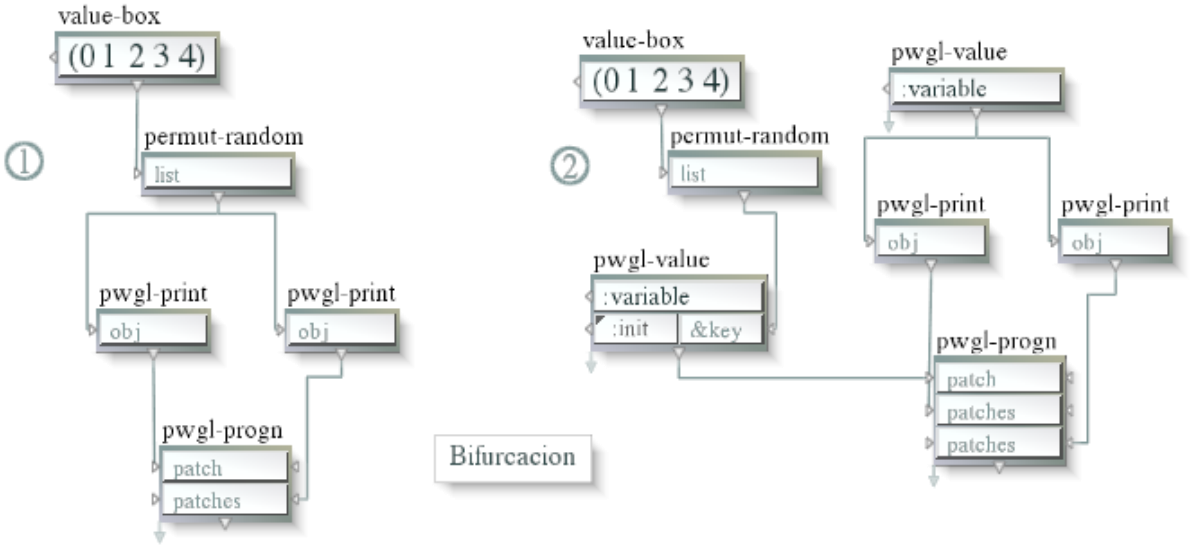
Por lo expuesto anteriormente, se torna necesario contar con la posibilidad de almacenar los datos que devuelve un objeto para usarlos repetidamente, sin que esto implique una nueva evaluación de ese objeto. En el *patch* siguiente vamos a almacenar una lista de tres números al azar generados por *g-random*, utilizando un objeto denominado *pwgl-value*. Este objeto es extensible, y en su primer campo escribimos (o ingresamos por su entrada) un nombre de variable precedido por el símbolo":" (en nuestro ejemplo, :*variable*), en la segunda entrada colocamos el valor inicial, y mediante la tercera podemos asignar nuevos valores a nuestra variable.

El objeto *pwgl-progn* efectúa en primer término la evaluación de *g-random* y la inicialización de *pwgl-value*. Posteriormente, evalúa el resto del *patch*, tomando los datos almacenados en la variable y repitiéndolos 10 veces en una lista, que es finalmente devuelta por *pwgl-progn*.



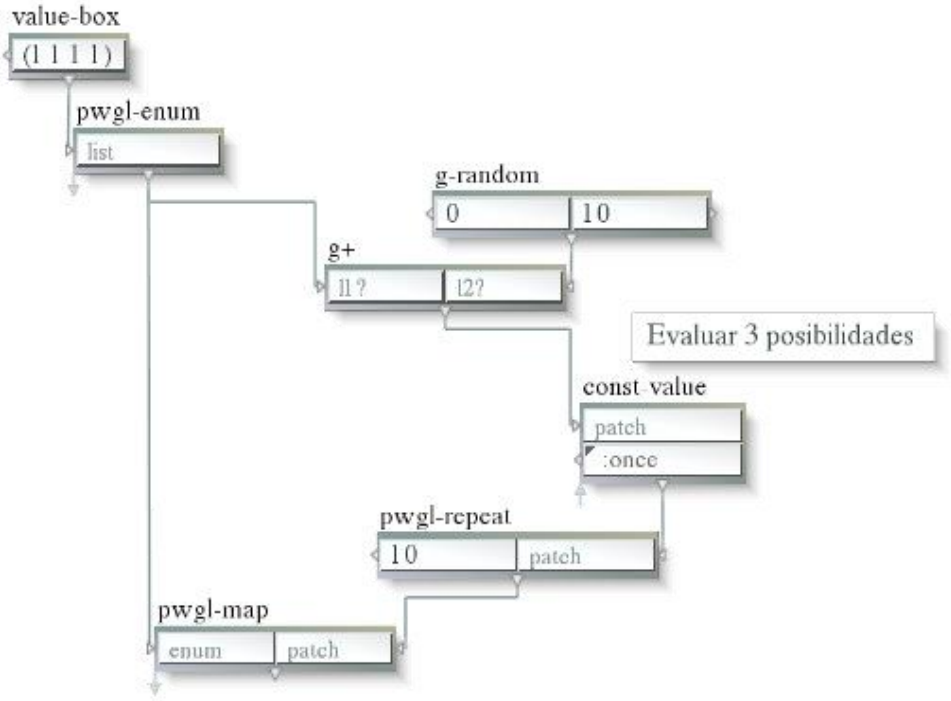
10-Control

En el ejemplo 11, *patch* 1, utilizamos el objeto *permut-random* (opción *Sets/Combinations*) para obtener una permutación al azar de los elementos de una lista. Pero al igual que lo sucedido con *g-random*, este objeto devuelve una nueva permutación cada vez que es evaluado. A fin de evitar este comportamiento, utilizamos nuevamente el objeto *pwgl-value*.



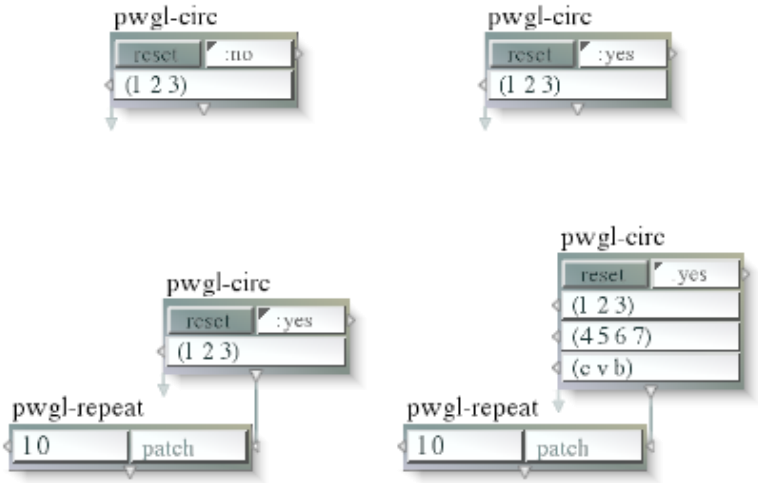
11-Control

En el *patch* siguiente enumeramos los elementos de una lista, generando 10 números al azar por cada elemento de la lista, a los cuales se suma el valor numérico del elemento. Pero en este caso vamos a incorporar el objeto *const-value* con sus tres posibilidades de operación. *Const-value* conserva el valor del *patch* conectado a su entrada, aún cuando su salida esté conectada a varios objetos, o a un objeto *pwgl-repeat*. Las tres opciones son :*once* (el objeto evalúa sólo una vez durante todo el bucle), :*loopinit* (el objeto evalúa cada vez que se inicia el bucle), y :*eachtime* (el objeto evalúa a cada ciclo de repetición del bucle). La ejecución del ejemplo en sus tres formas, y la observación de los resultados, sirve a una mejor comprensión de los alcances de este objeto.



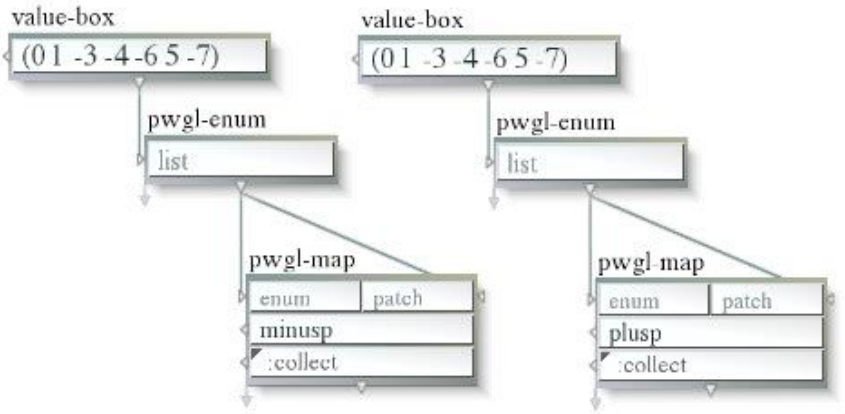
12-Control

Con *pwgl-circ* realizamos permutaciones circulares de una lista.



13-Control

Para finalizar con el análisis de los objetos de control veamos otros usos posibles del par *enum-map*. Si extendemos el objeto *pwgl-map* (+) vemos que contamos con nuevas opciones. En el segundo cuadro que aparece podemos establecer una condición, y tal condición puede ser que el número a incluir en el proceso sea negativo, impar, etc. En el tercer cuadro, observamos las siguientes variantes, :*collect* (incluye los elementos que pasaron el test en la lista resultante), :*append* (une los elementos cuando se trata de listas) , :*sum* (suma el valor numérico de los elementos) :*count* (cuenta la cantidad de elementos enumerados), :*maximize* (devuelve el valor máximo de los elementos que pasaron el test) y :*minimize* (devuelve el valor mínimo). El cuadro restante (*identity*) requiere un manejo avanzado del lenguaje LISP, por lo cual lo dejaremos pendiente por el momento.



14-Control

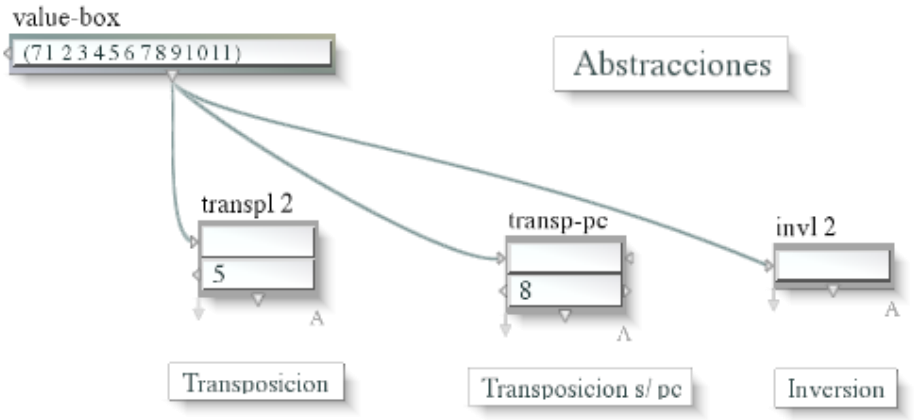
Abstracciones

Podemos crear nuevos objetos en los cuales incluimos *patches* de PWGL, creados por nosotros mismos. A estos objetos los denominamos abstracciones.

Para crear una abstracción hacemos un click derecho sobre el fondo de la ventana, elegimos la opción *Abstraction*, y luego *abstract-box*. Aparece así en pantalla el nuevo objeto, y si hacemos doble click sobre él se abre una nueva ventana donde podemos programar, o bien, pegar un *patch* previamente realizado.

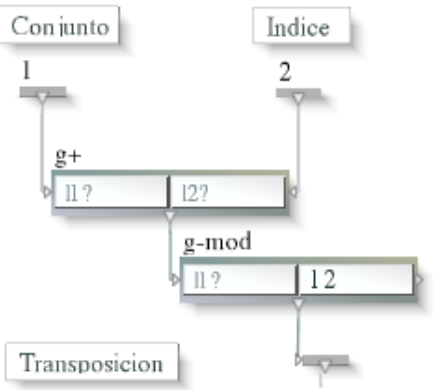
Al hacer click derecho sobre esa ventana, y al elegir nuevamente la opción *Abstraction*, vemos que aparecen nuevos ítems: *abstract-input* y *abstract-output*. Elegimos esas opciones para crear las entradas y salidas de nuestro objeto.

El *patch* siguiente contiene tres abstracciones, cuyas funciones son transportar una lista de grados una cantidad determinada de semitonos, transportar la lista comenzando con un grado en particular, e invertir módulo 12 la lista.



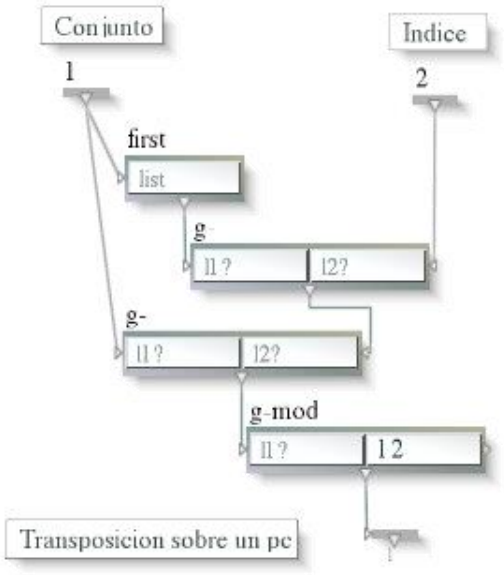
15-Abstracciones

La primera abstracción suma los grados de la lista a un índice de transposición, y luego calcula el módulo 12, pues el resultado de la suma podría ser mayor que 11.



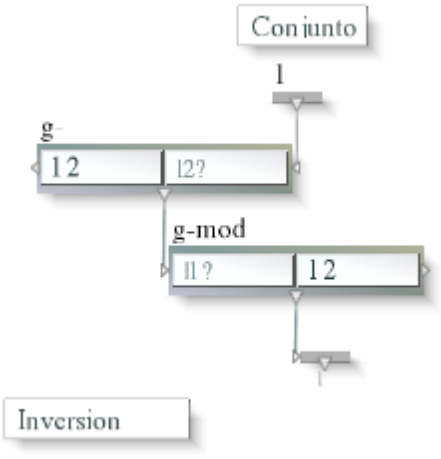
Abstracción *transp12*

En la segunda abstracción utilizamos el objeto *first* (opción *Lisp*) para obtener el primer elemento de la lista de grados cromáticos. A ese número le restamos el grado sobre el cual deseamos transportar la secuencia, obteniendo así el índice de transposición. Posteriormente, restamos a cada elemento de la lista el índice, y aplicamos módulo 12 para limitar el resultado entre 0 y 11.



Abstracción 21tore21s-*pc*

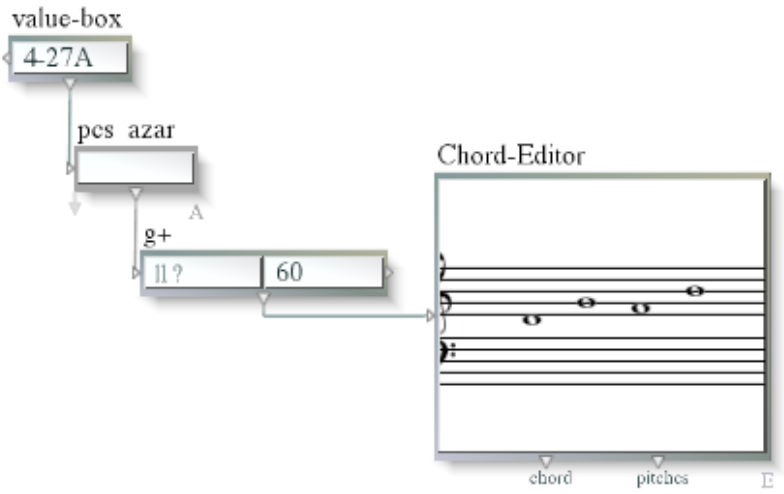
En la tercera y última abstracción realizamos la inversión módulo 12 de la lista de grados restando a 12 el grado a invertir. Finalmente aplicamos módulo 12 para limitar el resultado entre 0 y 11.



Abstracción *inv12*

Para el siguiente programa vamos a crear una abstracción que reciba el nombre de un conjunto de grados cromáticos (*pitch class set*) y nos devuelva una versión permutada, transpuesta y posiblemente invertida, al azar, de su forma prima.

Luego de obtener el conjunto, sumamos 60 a cada grado de la lista resultante a fin de obtener una representación musical en la octava central.



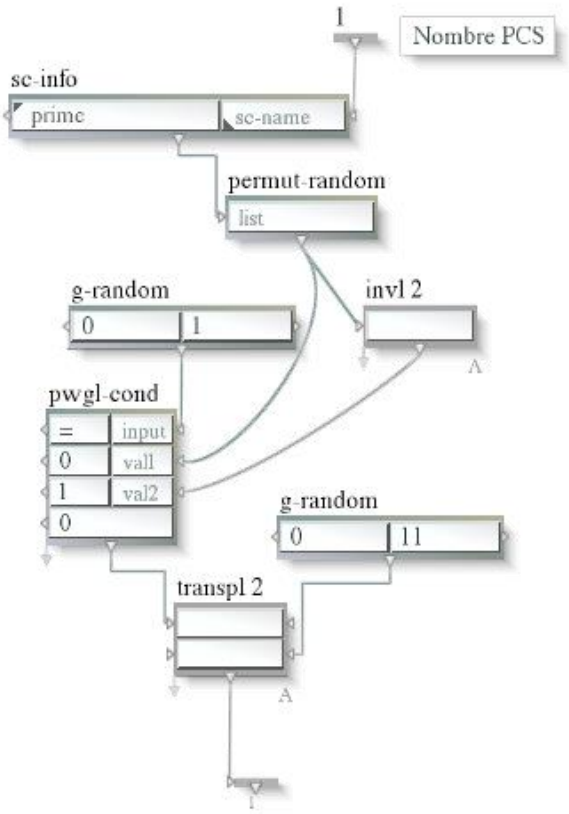
16-Abstracciones

En esta abstracción vamos a incluir las abstracciones *inv12* y *transp12* que realizamos anteriormente.

