Introducción al lenguaje Faust

El objetivo de este documento es ayudarles a familiarizarse con el lenguaje Faust a través de ejemplos muy simples de procesamiento de señales y síntesis de sonido. Para mantener la sencillez, se explican los principios básicos de operación de Faust, pero no se incluye la documentación completa de cada elemento de su sintaxis, para ello referirse al documento correspondiente.

Para trabajar en los ejemplos se recomienda utilizar la IDE (Entorno de Desarrollo Integrado) disponible online:

- **https://faustide.grame.fr**

Dado que trabajaremos en el navegador web, es posible que los sonidos producidos con la IDE presenten algunos clics, se puede usar el editor online, que es más simple, pero más ligero:

- **https://fausteditor.grame.fr**

Faust en pocas palabras

- Faust es un lenguaje de programación para el procesamiento y síntesis de señales en tiempo real (como Csound, Max/MSP, Supercollider, Puredata, etc...).
- Faust se basa en un enfoque puramente funcional.
- Un programa Faust describe un procesador de señal digital (DSP): ***una función que asigna señales de entrada a señales de salida***.
- La programación en Faust se basa esencialmente en combinar estos procesadores de señal por medio de un ***álgebra*** de 5 operaciones: `<: :> : , ~`
- Faust es un lenguaje compilado, en donde el rol del compilador es producir (o sintetizar) las implementaciones más eficientes posibles para el procesador que hemos diseñado.
- Faust ofrece a los usuarios finales una alternativa de alto nivel al lenguaje C para desarrollar aplicaciones de audio para una gran variedad de plataformas.

Contenidos | Parte 1

Parte 1: ejemplos muy simples

Comencemos con algunos ejemplos simples de programas en Faust. Se puede copiar y pegar el código en la IDE online, o utilizar los enlaces “Pruébalo aquí >>”.

Ejemplo 1: el programa Faust más simple

Este es el programa Faust más simple imaginable. Contiene una sola línea de código, la definición: `process = _;`.


```
process = _;
```

Pruébalo aquí >>

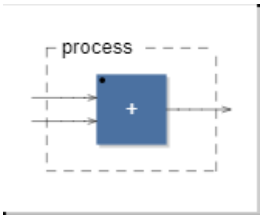
Se pueden aprender varias lecciones de este ejemplo tan simple:

- Un programa Faust tiene como mínimo una ***definición***, la de la palabra clave `process` que indica el punto de entrada del programa.
- Una definición siempre termina con el caracter `;` (al igual que el lenguaje C y derivados). Un error común es olvidar el punto y coma al final de una definición.
- El guión bajo `_` representa una de las ***primitivas*** del lenguaje (las primitivas son las funciones predefinidas del lenguaje). Representa un ***cable de audio*** que deja pasar la señal sin transformarla. Esto es lo que se llama en matemáticas la función de ***identidad***, la señal que ingresa a la izquierda sale idénticamente a la derecha.

Ejemplo 2: Agregar dos señales

Vimos en el ejemplo anterior la primitiva `_`. Faust tiene una gran cantidad de primitivas, incluidas todas las operaciones matemáticas.

La primitiva `+`, por ejemplo, se usa para agregar dos señales. Por lo tanto, se puede usar para transformar una señal estereofónica (en dos canales) en una señal monofónica como en el siguiente ejemplo:

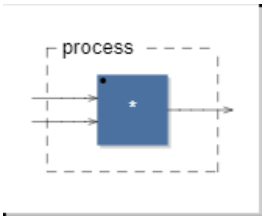


```
process = +;
```

Pruébalo aquí >>

Ejemplo 3: Multiplicar dos señales

La primitiva `*` se usa para multiplicar dos señales.




```
process = *;
```

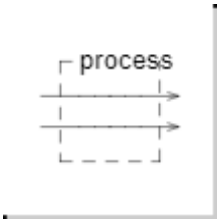
Pruébalo aquí >>

Como puede escucharse, multiplicar los dos canales de una señal entre ellos transforma el sonido de manera drástica…

Ejemplo 4: composición paralela

La programación en Faust consiste en ensamblar operaciones primitivas para formar circuitos de audio más o menos complejos. Para llevar a cabo estos ensambles, Faust tiene 5 operaciones de composición: `~`, `,`, `;`, `<:`, `:>`.