Partimos del objeto *sc-info* (que se encuentra en la opción *PC-set-theory*), al cual le indicamos el nombre de un PCS y nos da por resultado una lista con su forma prima, y luego permutamos la lista valiéndonos de *permut-random*. A fin de decidir si lo vamos a invertir o no, utilizamos el objeto g-*random* para que genere un número al azar entre 0 y 1, y el objeto *pwgl-cond* para establecer el flujo de datos de acuerdo a la opción resultante: si sale 0 no se invierte, y si sale 1, la lista pasa por la abstracción *inv12*.

Es importante tener en cuenta que del objeto *permut-random* salen dos cables. Por lo cual, la permutación no va a ser la misma para la versión original que para la invertida. De todas formas, en nuestro caso esto no representa un problema, dado que igualmente se cumple el objetivo de obtener una permutación aleatoria.

Finalmente, generamos una transposición al azar con la abstracción *transp12*, que obtiene el índice de transposición de manera aleatoria.

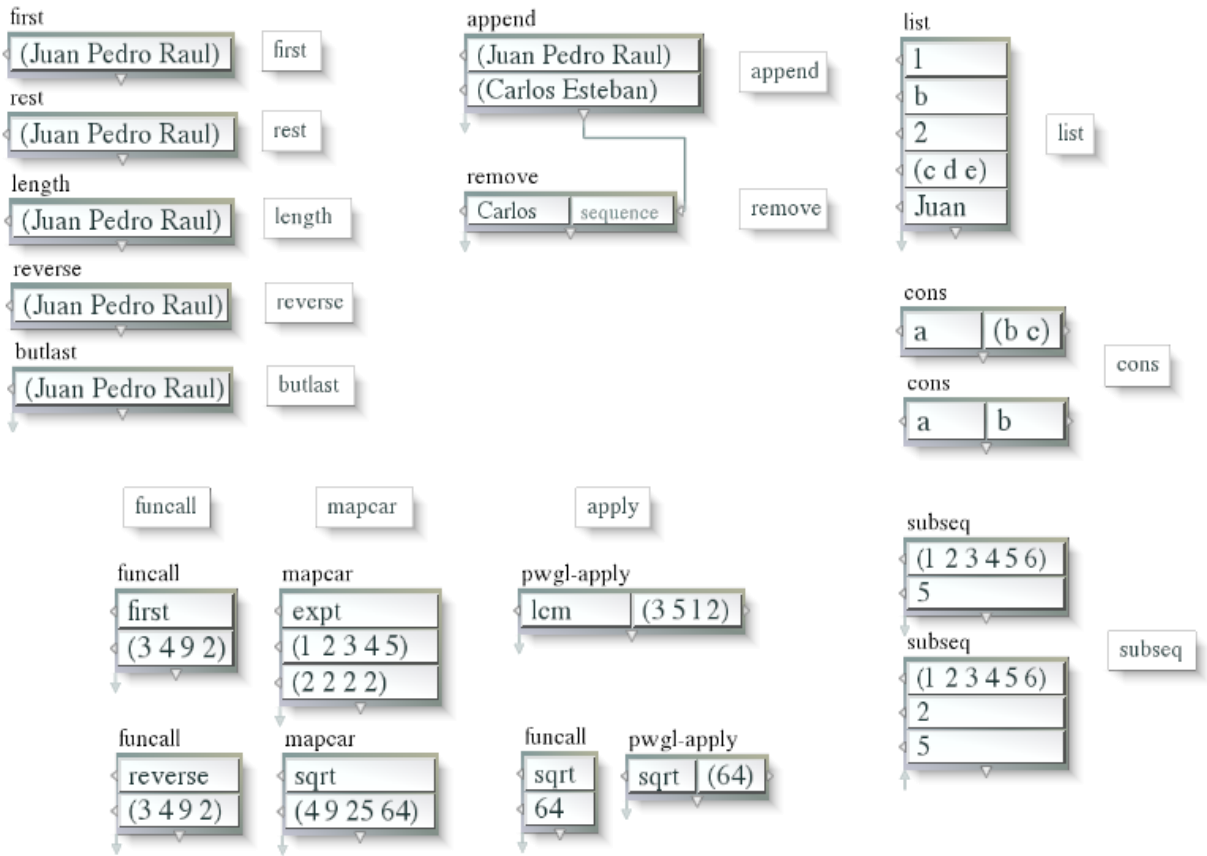


Abstracción *pcs-azar*

Funciones del menú LISP

En este menú encontraremos las funciones LISP más utilizadas. En el ejemplo 17 observamos diversas funciones para el tratamiento de listas y para el llamado a otras funciones.

first: devuelve el primer elemento de una lista.
rest: devuelve la lista, menos el primer elemento.
length: devuelve la longitud de la lista.
reverse: retrograda una lista.
butlast: devuelve todos los elementos de una lista, menos el último.
append: concatena dos o más listas.
remove: remueve un elemento especificado de una lista.
list: crea una lista a partir de números, símbolos o listas.
cons: función para construir listas (crea y devuelve una nueva celda CONS).
subseq: devuelve parte de una lista . El resto que sigue a un elemento especificado (un argumento), o el resto que sigue a un elemento especificado hasta alcanzar otro elemento especificado (dos argumentos).
funcall: función que devuelve el resultado de aplicar una función (primer argumento) con el resto de los argumentos.
apply: devuelve el resultado de aplicar la función que se determina a los argumentos que la acompañan. A diferencia de *funcall*, los argumentos se colocan en una lista.
mapcar: aplica una función a los elementos de una lista o conjunto de listas.

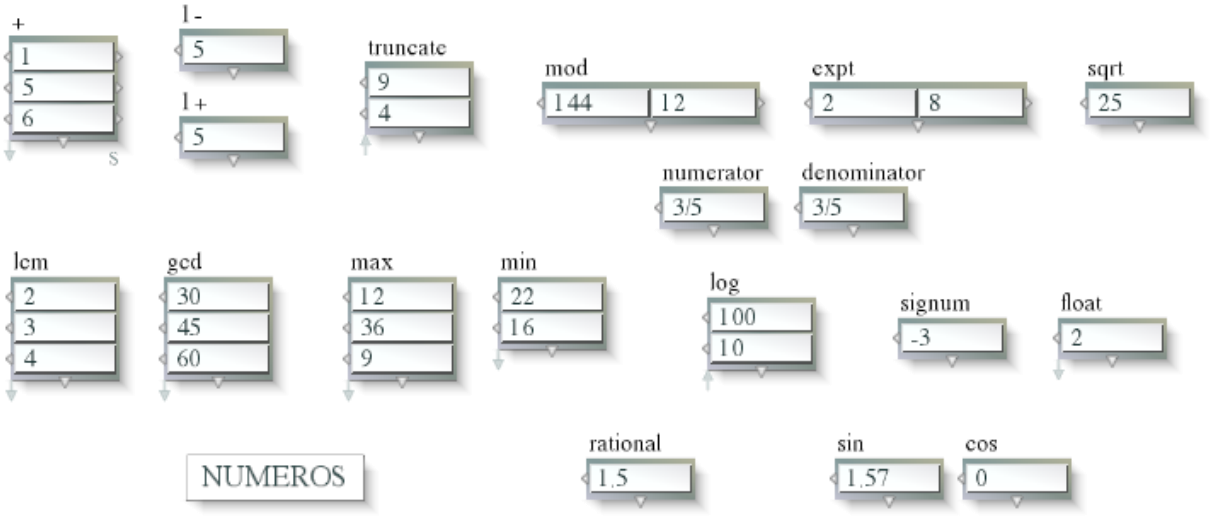


17-Lisp

Funciones primitivas de LISP

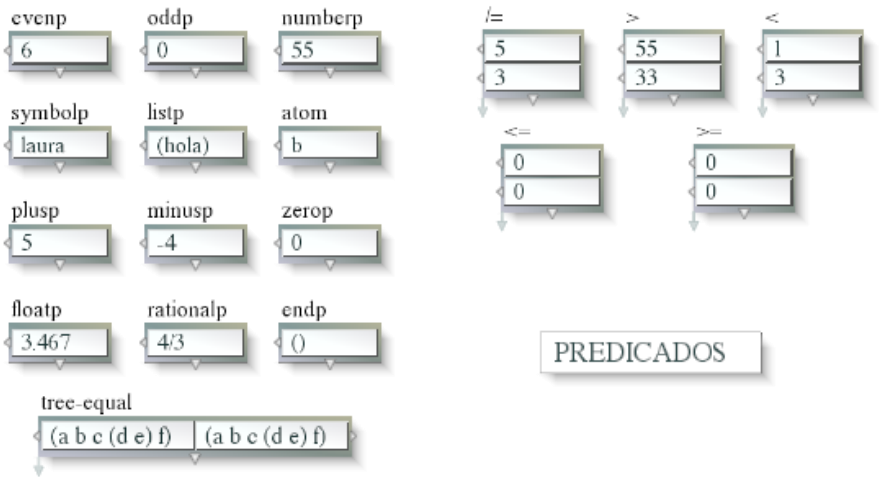
Es posible incorporar las funciones *built-in* de LISP a nuestro *patch* en PWGL. Para ello, hacemos doble *click* en el fondo de la ventana, y escribimos el nombre de la función en el cuadro de diálogo que aparece.

En el ejemplo siguiente observamos las funciones + (suma), 1- (decremento), +1 (incremento), *truncate* (truncamiento), *mod* (módulo), *expt* (potenciación), *sqrt* (raíz cuadrada), *numerator* (numerador de un número fraccionario), *denominator* (denominador de un número fraccionario), *lcm* (mínimo común múltiple), *gcd* (máximo común divisor), *max* (valor máximo de una lista), *min* (valor mínimo de una lista), *log* (logaritmos en base *n*), *signum* (signo de un número), *float* (conversión a decimal), *rational* (conversión a fraccionario), *sin* (seno) y *cos* (coseno).



18-Primitivas

Dentro de las funciones primitivas se encuentran los predicados, que devuelven *T* o *nil* en respuesta a la veracidad o falsedad de un enunciado. Entre ellos, *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. Un átomo es un número, carácter o símbolo, y no una lista), *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). Asimismo, los comparadores /=, >, <, <= y >=.



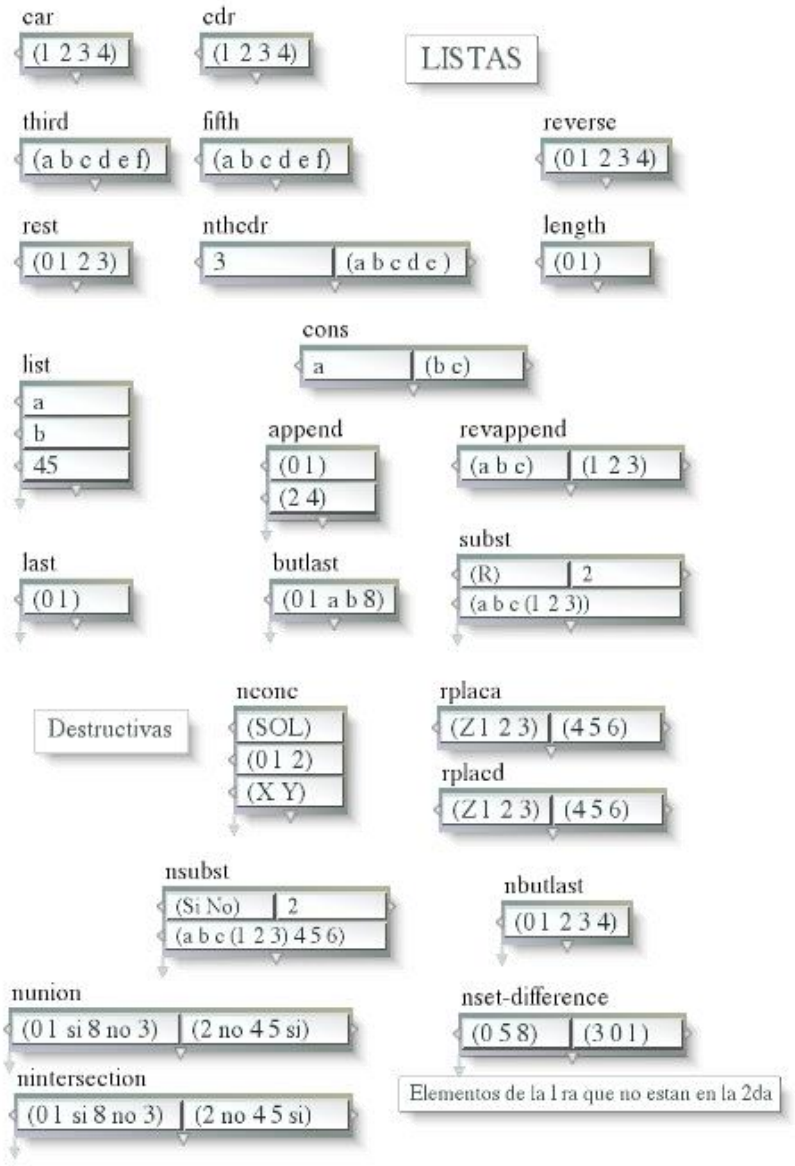
19-Primitivas

También existen funciones primitivas para las operaciones con listas, entre ellas:

car, *cdr* y *cons*, que aplican las funciones ya mencionadas al tratar los conceptos básicos de LISP.
third, *fifth*, que extraen el tercer y quinto elemento de una lista, respectivamente.
reverse, que retrograda una lista.
rest, que devuelve la lista menos el primer elemento.
nthcdr, que devuelve un elemento, especificando su posición dentro de la lista.
list, que crea una lista a partir de elementos o sublistas dadas.
append, que une dos listas, y *revappend*, que une el retrógrado de la primera lista a la segunda.
last, que devuelve el último elemento, y *butlast*, que devuelve todos menos el último.
subst, que sustituye una parte de una lista por otra.

Y las funciones destructivas (que modifican la lista argumento al devolver el resultado).

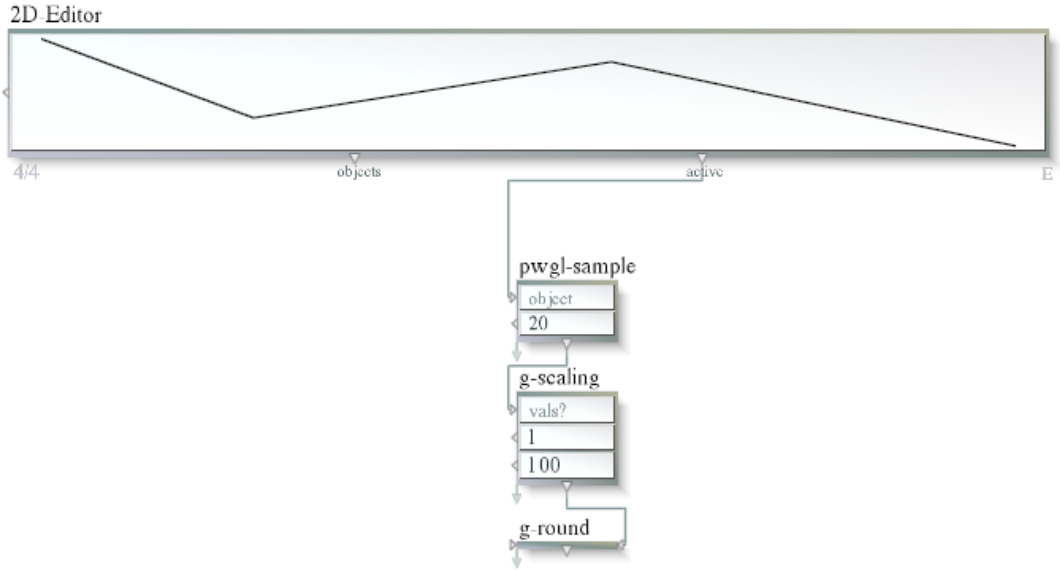
nconc, que concatena elementos o listas.
rplaca, que reemplaza el CAR de una lista por otro elemento o sublista.
rplacd, que reemplaza el CDR de una lista por otro elemento o sublista.
nsubst, que sustituye elementos de una lista por sublistas.
nbutlast, similar a *butlast*.
nunion y *nintersection*, unión e intersección de dos listas, respectivamente.
nset-difference, que devuelve los elementos de la primera lista que no están en la segunda.



20-Primitivas

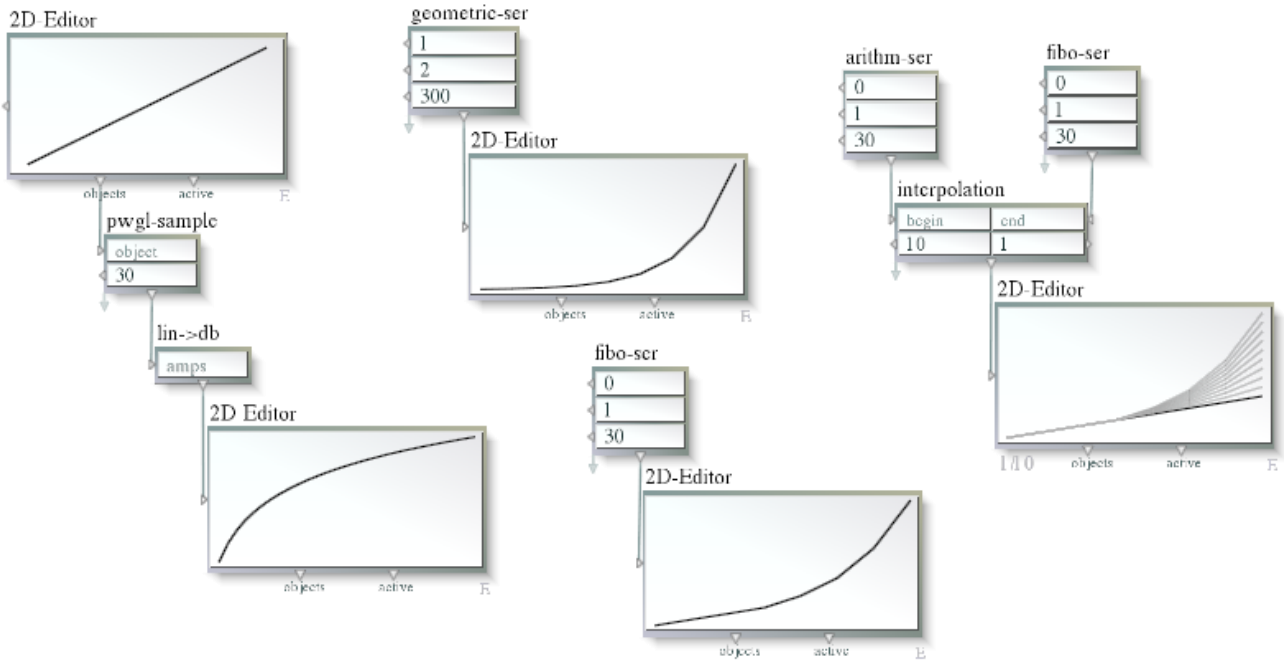
Editor 2D y series numéricas

A través del Editor 2D (opción *Editors*) es posible definir una función por medios gráficos, utilizando *breakpoints*. El objeto *pwgl-sample* sirve para obtener una cantidad predeterminada de muestras de esa función, que pueden ser escaladas mediante el objeto *g-scaling*.