Primero veamos la ***composición paralela***, representada por la coma `,`.




```
process = _, _;
```

Pruébalo aquí >>

Con esto, hicimos un ***cable estéreo***, y cuando reproducimos un archivo de audio, ahora lo escuchamos en ambos altavoces. Habrán notado que hasta aquí solo escuchábamos por el altavoz izquierdo.

Es muy importante distinguir entre ***primitivas***, como `_`, `+` o `*`, y ***operaciones de composición*** como `,` o `;`. Las ***primitivas*** representan **operaciones en señales de audio**,

mientras que las operaciones de composición se utilizan para **vincular dos operaciones** de audio. En otras palabras, se puede escribir `+` o `*` solos, porque representan operaciones de audio válidas, pero nunca se pueden escribir `,` o `:` solos porque se usan para interconectar dos operaciones de audio. Siempre se debe escribir `A,B` o `A:B`.

Las primitivas de Faust están organizadas en varias categorías. Encontramos todas las funciones numéricas del lenguaje C, pero aplicadas a las señales de audio:

Categoría	Primitivas
Aritmética	`+`, `-`, `*`, `/`, ...
Comparación	`<`, `==`, `!=`, `<=`, ...
Trigonométricas	`sin`, `cos`, ...
Logaritmos y exponentes	`log`, `exp`, ...
Mínimo y máximo	`min`, `max`, ...
Selectores	`select2`, `select3`, ...
Retardos y tablas	`@`, `rdtable`, ...
Interfaz gráfica (GUI)	`hslider()`, `button()`, ...

Aquí hay una tabla resumen de los cinco operadores de composición:

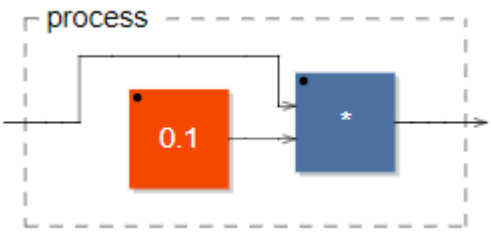
Sintaxis	Prioridad	Asociación	Descripción
`A ~ B`	4	izquierda	Composición recursiva
`A , B`	3	derecha	Composición paralela
`A : B`	2	derecha	Composición secuencial
`A <: B`	1	derecha	Composición divergente
`A :> B`	1	derecha	Composición convergente

(Mayor prioridad cuanto mayor sea su número de prioridad)

Ejemplo 5: control del volumen

Veamos un ejemplo en el que se combinan tres primitivas: `_`, `0.1` y `*`, con dos operadores de composición: `,` y `:`.

La idea aquí es reducir el volumen de la señal entrante a una décima parte de su valor inicial. Esto se hace multiplicando la señal entrante por `0.1` (1/10=0.1):




```
process = (_, 0.1) : *; // probar reemplazar el 0.1 por otros valores entre 0 y 1
```

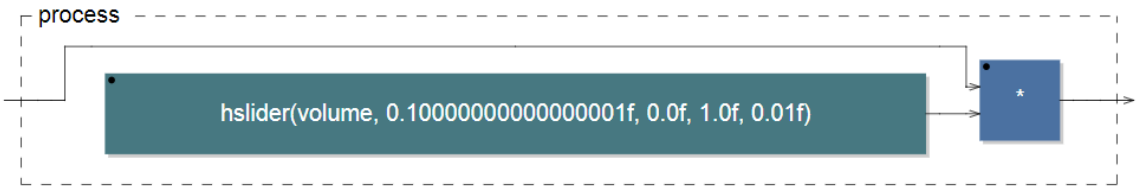
Pruébalo aquí >>

Tengamos en cuenta que hemos utilizado paréntesis en este ejemplo para especificar claramente el orden en que se deben hacer las cosas. Comenzamos poniendo `_` y `0.1` en paralelo, y luego los compusimos en secuencia con `*`.

Pero, al igual que en `(2*3)+7`, puede darse el caso de que los paréntesis no sean realmente necesarios, porque la multiplicación tiene prioridad sobre la suma. Podríamos escribir directamente `process = _, 0.1 : *;` sin los paréntesis, porque la composición paralela tiene prioridad sobre la composición secuencial. La prioridad de los operadores de composición se muestra en la tabla anterior.