21-2D Editor

En la opción *Num series* hallamos diversos objetos destinados a generar series numéricas (aritméticas, geométricas, de Fibonacci, de números primos), y objetos de escalamiento e interpolación. Veamos a continuación algunas aplicaciones.



22-Series numéricas

Código LISP

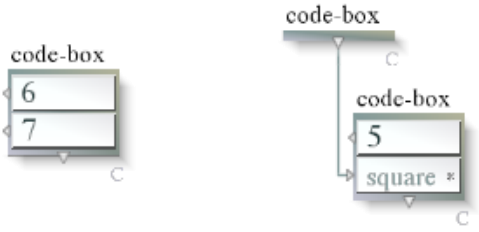
En la opción de menú *Data* encontramos el objeto c*ode-box*. Haciendo doble *click* sobre él se abre un ventana, en la cual podemos escribir directamente código LISP, y crear de esa manera un objeto que responda a nuestras necesidades.

En la figura 23 se observan dos ejemplos. En el de la izquierda, incluimos el texto `(+ a b)`, con lo cual creamos un objeto capaz de sumar dos números. Al colocar dos variables (*a* y *b*), el objeto se genera automáticamente con dos entradas. Dado que los argumentos son 6 y 7, la evaluación da por resultado 13.

En el ejemplo de la derecha, hay dos objetos *code-box*. El de arriba tiene el siguiente texto:

```
(defun square (n) (* n n))
(square n)
```

que sirve para crear la función *square*. El de abajo, contiene el código `(square n)`, que es el llamado a la función, con un argumento igual a 5. El resultado es 25.

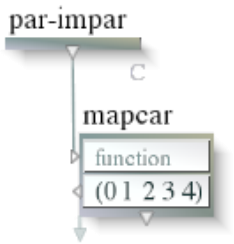


23-Código

El objeto *code-box* de la figura 24 contiene el código

```
(defun par-impar (n) (if (oddp n) 'impar 'par))
```

Se define así a la función *par-impar*, que devuelve el símbolo *impar* o *par* de acuerdo a la condición del número *n*. El objeto *mapcar* al que está conectado permite aplicar la función *par-impar* a todos los elementos de una lista. La evaluación de *mapcar* da como resultado la lista `(PAR IMPAR PAR IMPAR PAR)`.



24-Código

APLICACIONES

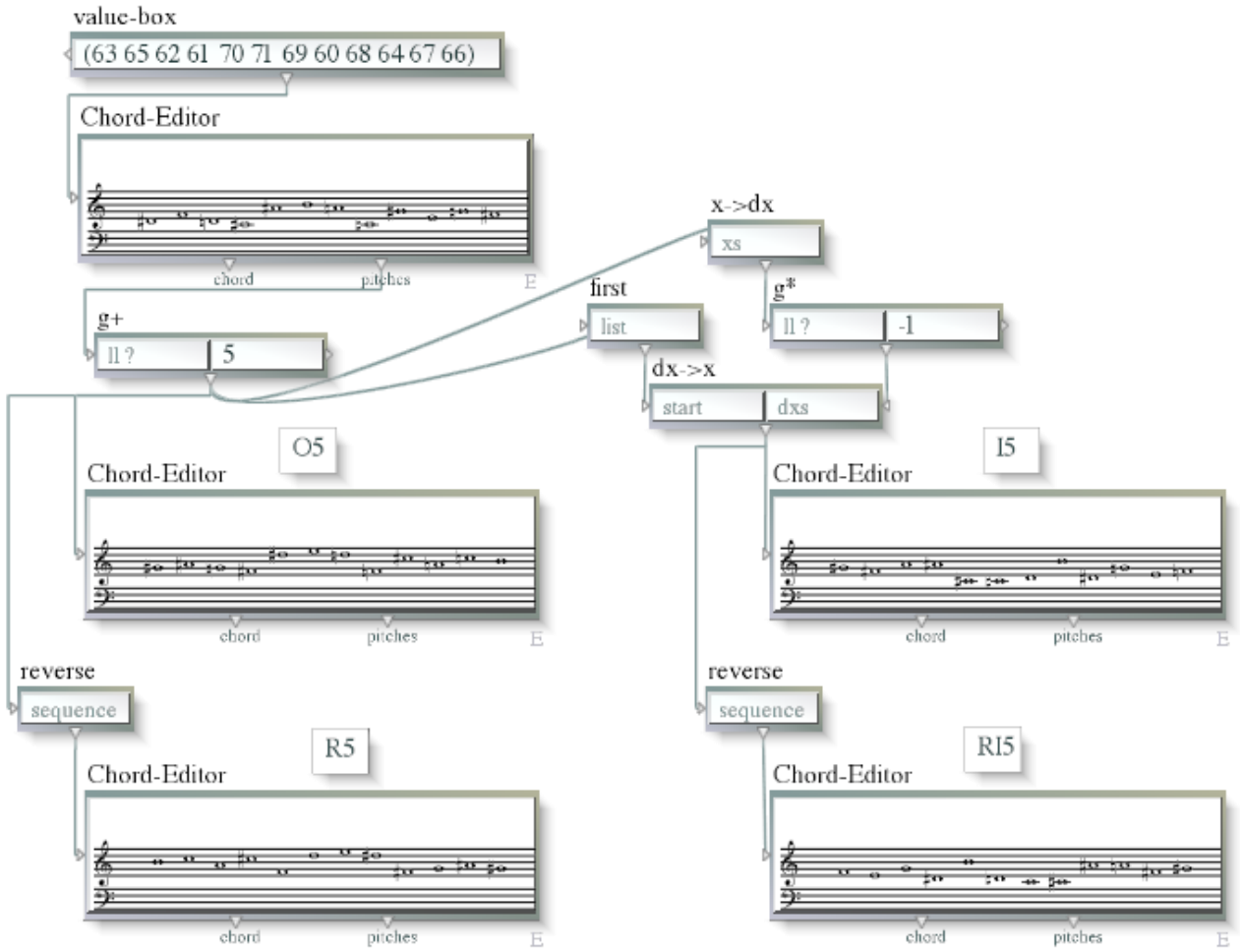
Modos seriales

Partiendo de una serie dodecafónica, vamos a calcular 4 modos: O5, I5, R5 y RI5.

La transposición se realiza con el objeto g+.

Para la inversión recurrimos al objeto *x->dx* (en la opción *Num series*), que calcula los intervalos entre las notas MIDI sucesivas, en cantidad de semitonos. Luego, multiplicamos los intervalos por -1 para invertirlos, y partiendo del primer elemento de la serie (obtenido con *first*), generamos las notas siguientes de la inversión con el objeto *dx->x*. Este último objeto tiene como argumentos una nota de partida y una lista de intervalos, y devuelve una lista de notas creadas a partir de esos intervalos.

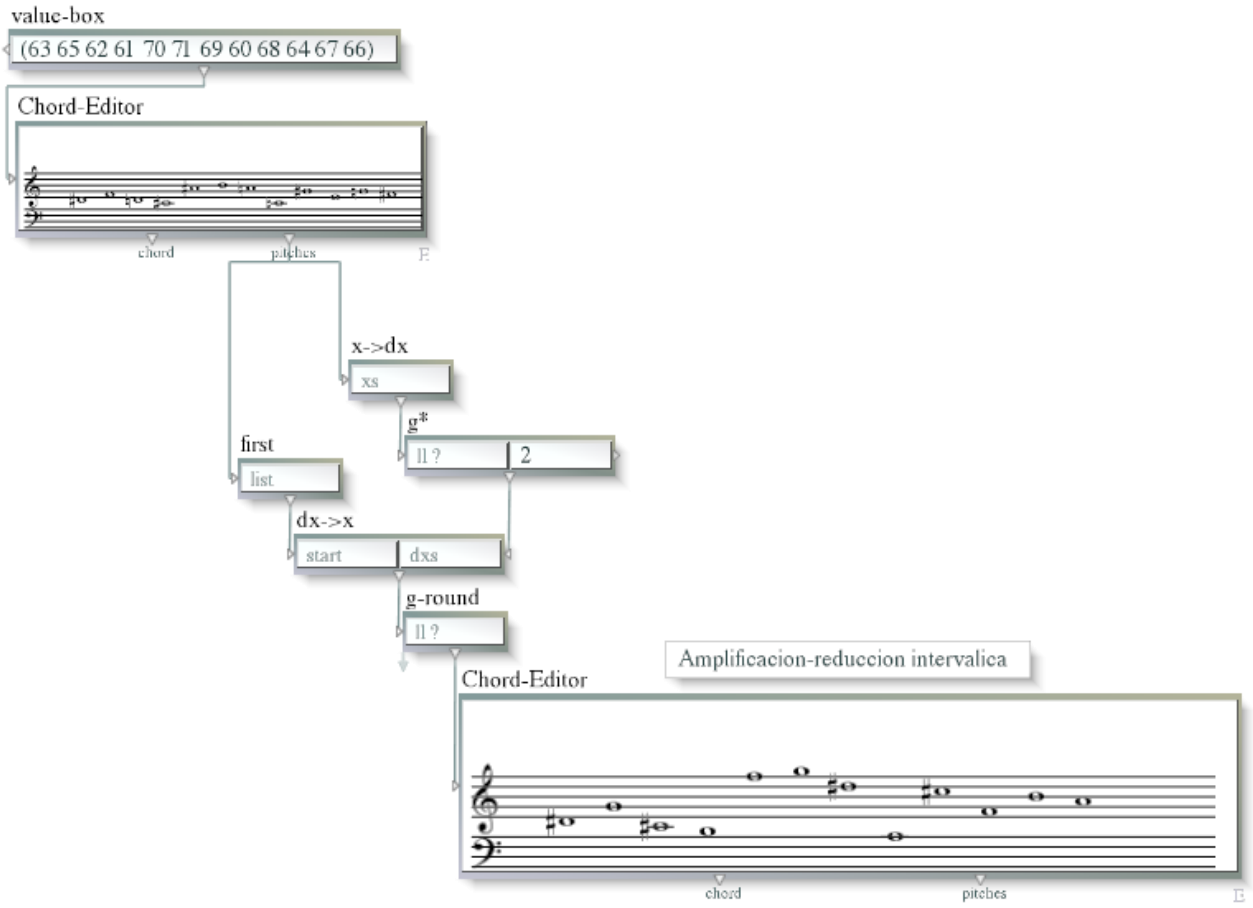
Calculadas O e I, R y RI se obtienen mediante *reverse*, objeto que retrograda una lista dada.



25-Modos seriales

Amplificación interválica

Tomando la misma serie anterior, vamos ahora a modificar el contenido interválico mediante la multiplicación de la cantidad de semitonos de cada intervalo por un factor (2, en el ejemplo). Finalmente, redondeamos el resultado de los productos, para llevarlos a la nota MIDI más próxima.

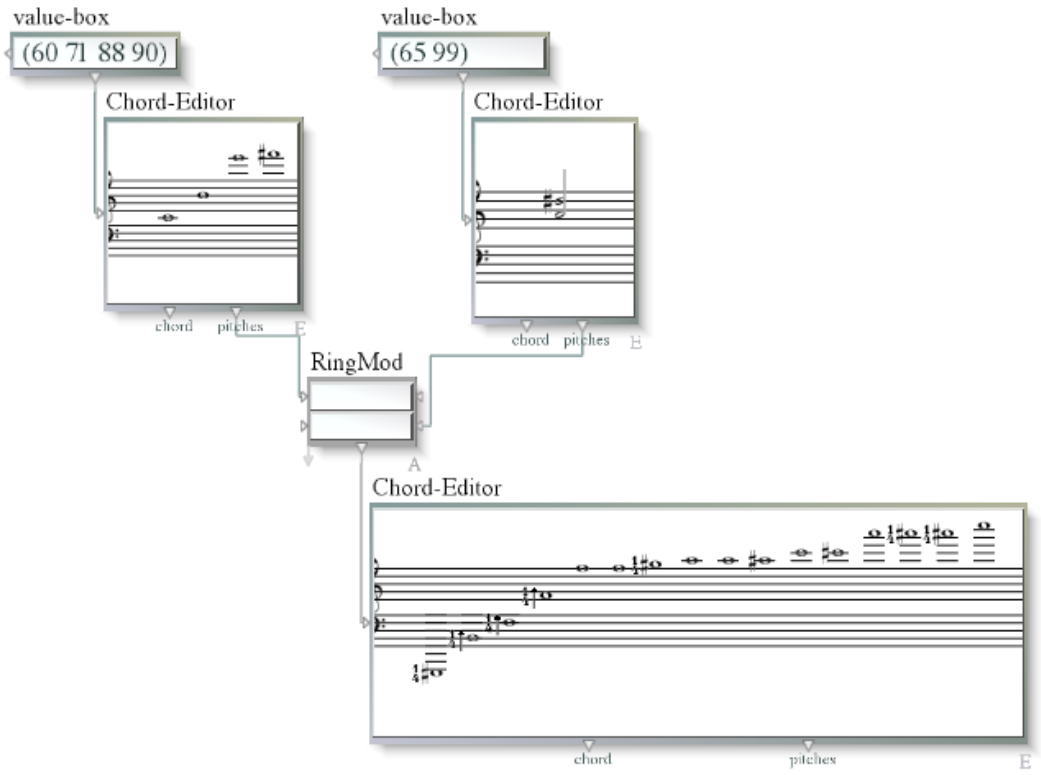


26-Amplificación interválica

Sonidos de combinación

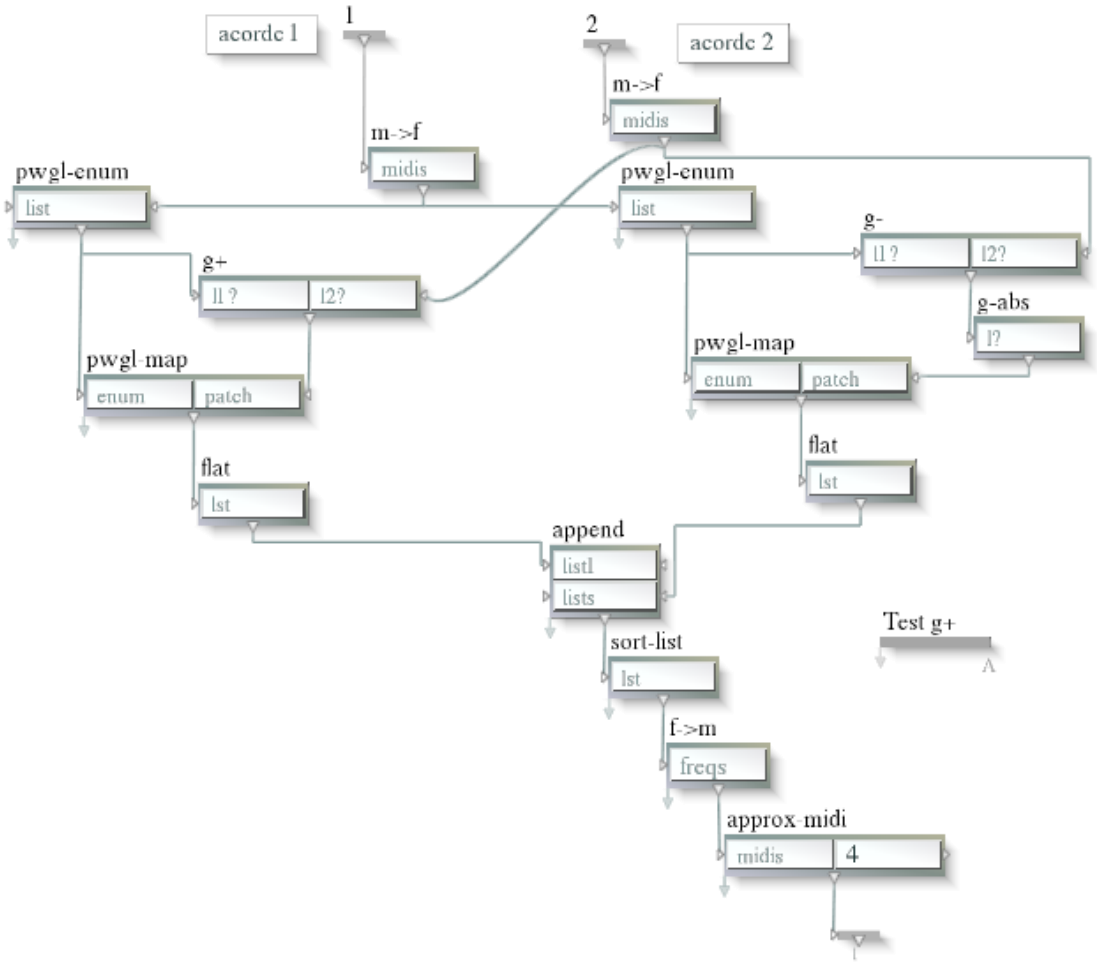
Los sonidos de combinación se dividen en adicionales y diferenciales, y se obtienen realizando la suma y la resta de cada componente de frecuencia de un espectro con cada una de las componentes de otro espectro. Se trata de un fenómeno de distorsión propio de la audición, pero a la vez, de un tipo de procesamiento habitual en la música electrónica. En tanto proceso, se denomina convolución espectral, pero también es comúnmente llamado modulación en anillo, por el tipo de circuito con el que se realizaba antiguamente en los medios analógicos. Posteriormente se empleó como procedimiento de generación de alturas microtonales en la música instrumental.

En la figura 27 observamos dos acordes, que sirven de base a la modulación. A partir de ellos, obtenemos la suma y la resta de la frecuencia de cada nota del primer acorde con la frecuencia de cada nota del segundo acorde. Dado que los resultados pueden no coincidir con las frecuencias de las notas del sistema temperado, realizaremos una aproximación al cuarto de tono más próximo, obteniendo así un nuevo acorde microtonal, con una resolución por cuartos de tono.



27-Ring mod

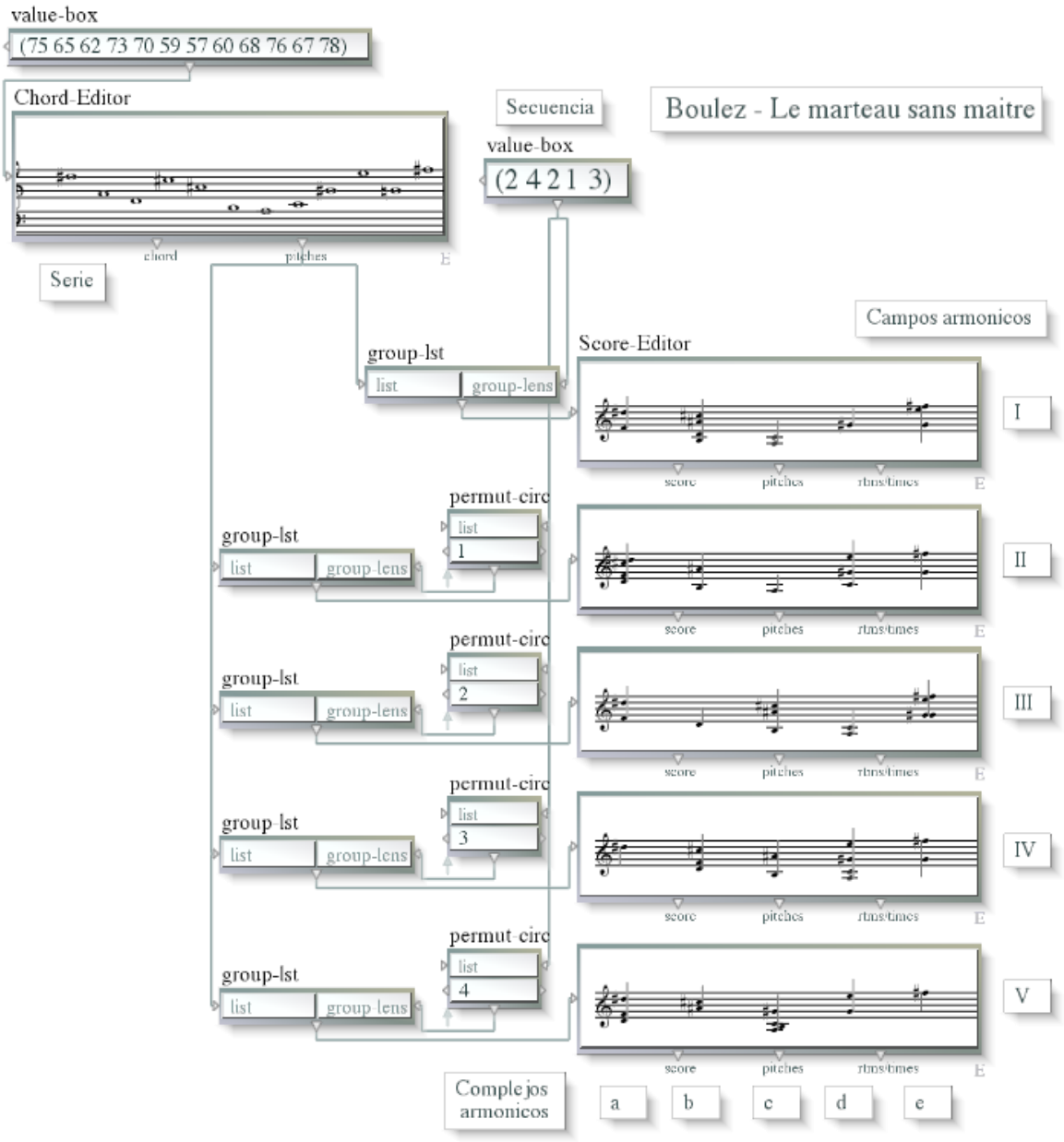
A fin de convertir las notas MIDI en frecuencias empleamos el objeto *m->f* (opción *Conversion*). Con *pwgl-enum* tomamos las frecuencias de cada nota del primer acorde para sumarlas y restarlas a las del segundo, y en el caso de la resta, utilizamos *g-abs* para que el resultado sea siempre un número positivo. Luego aplicamos *flat* para quitar los paréntesis internos que agregaron los objetos empleados, unimos los resultados de las sumas y las restas con *append*, ordenamos la lista resultante de menor a mayor con *sort-list*, convertimos de frecuencias a notas MIDI, y redondeamos al cuarto de tono más próximo con *approx-midi*.



Abstracción *RingMod*

Multiplicación de acordes

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 la secuencia (4 2 1 3 2, 2 1 3 2 4, etc.) 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*. El *patch* siguiente calcula esos campos armónicos, teniendo como datos la serie dodecafónica y la secuencia numérica.



28-Multiplicación

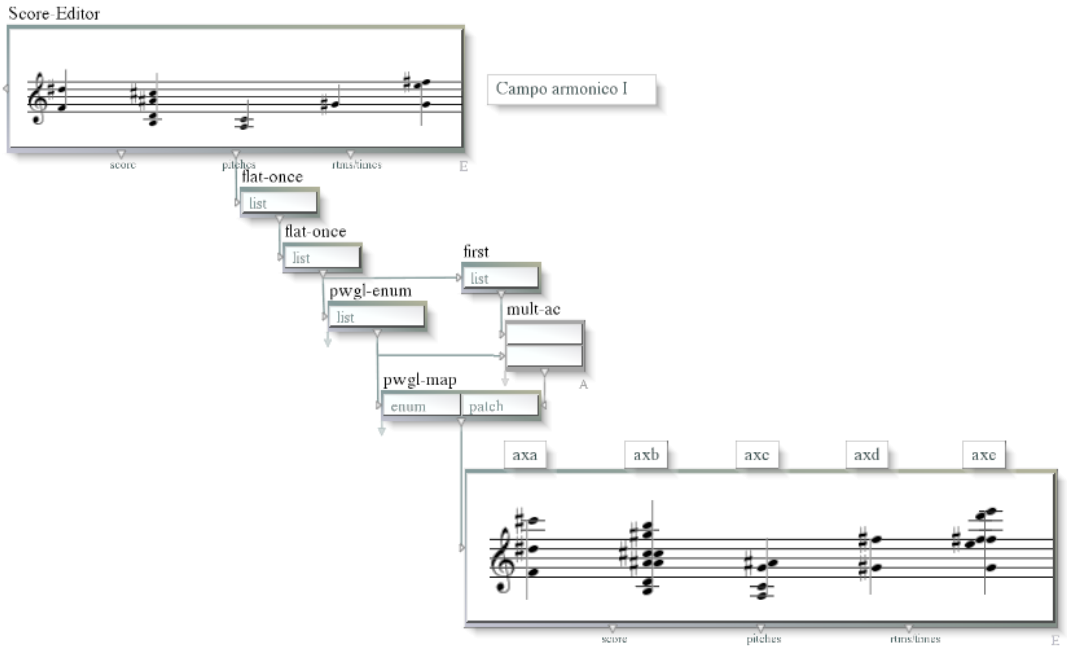
Para este ejemplo, utilizamos el objeto de notación musical *Score Editor*. Para representar en él una sucesión de acordes basta con ingresar una lista, con sublistas en su interior que contengan las notas MIDI de cada acorde. Para el primer campo armónico, la lista es:

```
((75 65) (62 73 70 59) (57 60) (68) (76 67 78))
```

Las agrupaciones de las notas de la serie se realizan con el objeto *group-list*, y las permutaciones circulares con *permut-circ*.

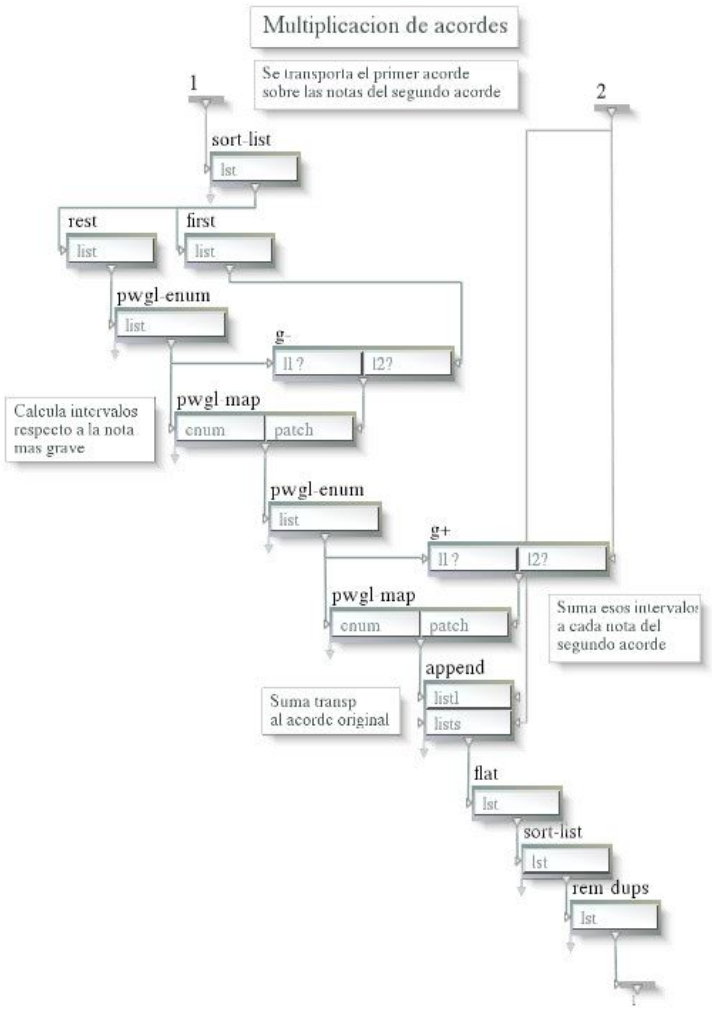
Obtenidos los campos armónicos, el compositor se vale de un procedimiento que denomina multiplicación de acordes. Para multiplicar dos acordes, transportamos al primer acorde sobre las notas del segundo acorde.

El *patch* siguiente calcula la multiplicación del acorde *a* con si mismo, y con *b*, *c*, *d* y *e*. Los objetos *flat-once* son utilizados aquí para eliminar dos niveles de paréntesis, lo cual puede analizarse evaluando la salida del *score-editor* y de los objetos mencionados. Con el objeto *first* elegimos al primer acorde (*a*) y con *pwgl-enum* seleccionamos de a uno los cinco acordes del editor, para enviarlos a la abstracción *mult-ac*, encargada de realizar la multiplicación.



29-Multiplicación

La abstracción *mult-ac* posee dos entradas, donde ingresan los dos acordes a multiplicar. En la rama de la izquierda (figura *Abstracción mult-ac*) se ordenan de menor a mayor las notas del primer acorde con *sort-list*, se separa la primera nota del resto con *first* y *rest*, y se calculan los intervalos respecto a la nota más grave. Posteriormente, se suman los intervalos a cada una de las notas del segundo acorde, se agregan las notas del primer acorde a la lista con *append*, se eliminan los paréntesis internos, se ordena la lista de menor a mayor nuevamente, y se eliminan las notas MIDI duplicadas, si las hay, con el objeto *rem-dups*.

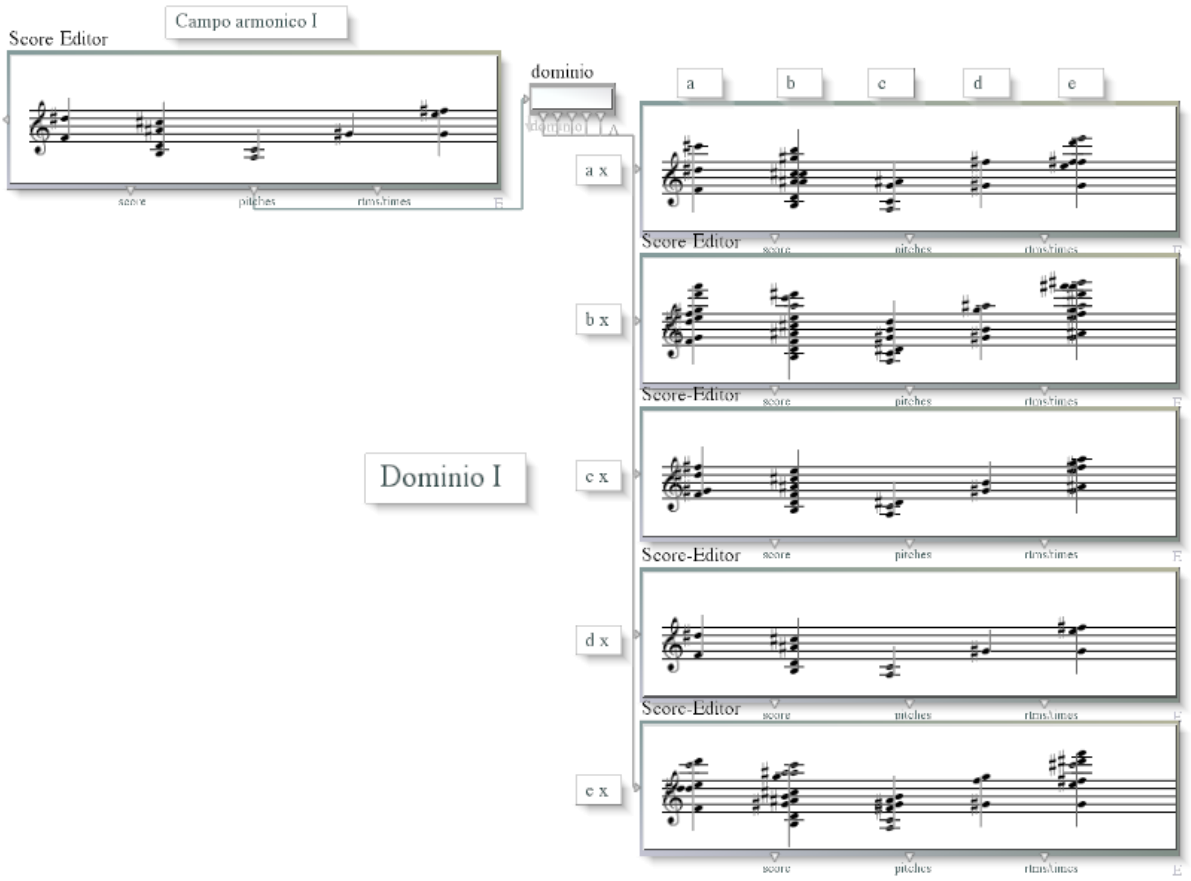


Abstracción *mult-ac*

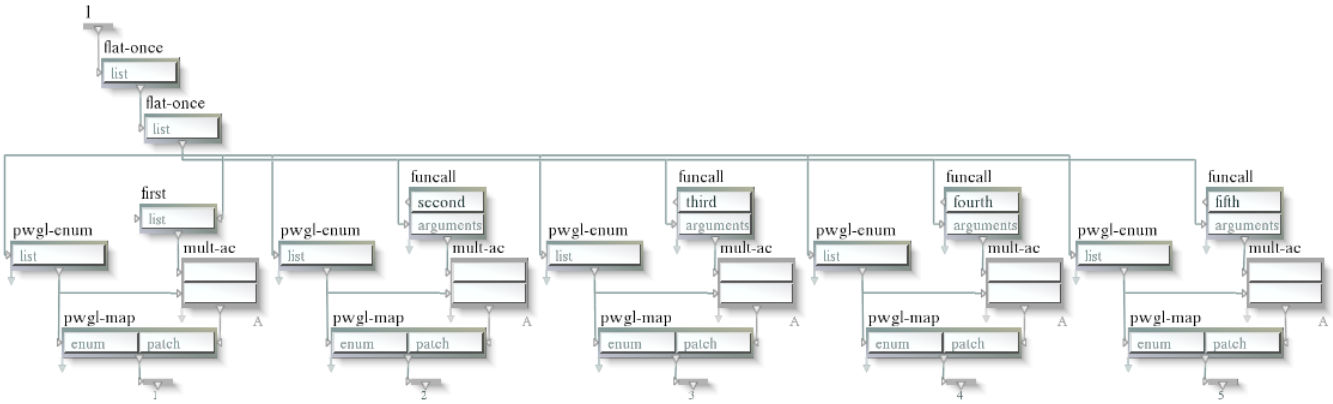
Boulez denomina *dominio* al conjunto de todas la multiplicaciones que se obtienen sobre cada una de las permutaciones circulares de la secuencia numérica. Cada dominio consta de 25 acordes, y al haber cinco permutaciones, el total de acordes resultantes es 125. En la figura 30 se observan todos los acordes que corresponden al primer dominio.

La abstracción *dominio* es una ampliación de lo visto. Simplemente realiza la multiplicación del primer acorde contra todos, del segundo contra todos, hasta llegar al quinto. El primer acorde, a multiplicar por todos, es elegido con *first*, mientras que el resto es seleccionado con *funcall*, que llama a las funciones primitivas *second*, *third*, *fourth* y *fifth*.