Ejemplo 6: controlar el volumen con un control deslizante

En lugar de controlar el volumen editando el código, es mucho más conveniente usar un control deslizante gráfico. Para ello, podemos usar el `hslider(...)`, un control deslizante horizontal. Se necesitan cinco ***parámetros***. El primero es un nombre a elección como `"volume"`, luego tenemos el valor predeterminado `0.1`, el valor mínimo `0`, el valor máximo `1` y un valor (o intervalo) de paso `0.1`.

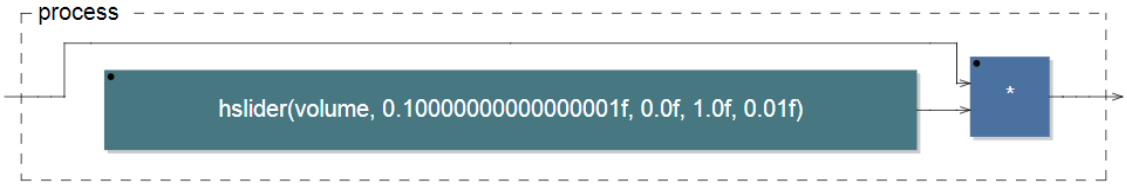



```
process = _, hslider("volume", 0.1, 0, 1, 0.01) : *;
```

Pruébalo aquí >>

Ejemplo 7: amplificador mono

Hemos escrito programas muy simples hasta ahora, que caben en una sola línea de código. Ahora presentaremos ***definiciones adicionales***. Una definición debe entenderse como una forma de dar un nombre a algo, lo que nos evita escribir la definición cada vez y hace que el programa sea más fácil de entender.

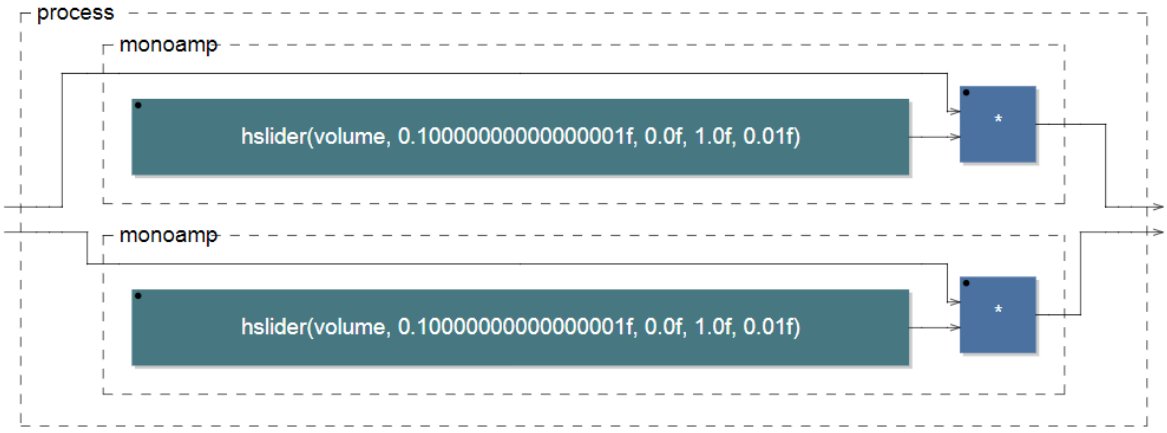



```
monoamp = _, hslider("volume", 0.1, 0, 1, 0.01) : *;
process = monoamp;
```

Pruébalo aquí >>

Ejemplo 8: amplificador estéreo

Continuando en la misma idea, definiremos un amplificador estéreo como dos amplificadores mono en paralelo.



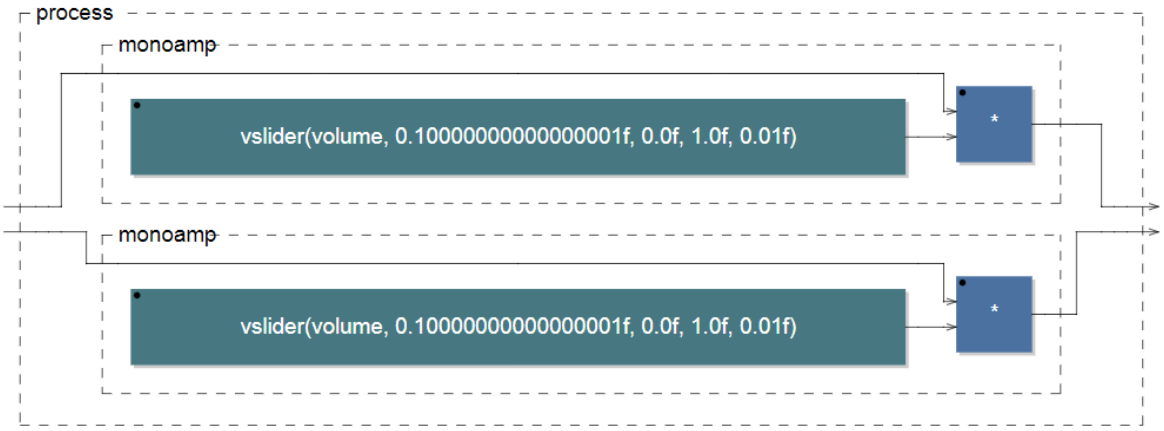

```
monoamp = _, hslider("volume", 0.1, 0, 1, 0.01) : *;
stereoamp = monoamp,monoamp;

process = stereoamp;
```

Pruébalo aquí >>

Tener en cuenta que incluso si el hslider de volumen aparece varias veces en nuestro código, solo habrá uno en la interfaz de usuario.

Ejemplo 9: deslizadores verticales

En lugar de controles deslizantes horizontales, podemos usar controles deslizantes verticales. Simplemente reemplazaremos hslider (...) por vslider (...).



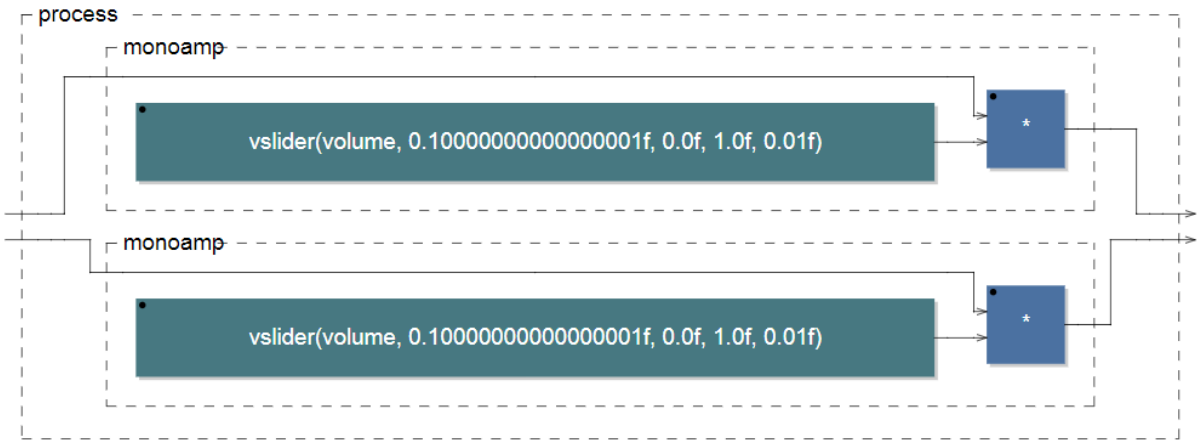

```
monoamp = _, vslider("volume", 0.1, 0, 1, 0.01) : *;
stereoamp = monoamp,monoamp;

process = stereoamp;
```

Pruébalo aquí >>

Ejemplo 10: perillas en lugar de controles deslizantes

Por defecto los controles deslizantes son... ¡controles deslizantes! Pero podemos cambiar su apariencia utilizando el mecanismo de ***metadatos***.

Los metadatos son información que se coloca entre corchetes en el nombre del control deslizante. Por ejemplo, el metadato "... [style:knob] ..." permite convertir el control deslizante en una perilla giratoria.