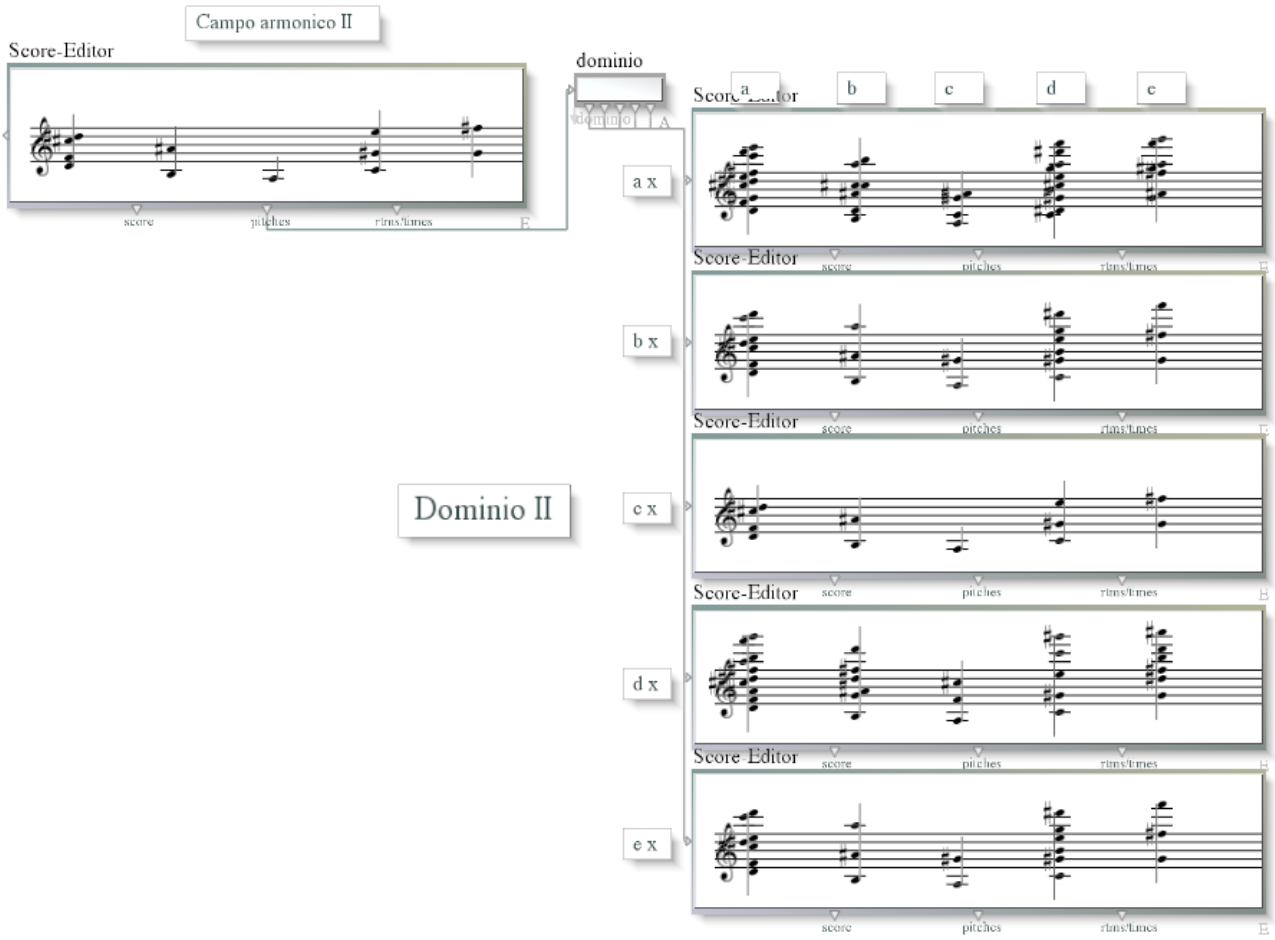
Por último, en la figura 31 observamos los acordes que corresponden al segundo dominio, que resulta de aplicar la primera permutación circular a la secuencia numérica 2 4 2 1 3.



30-Multiplicación



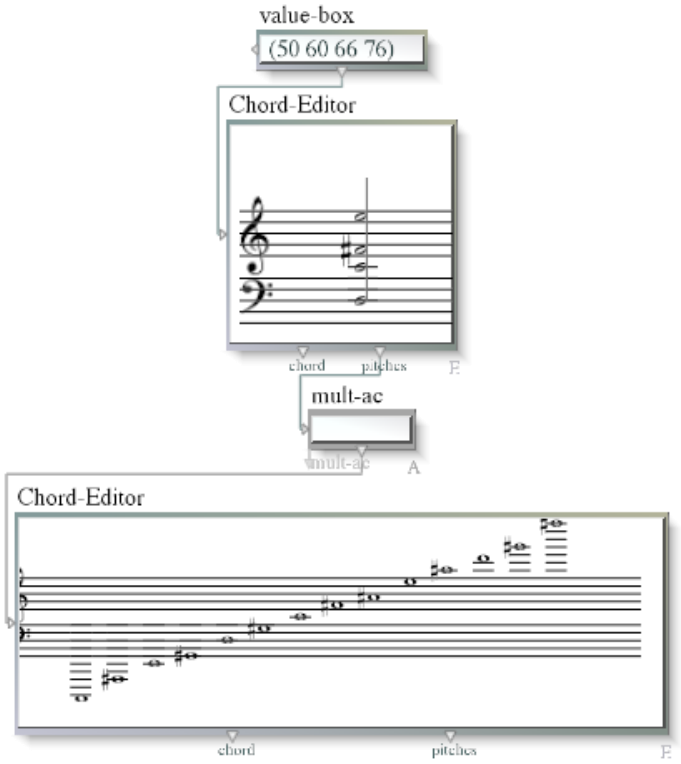
Abstracción *dominio*



31-Multiplicación

Multiplicación ascendente y descendente

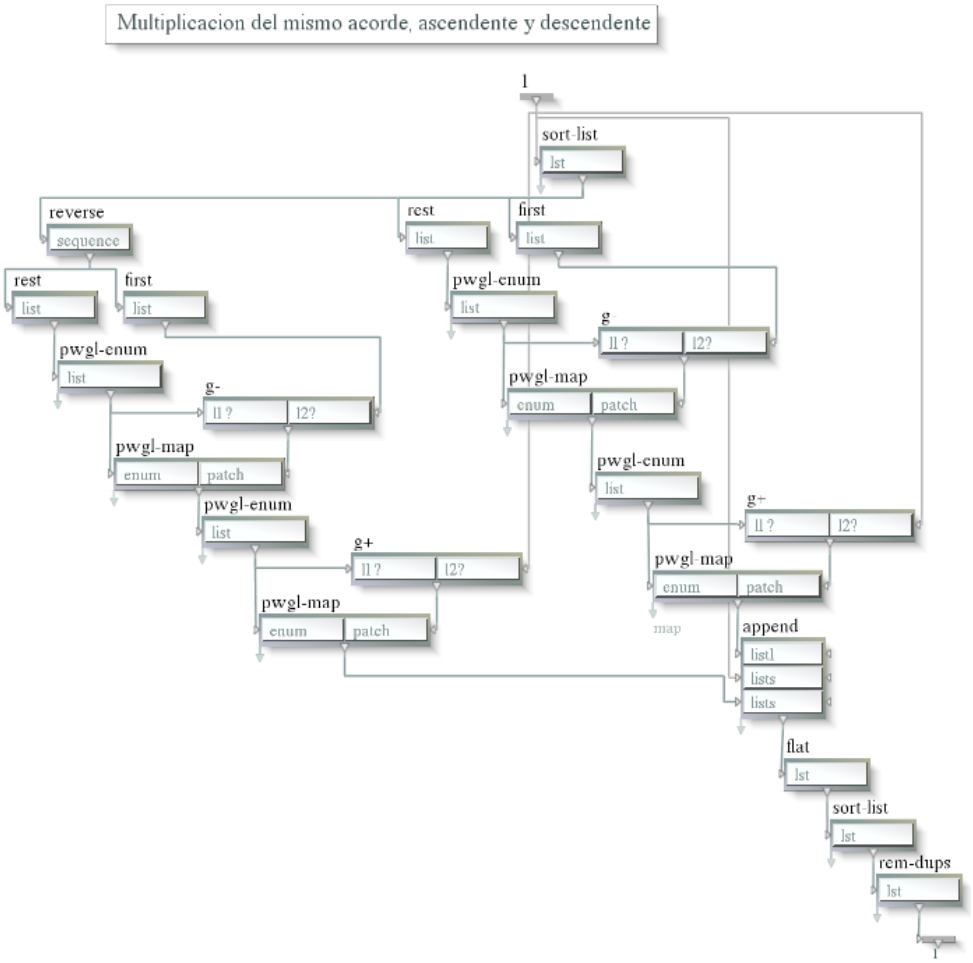
Una variante interesante del procedimiento anterior consiste en multiplicar un acorde por sí mismo, pero de manera ascendente y descendente. Obtenemos una sucesión de alturas en registración fija que puede resultar muy útil en la composición, particularmente si encadenamos dos o más que modulen en nivel de consonancia.



32-Multiplicación ascendente y descendente

En este caso, la abstracción *mult-ac* difiere ligeramente de la anterior dado, que debemos agregar las transposiciones descendentes. Dejo al lector el análisis de la programación, representada en la figura *abstracción mult-ac*.

Del grado de apertura del acorde de partida depende el ámbito de la secuencia resultante, y el grado de coherencia interválico es alto, debido a que un acorde de cuatro sonidos, como el del ejemplo, se encuentra dispuesto siete veces dentro de la secuencia final.



Abstracción *mult-ac*

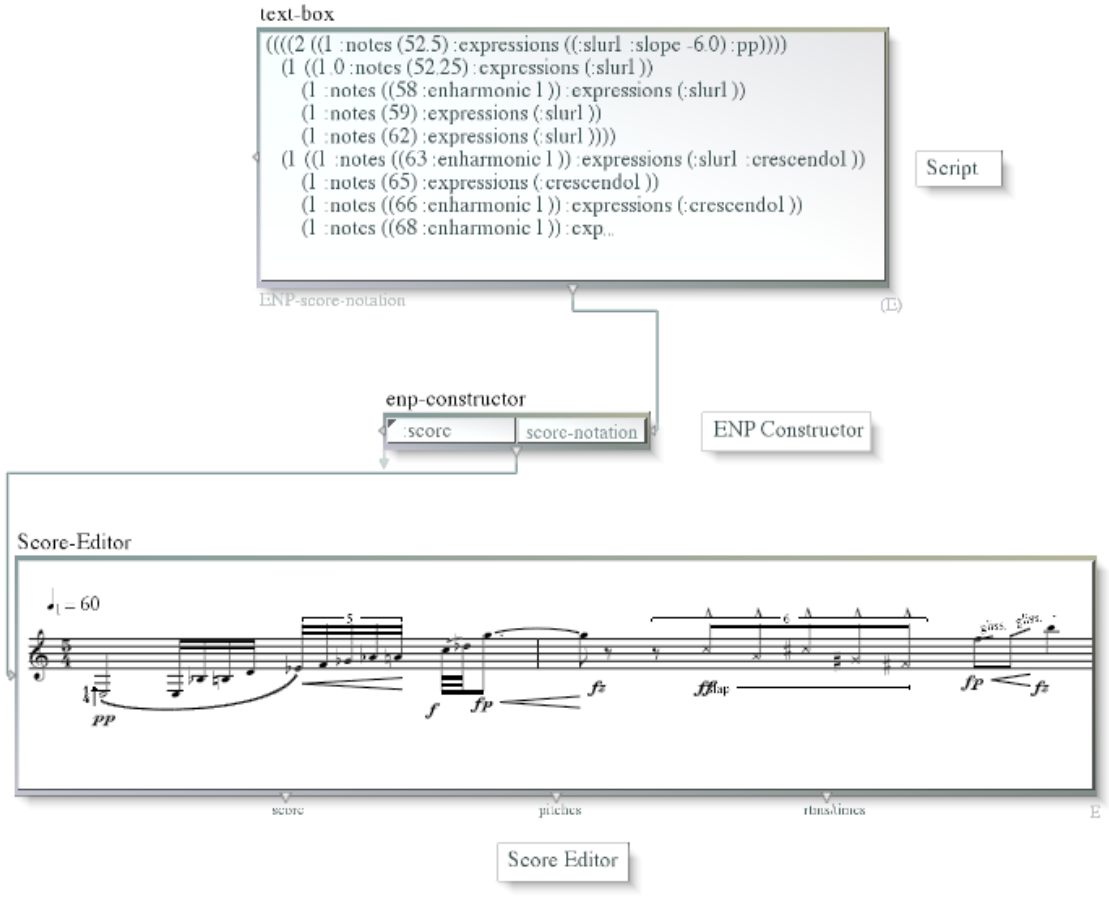
Notación musical

PWGL incluye un editor de partituras basado en formato de texto, denominado *ENP-score-notation*. En la opción de menú *Editors* encontramos algunos objetos que pertenecen al paquete *ENP* (*Expressive Notation Package*). La edición de la partitura se realiza en un cuadro de texto, a través de un *script*. La estructura de una partitura en *ENP* consta de varias partes, y la descripción de estas partes está ligada al modo de ubicar los paréntesis en el texto:

score –> part –> voice –> measure –> beat –> chord -> note

Es posible realizar una partitura completa (*score*), o bien una parte (*part*), una voz (*voice*), compás, tiempo de compás, acorde o incluso una sola nota. En cada caso es necesario emplear un constructor ligado al texto, y un objeto *Score-Editor* para visualizar los resultados.

A fin de simplificar los ejemplos que estamos realizando, vamos a emplear objetos propios de librerías de notación musical simplificadas, como *KSQuant* (que integra la distribución de PWGL) o *studio-flat* (que citamos en la Introducción de este escrito). Para conocer más acerca de *ENP* puede consultarse Kuuskankare y Laurson (2004)[6].



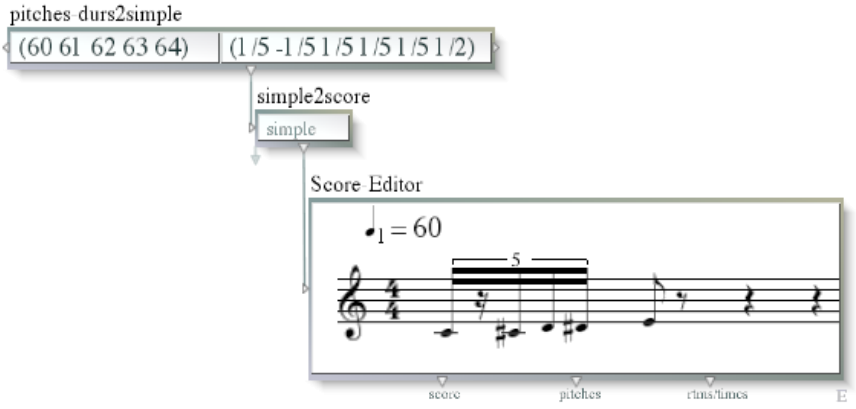
33-Score Editor

[6] Kuuskankare, M. y Laurson, M. "Recent developments in ENP score notation". En *Sound and Music Computing 04*, Octubre de 2004. http://smc04.ircam.fr

Los archivos de las librerías se encuentran en */PWGL/pwgl-libraries*. Si deseamos instalar una librería adicional copiamos su carpeta en ese directorio, vamos al menú *File*, y elegimos la opción *Autoload Libraries*.

En el próximo ejemplo vamos a utilizar la librería *KSQuant*. Se accede a los objetos haciendo *click* derecho sobre el fondo de la ventana de edición, y eligiendo la opción *KSQuant*. Si esta opción no apareciera, debemos ir a *File* y elegir *Autoload Libraries*.

El objeto *pitches-durs2simple* combina notas MIDI y duraciones. La duración de cada nota se especifica con un número fraccionario, y los silencios anteponiendo un signo menos al número. Se trata de un objeto extensible, y entre las opciones disponibles permite indicar el compás, el *tempo*, y la escala (1/4 significa que la negra vale 1, la corchea ½, como en el ejemplo). Si colocamos una única duración (una lista de duraciones con un solo elemento), esa duración se repite para todas las notas MIDI. El objeto *simple2score* sirve finalmente para la conversión del formato *KSQuant* al formato *Score-Editor*.

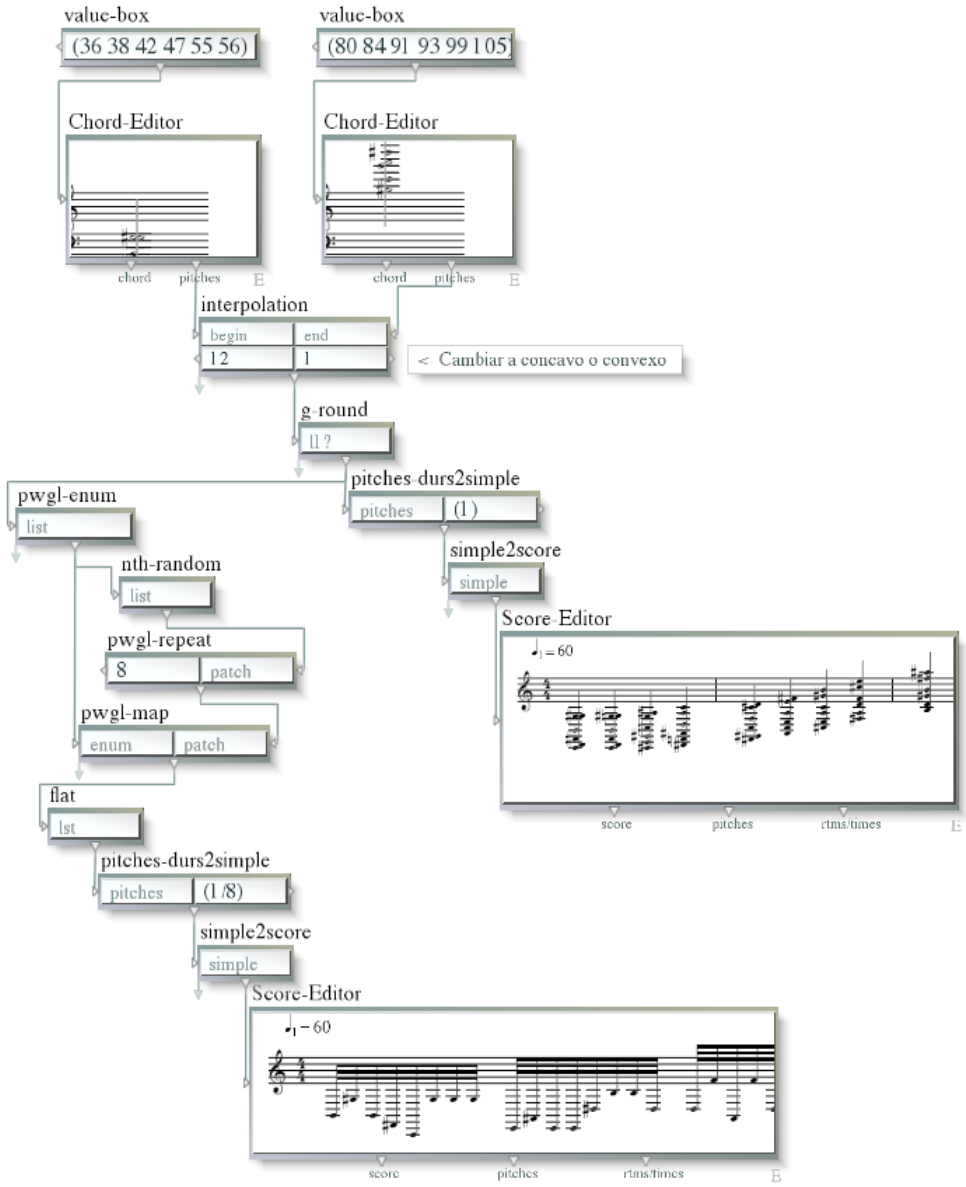


34-KSQuant

Interpolación de acordes

Veamos un ejemplo de aplicación de los objetos de notación musical vistos anteriormente. Vamos a realizar una interpolación lineal entre dos acordes, generando diez nuevos acordes intermedios. Para ello, utilizamos el objeto *interpolation*. El argumento *curves* de este objeto establece el tipo de curva de interpolación; el 1 corresponde a una recta, un número menor que uno genera una curva convexa, y un número mayor que uno genera una curva cóncava. Luego redondeamos los valores a entero para obtener notas MIDI temperadas. La representación musical de la interpolación se observa en la rama de la derecha del *patch*.

Al mismo tiempo, a la izquierda, producimos un acorde por negra, pero con 8 notas escogidas al azar en fusas. Para ello, extraemos cada acorde con *pwgl-enum*, obtenemos un elemento de la lista-acorde al azar con *nth-random*, y repetimos el procedimiento 8 veces con *pwgl-repeat*. Finalmente, eliminamos los paréntesis con *flat*, y representamos en notación musical.



35-Interpolación

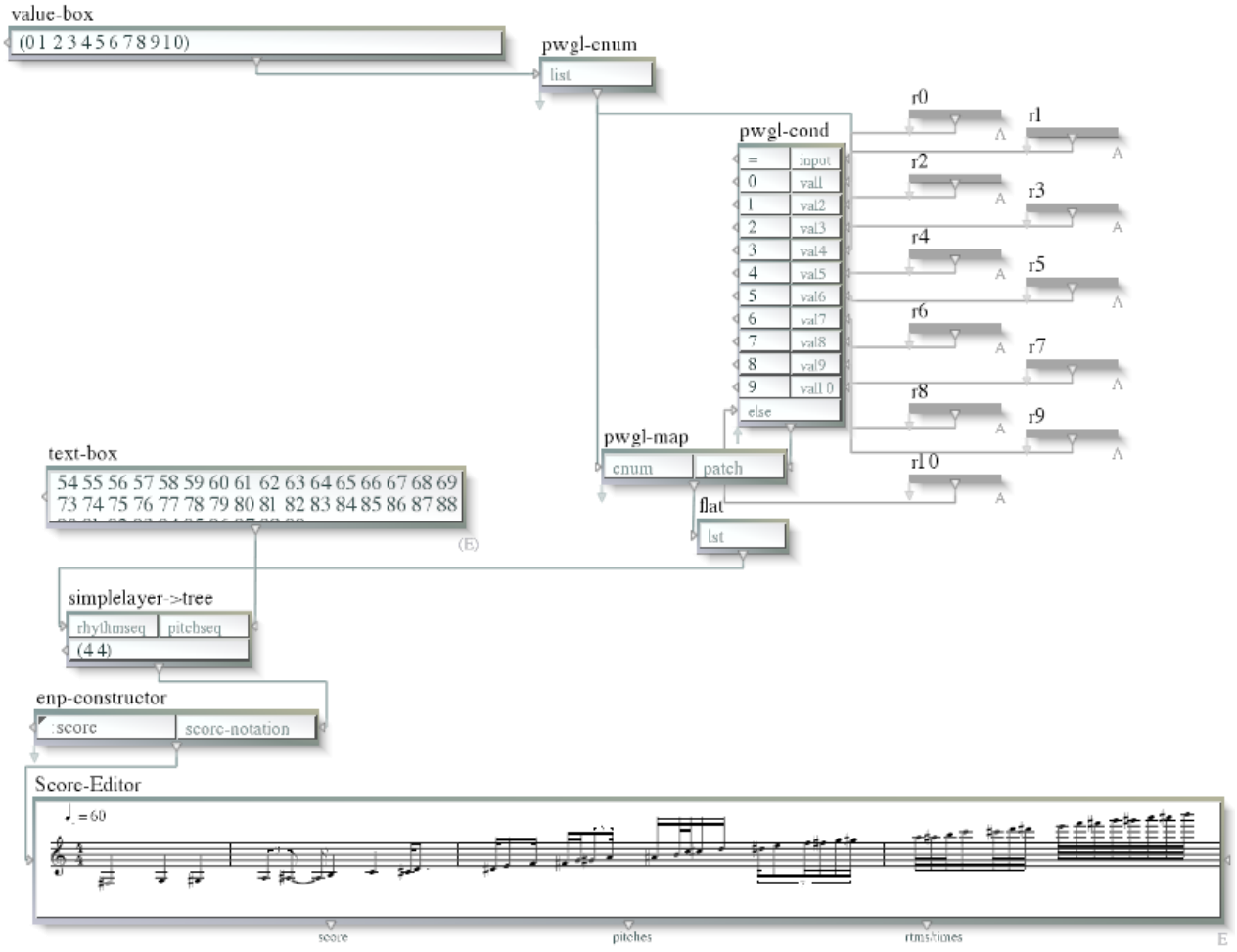
Ritmo-Altura

La librería *studio-flat* cuenta con tres objetos que resultan de utilidad al vincular secuencias rítmicas con secuencias de alturas e intensidades. El objeto *simplelayer->tree* convierte una sucesión de notas MIDI y sus duraciones respectivas en una lista anidada, cuyo formato es reconocido como una notación válida para el objeto *enp-constructor*.

En el siguiente ejemplo vamos a construir ritmos al azar, ordenados por cantidad de ataques por tiempo.

Se observa en el *patch* la enumeración de los elementos de una lista, que en nuestro caso son números entre 0 y 10. El número 0 representa a la blanca (1/2). El 1 representa a dos eventos por blanca, pudiendo darse el caso de dos negras (1/4 1/4), negra con puntillo y corchea (3/8 1/8), corchea y negra con puntillo (1/8 3/8), blanca y negra en un tresillo (1/3 1/6), y negra y blanca en un tresillo (1/6 1/3). Alguna de esas opciones va a ser elegida al azar. El número 2 representa a células con tres elementos por blanca, el número 3 a una duración por negra (1/4), el 4 a células de dos eventos por negra, el 5 a células de tres eventos por negra, y así siguiendo hasta alcanzar 8 eventos por negra. De acuerdo a esto, podemos inferir que los números de la lista remiten a un índice de densidad cronométrica, y que las células rítmicas que obtenemos responden a un criterio de cantidad de eventos por unidad de tiempo.

Las abstracciones que comienzan con la letra *r* devuelven una célula rítmica al azar, a excepción de *r*0 (siempre una blanca), *r*3 (siempre una negra) y *r*10 (siempre 8 fusas).

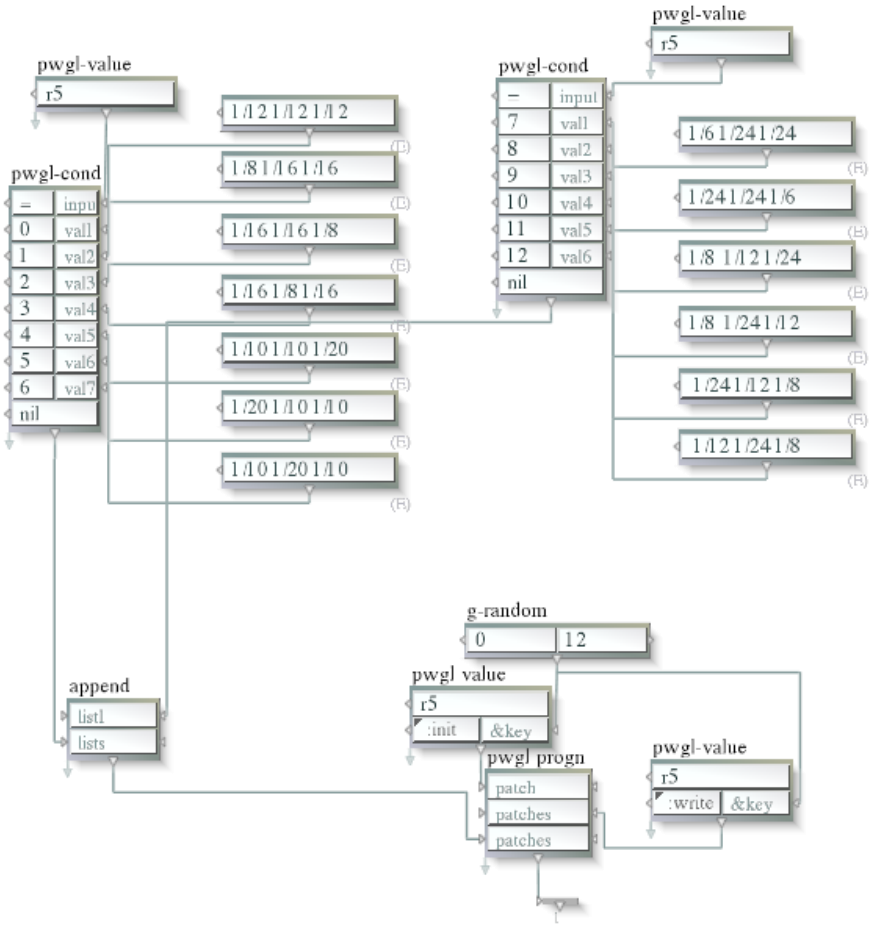


36-Ritmo-Altura

El objeto *pwgl-enum* extrae cada uno de los números de la lista de densidad cronométrica, y el objeto *pwgl-cond* evalúa a la abstracción *r* que coincide con el número obtenido. Una vez finalizada la enumeración, y la generación de todas las células que responden a los índices de densidad, se eliminan paréntesis sobrantes. De este modo se conforma una lista que contiene a todas las duraciones de la secuencia rítmica.

Al observar el resultado en notación musical, notamos con mayor claridad el rol que juega la lista de índices del ejemplo, que va de 0 a 10.

En la figura *abstracción r5* podemos apreciar el modo en que se elige una célula al azar, en este caso entre 13 candidatas, formada por 3 elementos por negra. Utilizamos aquí dos condicionales, de 0 a 6 y de 7 a 12, por la simple razón que un único objeto condicional sólo permite evaluar hasta 10 posibilidades distintas. El número de célula elegido al azar debe mantenerse constante, tanto para un condicional como para el otro, por lo cual se lo ingresa en la variable *r5*. Notar también el uso de *pwgl-progn*, que evalúa la inicialización de la variable *r5*, la asignación del número al azar en la variable, y finalmente evalúa el resto y devuelve la célula elegida. El objeto *append* se usa aquí para unir los dos condicionales, pues el resultado puede aparecer en cualquiera de ellos. Uno de los dos devuelve una célula, mientras que el otro devuelve *NIL*.



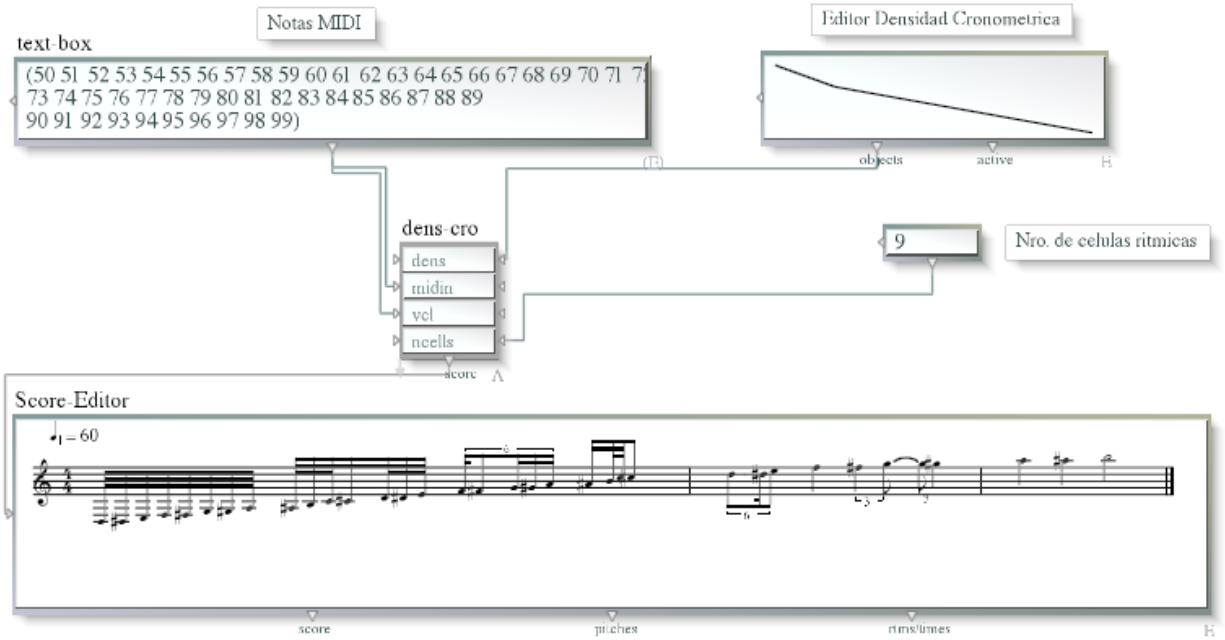
Abstracción *r5*

Densidad cronométrica

En este *patch* partimos del ejercicio anterior para especificar la densidad cronométrica a través de una curva. A través de estas modificaciones intentamos acercarnos a la idea de generar gestos musicales, estableciendo la variación de distintos parámetros a través de funciones.

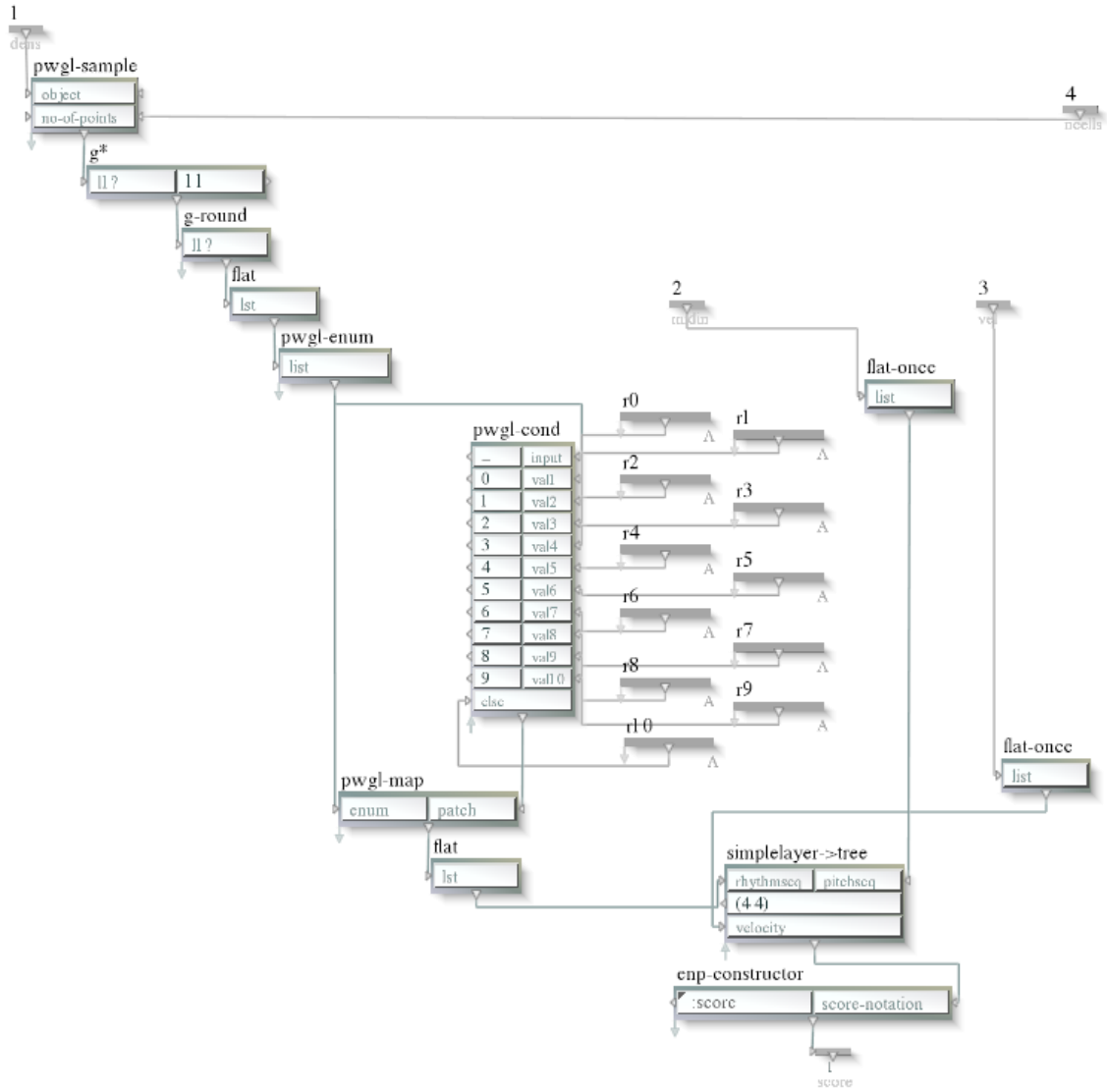
Resulta ahora necesario establecer el número de células rítmicas por medio de un *num-box*. Ese número se emplea como la cantidad de muestras a extraer de la curva, con el objeto *pwgl-sample*.

La lista de alturas MIDI se emplea también, en este ejemplo, como lista de *key velocities*.



37-Densidad cronométrica

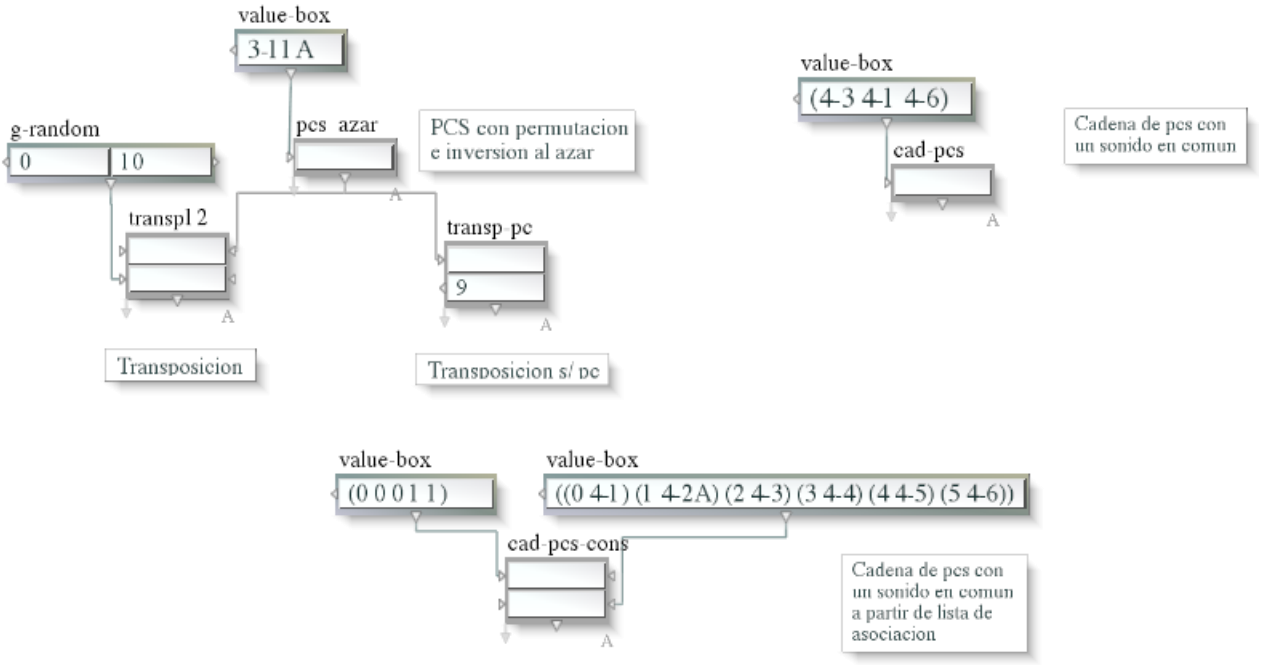
La figura que sigue muestra el contenido de la abstracción *dens-cro*, donde se pueden apreciar las reformas introducidas.



Abstracción *dens-cro*

Conjuntos de grados cromáticos

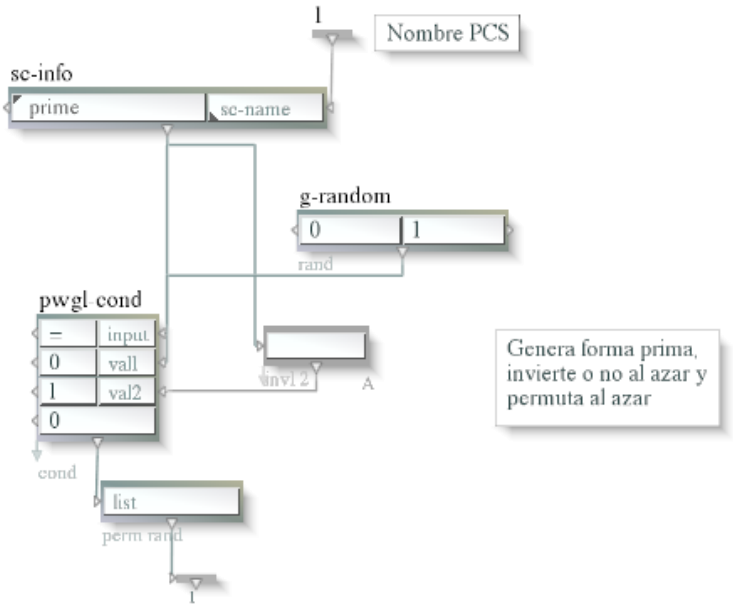
Realizaremos ahora tres ejemplos utilizando conjuntos de grados cromáticos (*Pitch Class Sets*). El primero de ellos genera un conjunto a partir de su nombre, cuyo índice *TnI* y permutación son establecidos de manera aleatoria. El segundo, produce una cadena de PCS con una nota en común, a partir de sus nombres. Por último, el tercer *patch* genera una cadena de PCS mediante una lista de nivel de consonancia, y otra lista de asociación que vincula cada índice de consonancia con un PCS en particular.



38-PCS

El objeto *sc-info* (que se encuentra en la opción de menú *PC-set-theory*) recibe el nombre de un PCS y devuelve su forma prima. En PWGL los nombres de los PCS difieren si están o no en inversión. El PCS 4-2, por ejemplo, se denomina 4-2A en estado original, y 4-2B en inversión. No obstante, aquellos PCS que poseen invariantes por transposición-inversión, se denominan simplemente por su nombre sin letras agregadas (por ejemplo, 4-1).

La abstracción *pcs-azar* obtiene la forma prima con *sc-info*, decide su inversión o no con *g-random* y *pwgl-cond*, y permuta al azar con *pwgl-perm*. Se aprecia la programación en la figura que sigue.

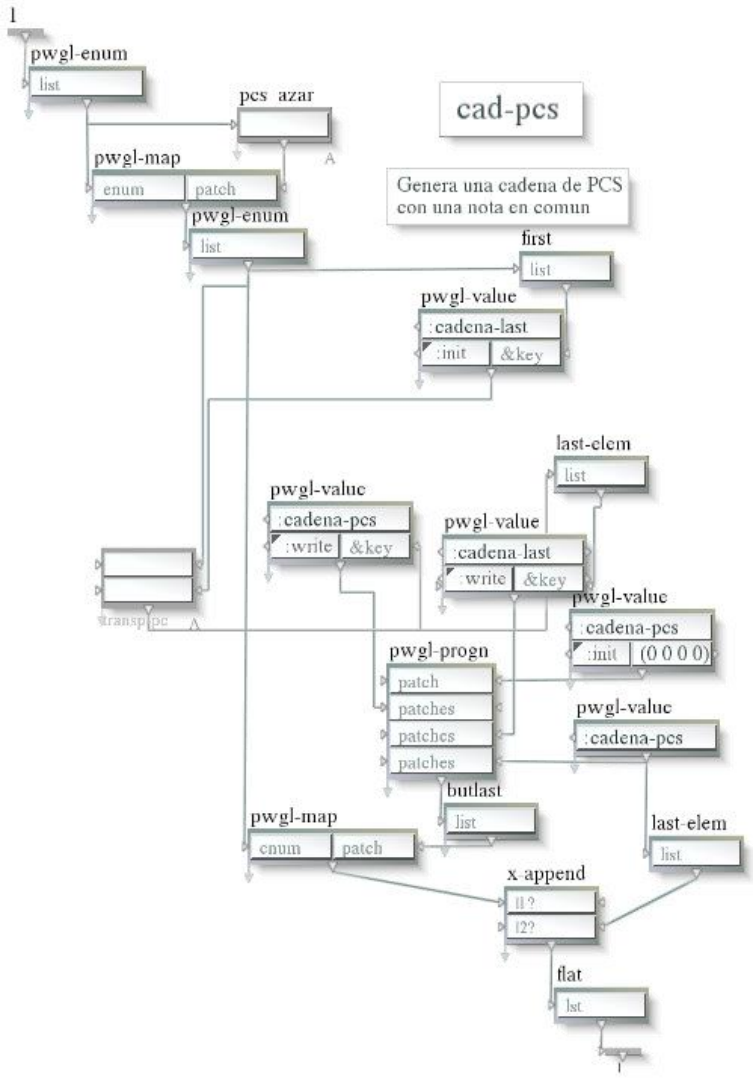


Abstracción *pcs-azar*

Para armar una cadena de PCS, con una nota en común, es preciso transportar cada conjunto de forma tal que la primera nota del conjunto siguiente sea igual a la última nota del conjunto anterior. También es necesario considerar que la primera nota de cada conjunto, a excepción del primero, debe ser descartada para que no existan notas repetidas en la unión de dos PCS consecutivos.

La abstracción que produce estas cadenas se analiza a continuación:

1. Se genera una lista de PCS al azar, a partir de enumerar la lista de nombres, y cada PCS es ubicado en una sublista.
2. Se enumeran las sublistas de esa lista por medio de un bucle. En la primera pasada se inicializa una variable denominada *cadena-last* con el primer elemento (*first*) de la primera sublista. En las siguientes pasadas, sólo se lee el valor de *cadena-last*, que es asignado más adelante.
3. Se transportan los grados de la sublista sobre el grado guardado en *cadena-last*.
4. Con *pwgl-progn* se evalúan en orden las siguientes acciones: a) inicialización de la variable *cadena-pcs*. b) la sublista transpuesta se guarda en *cadena-pcs*. c) Se almacena el último elemento de la sublista transpuesta en *cadena-last*. d) Se obtiene la última sublista guardada en *cadena-pcs*. e) De esa sublista se toman todos los elementos menos el último, con *butlast*.
5. Una vez finalizado el bucle de PCS, se agrega el último elemento de la secuencia, con *append*, y se eliminan los paréntesis con *flat*.

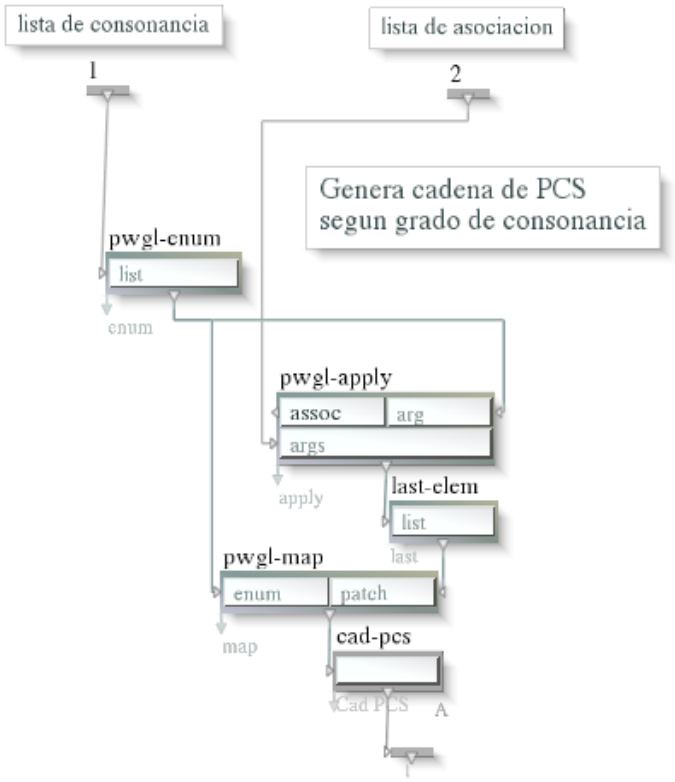


Abstracción *cad-pcs*

El tercer ejemplo es similar al anterior, salvo que se emplea una lista de asociación para vincular un índice de consonancia con un PCS en particular. La cadena no se arma a partir de los nombres de los PCS, sino de una lista de niveles de consonancia-disonancia. En nuestro ejemplo, el número 0 representa el mayor grado de disonancia, y el 5 el menor grado.

En la figura siguiente se aprecia el *subpatch* que realiza estas acciones.

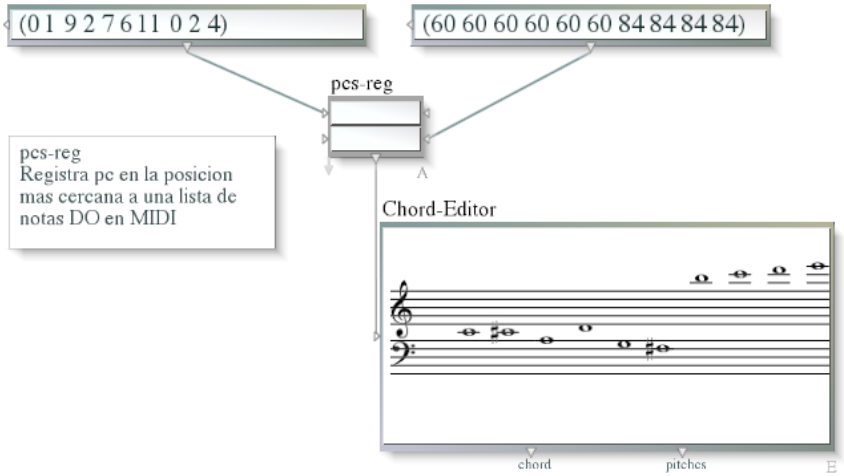
El objeto *pwgl-apply* invoca a la función primitiva de LISP *assoc*, que asocia al primer elemento (índice de consonancia) de cada sublista, con el segundo elemento (nombre de PCS). De cada asociación se toma el último elemento (nombre del PCS), y la lista de nombres resultante se pasa a la abstracción que genera la cadena.



Abstracción *cad-pcs-cons*

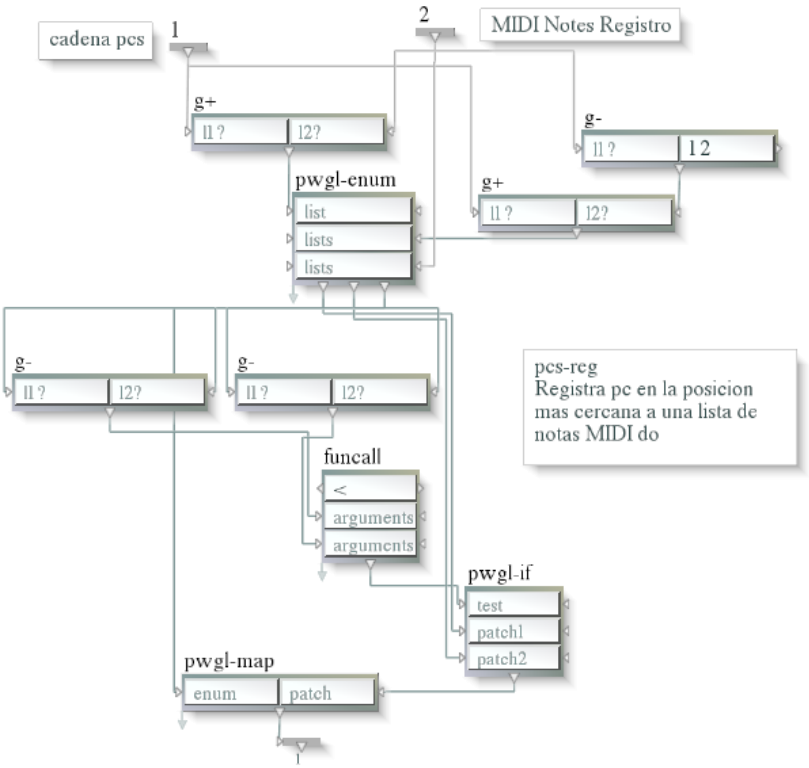
Registro

El siguiente *patch* ubica grados cromáticos en el registro, partiendo de una lista de notas MIDI *do* que sirven de referencia.



39-Registro

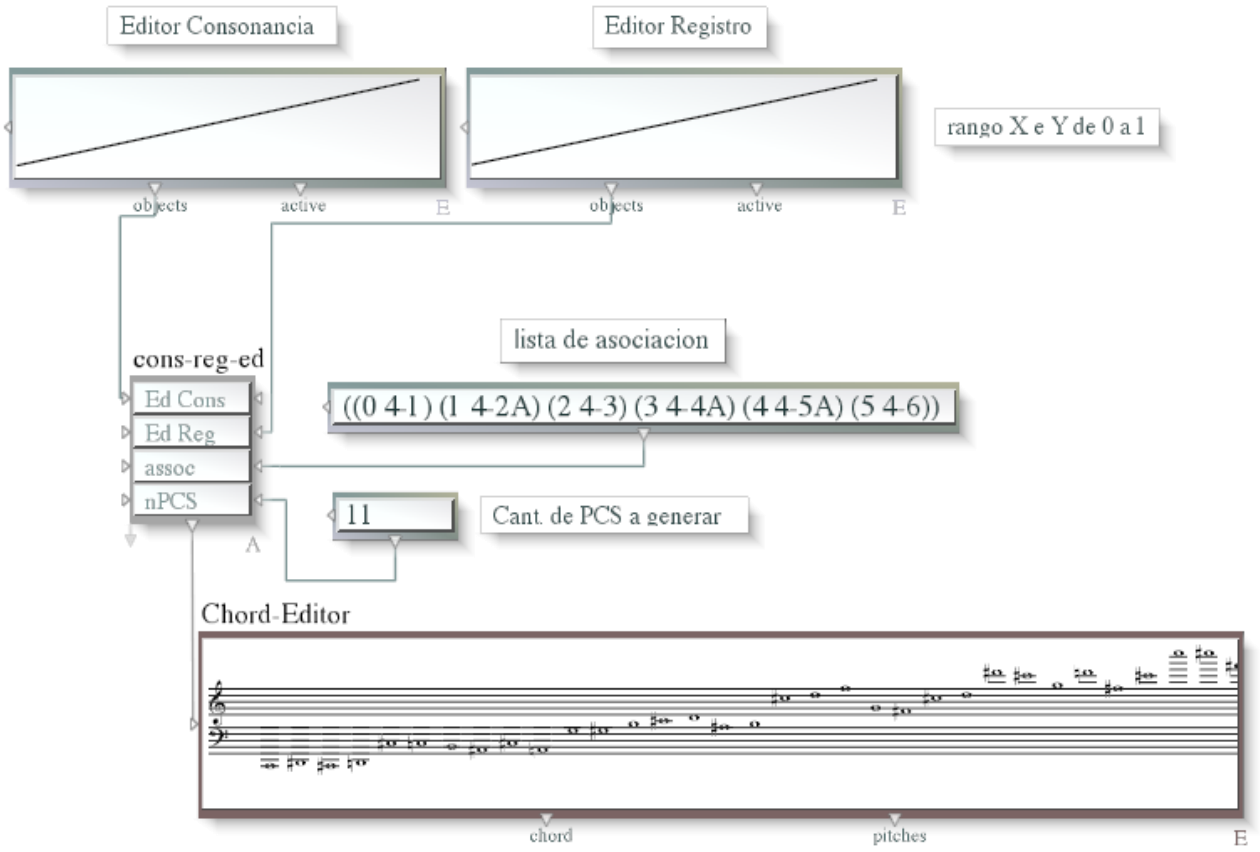
Se suman los grados de un conjunto a una nota MIDI *do* de referencia (48, 60, 72, etc.). Paralelamente se suman los mismos grados a las notas *do* de referencia, pero estas últimas una octava abajo. Y por último, se elige la versión más cercana a la nota *do* de referencia. Se observa el uso de *pwgl-enum* expandido, con tres entradas y tres salidas.



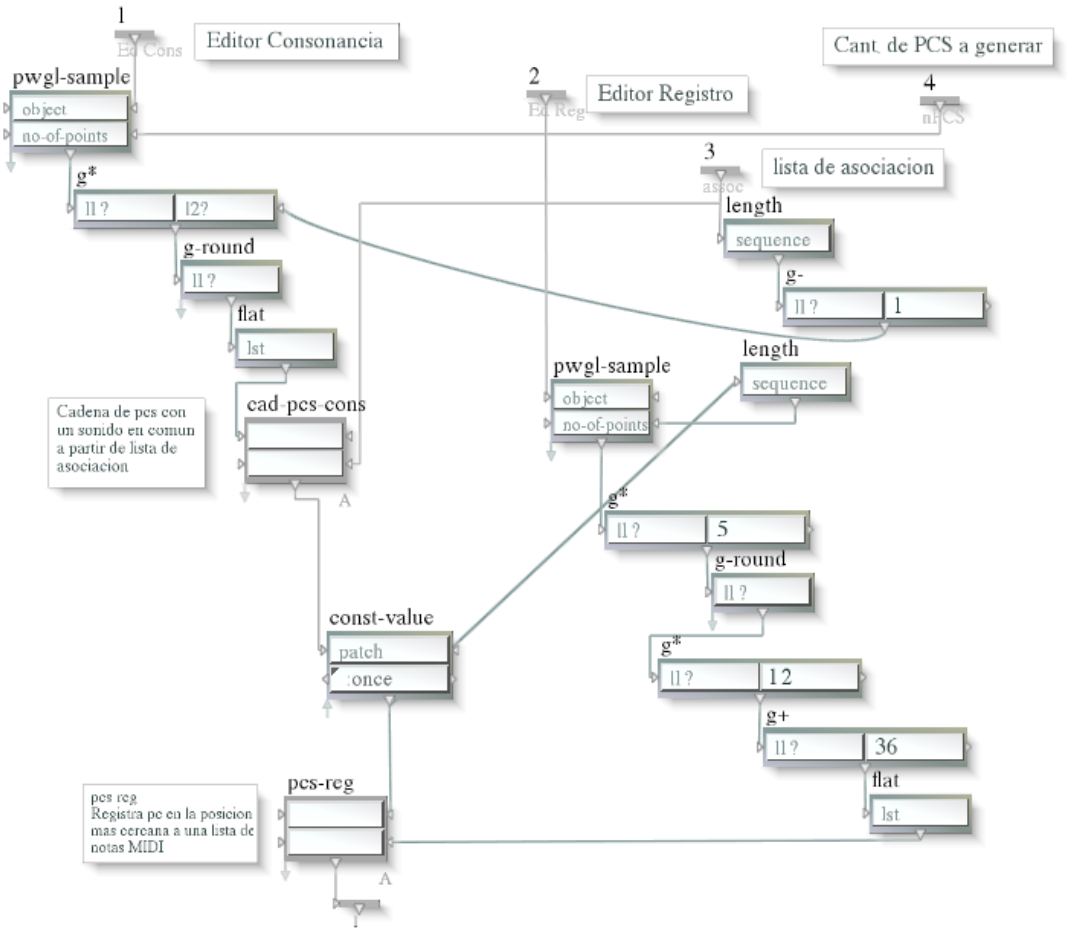
Abstracción *pcs-reg*

Consonancia

En el programa que sigue vinculamos el nivel de consonancia y la ubicación de las notas en el registro por medios gráficos. Incorporamos como datos una curva de consonancia y otra de registro (ambas con rango y dominio de 0 a 1), una lista de asociación de niveles de consonancia con PCS, y la cantidad de PCS a generar. Los valores *x* e *y* de la curva de consonancia se escalan en función del número de conjuntos a considerar y de la longitud de la lista de asociación. En el caso de la curva de registro, se escalan de acuerdo al número de PCS a generar y de un ámbito limitado entre las notas MIDI 36 y 96.



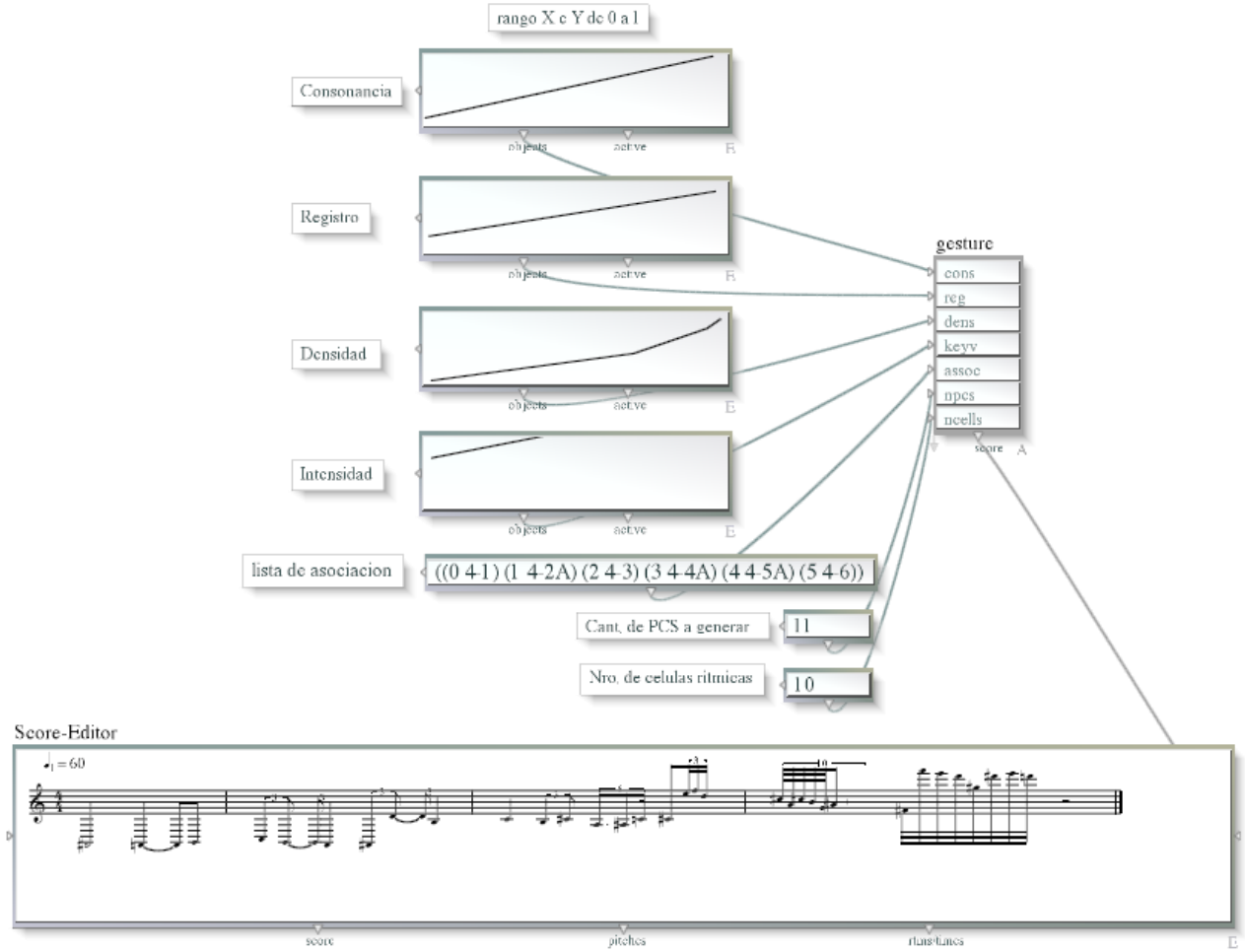
40-Consonancia



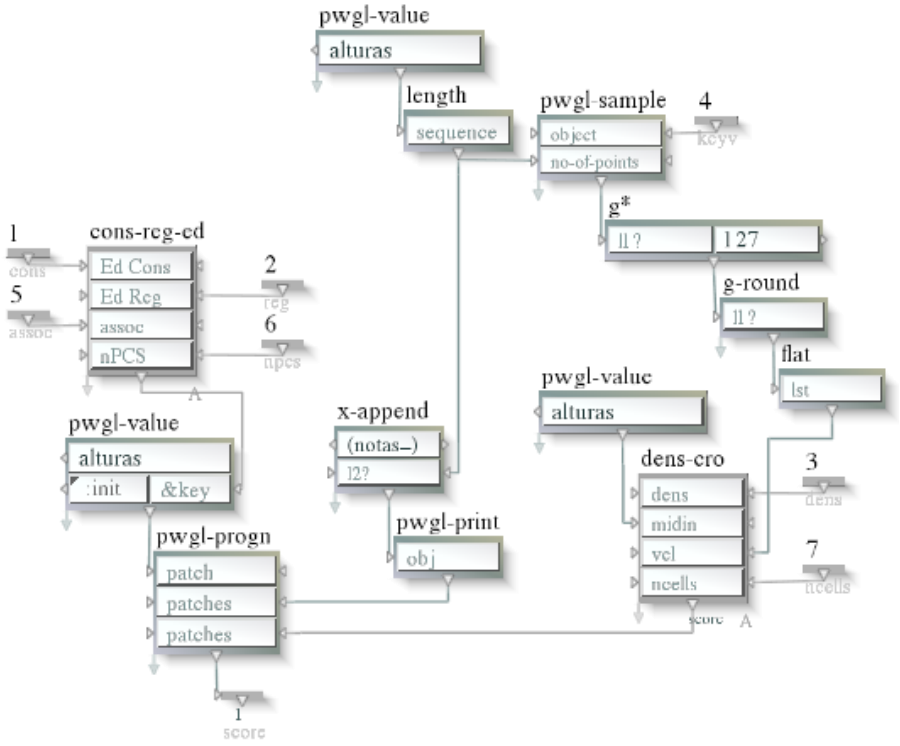
Abstracción *cons-reg-ed*

Gestos

Finalmente, podemos relacionar las aplicaciones rítmicas ya tratadas, con el *patch* de la figura 40, con el propósito de producir pequeños gestos musicales, estableciendo variaciones de consonancia-disonancia, registro, densidad cronométrica, y dinámica, por medios gráficos.



41-Gestos



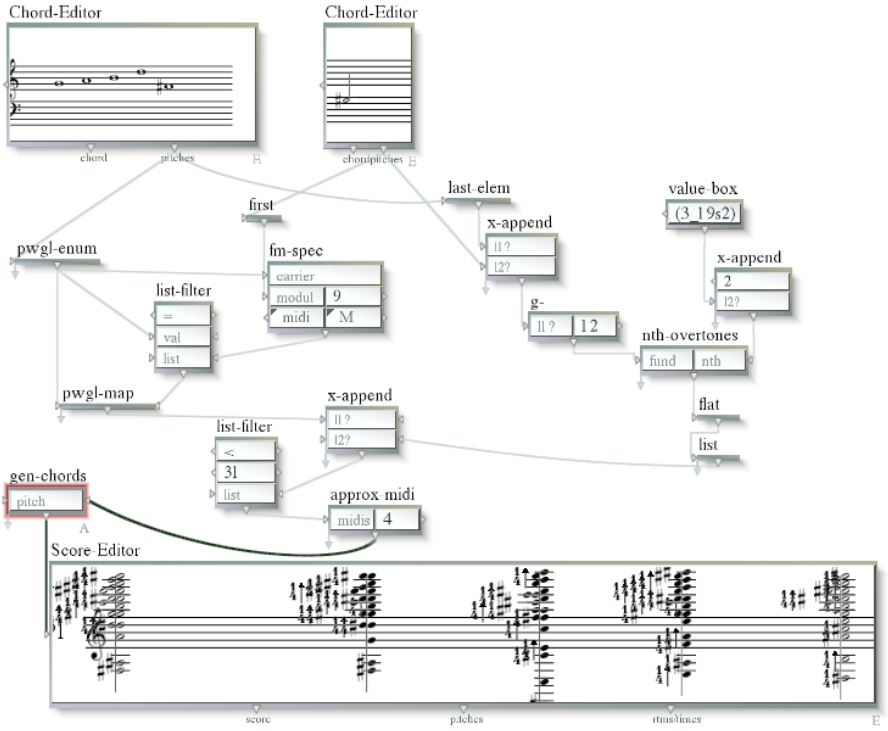
Abstracción *gesture*

Armónicos y Frecuencia Modulada

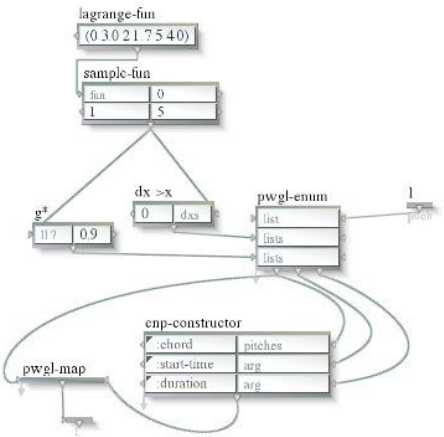
En este último *patch* vemos el uso de algunos objetos de la librería *Esquisse*, desarrollada originalmente por Tristan Murail para el entorno *Patchwork*. La programación corresponde a la generación de alturas microtonales a partir de escalas de armónicos y síntesis por frecuencia modulada (Chowning, 1973), utilizadas por Murail en su obra para orquesta *Gondwana* (1989).

Un análisis de las técnicas empleadas en la obra puede encontrarse en Rose (1996).

El objeto *fm-spec* computa un espectro por frecuencia modulada, con índices de modulación entre 1 y 25. El objeto *nth-overtones* genera las notas MIDI (en midicents) que corresponden a una lista de armónicos de una fundamental dada. Las frecuencias son aproximadas al cuarto de tono.



42-Gondwana



Abstracción *gen-chords*

BIBLIOGRAFÍA

- Berbiela, J. *Guía de referencia LISP*.Ra-ma. Madrid. 1991.
- Cetta, P. "Modelos de localización espacial del sonido y su implementación en tiempo real". En *Altura – Timbre – Espacio*. Cuaderno Nº 5 del Instituto de Investigación Musicológica "Carlos Vega", pp. 269-292. EDUCA. Buenos Aires. 2004. (ISBN 987-1190-13-1).
- Cetta, P. "Procesamiento en tiempo real en la obra de Luigi Nono". En *Altura – Timbre – Espacio*. Cuaderno Nº 5 del Instituto de Investigación Musicológica "Carlos Vega", pp. 235-257. EDUCA. Buenos Aires. 2004. (ISBN 987-1190-13-1).
- Chowning, J. "The synthesis of complex audio spectra by means of frequency modulation". *Journal of the Audio Engineering Society*, 21:526-534, 1973.
- Kuuskankare, M. y Laurson, M. "ENP-Expressions, Score-BPF as a Case Study". En Proc. ICMC03, pp. 103-106, Singapur, Sept. 2003.
- Kuuskankare, M. y Laurson, M. "Intelligent Scripting in ENP using PWConstraints". En Proc. ICMC04, Miami.
- Kuuskankare, M. y Laurson, M. "Expressive Notation Package". Computer Music Journal, vol. 30, no. 4, 2006.
- Kuuskankare, M. y Laurson, M. "Recent developments in ENP score notation". En Sound and Music Computing 04, Octubre de 2004. http://smc04.ircam.fr
- Laurson, M. y Kuuskankare, M. *PWGL Book*. Publicación on-line. http://www2.siba.fi/pwgl/downloads.html
- Laurson, M. y Kuuskankare, M. "PWGL: A Novel Visual Language based on Common Lisp, CLOS and OpenGL". In *Proceedings of International Computer Music Conference*, pp. 142–145, Gothenburg, Sweden. 2002.
- Laurson, M. y Kuuskankare, M. "Some Box Design Issues in PWGL". En Proc. ICMC03, pp. 271-274, Singapur, Sept. 2003.
- Laurson, M. y Kuuskankare, M. "PWGL Editors: 2D-Editor as a Case Study". En Proc. SMC04, 2004.
- Laurson, M., Norilo, V. y Kuuskankare, M. "PWGLSynth, A Visual Synthesis Language for Virtual Instrument Design and Control". Computer Music Journal, vol. 29, no. 3, pp. 29-41, 2005.
- Rose, F. "Introduction to the Pitch Organization of French Spectral Music." *Perspectives of New Music* 34, no. 2 (Summer): 6–39. 1996. Traducción del inglés de E. Checchi y P. Cetta.
- Seibel, P. *Practical Common LISP*. Apress. New York. 2005.
- Touretzky, D. *COMMON LISP: a gentle introduction to symbolic computation*. The Benjamin/Cumming Publishing Company. Redwood CA. 1990.