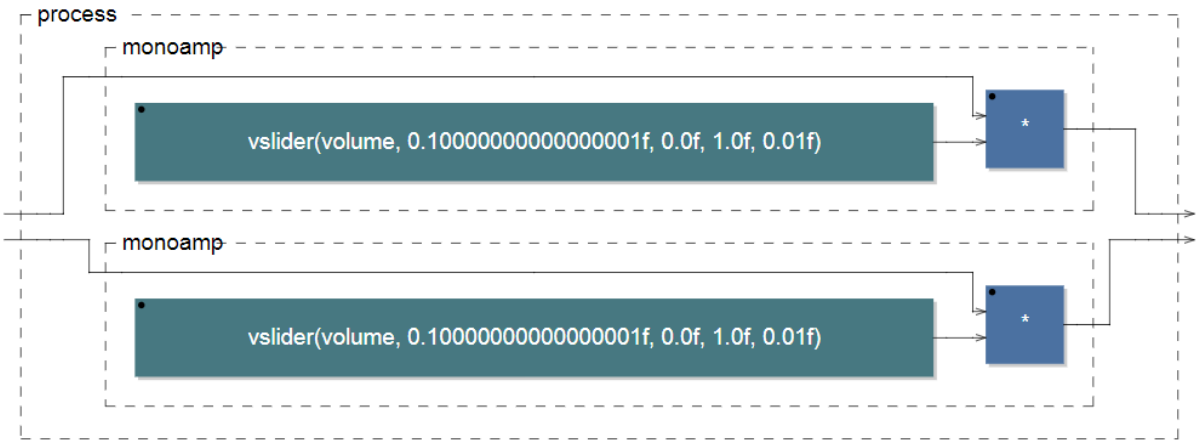
```
monoamp = _, vslider("volume[style:knob]", 0.1, 0, 1, 0.01) : *;

stereoamp = monoamp,monoamp;

process = stereoamp;
```

Pruébalo aquí >>

Ejemplo 11: endulzando la sintaxis

Hemos usado la sintaxis principal de Faust hasta ahora. Por ejemplo, para multiplicar una señal de entrada por `0.1` hemos escrito `_,0.1:*`. Para las expresiones numéricas, esta notación no siempre es la más conveniente y, a veces, preferiremos utilizar la notación ***infija***, más tradicional e intuitiva, y escribir en su lugar `_*0.1`. También podemos usar la notación ***prefija*** y escribir `*(0.1)`. Los nombres pueden resultar confusos, pero los conceptos son muy simples y es más importante comprender que tenemos distintas alternativas para un mismo resultado.

Reescribamos la definición del amplificador mono usando la notación de prefijo:




```
monoamp = *(vslider("volume[style:knob]", 0.1, 0, 1, 0.01));

stereoamp = monoamp,monoamp;

process = stereoamp;
```

Pruébalo aquí >>

A continuación se muestra una tabla de notaciones equivalentes donde las expresiones de infijo y prefijo se traducen a la sintaxis principal.

Expresión	Descripción
`_,0.1:*`	sintaxis principal
`_*0.1`	notación infija
`*(0.1)`	notación de prefijo

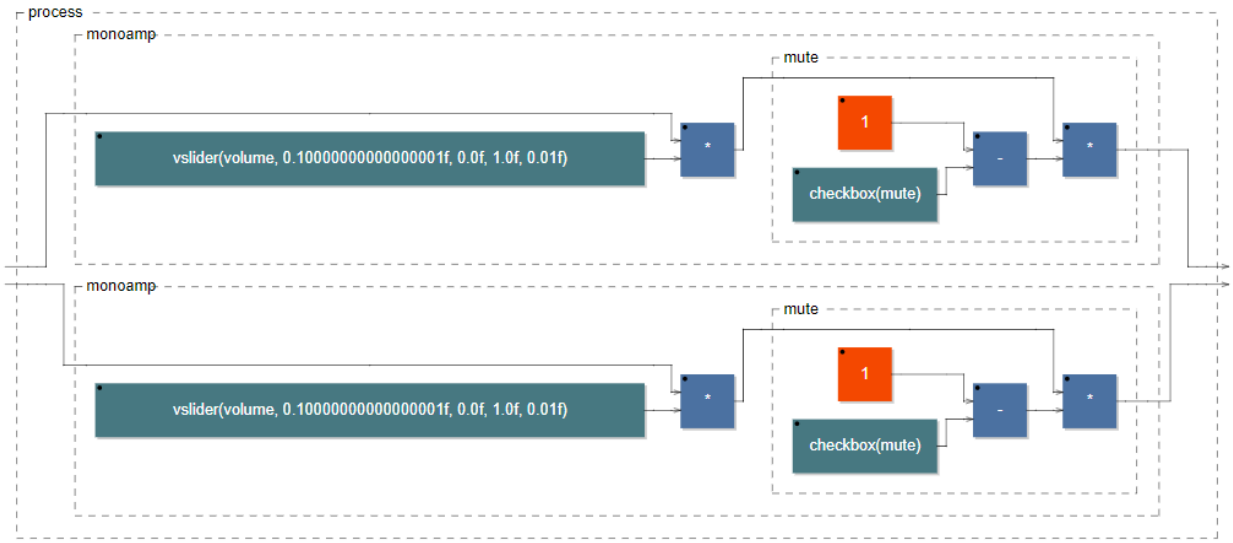
Estas notaciones se pueden combinar libremente. Por ejemplo, las siguientes expresiones son todas equivalentes:

Expresión	Descripción
`*(1-m)`	prefijo + notación de infijo
`_*(1-m)`	solo notación infija
`_,(1,m:-):*`	solo sintaxis central

Ejemplo 12: un botón de *silencio (mute)*

Ahora digamos que nos gustaría poder silenciar completamente el sonido con solo tocar un botón, sin tener que cambiar el volumen.

Agreguemos una etapa de silencio a nuestro amplificador mono. Para silenciar la señal solo tenemos que multiplicarla por 0. Utilizaremos para ese fin a `checkbox(...)`, un elemento de interfaz de usuario que produce una señal que es ***0 por defecto***, y ***1 cuando está marcado***. Como queremos multiplicar la señal por 0 cuando la casilla de verificación está marcada, usaremos `1-checkbox("mute")`. Al partir de 1 y restarle el valor del checkbox, es como si invirtiéramos el comportamiento del control.

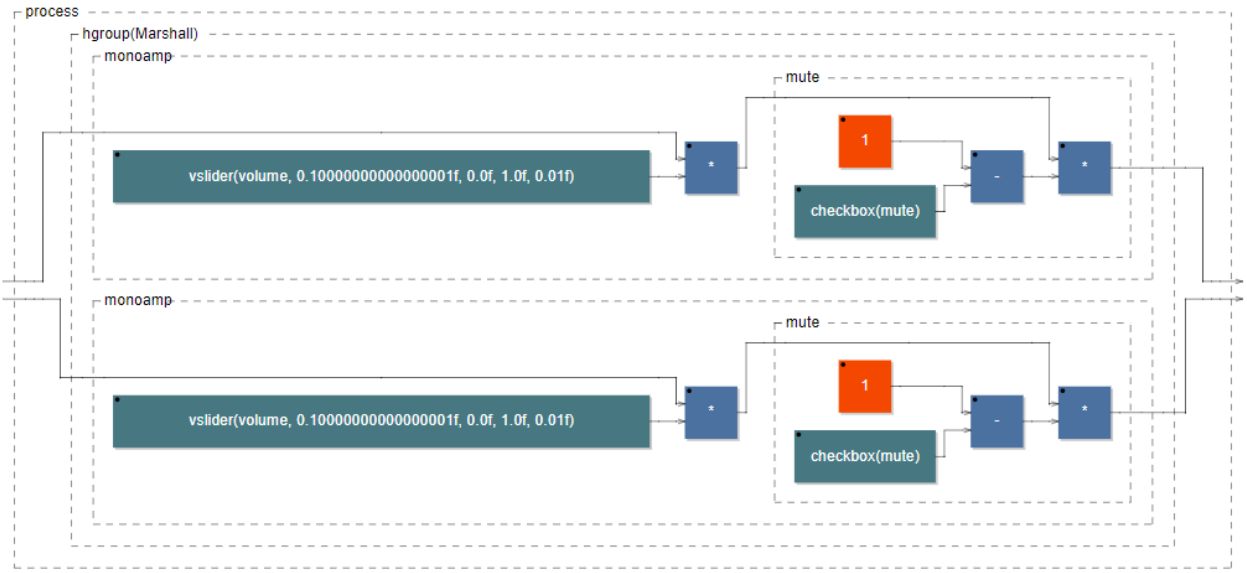


```
mute = *(1-checkbox("mute"));     // se puede experimentar con y sin el 1-

monoamp = *(vslider("volume[style:knob]", 0.1, 0, 1, 0.01)) : mute;

stereoamp = monoamp,monoamp;

process = stereoamp;
```

Pruébalo aquí >>

Ejemplo 13: diseño vertical y horizontal

Como se vio en el ejemplo anterior, por defecto, la disposición de los elementos gráficos es generalmente vertical. Podemos cambiar este diseño utilizando `hgroup(...)` y `vgroup(...)`. Por ejemplo, para que el diseño sea horizontal, se puede escribir:



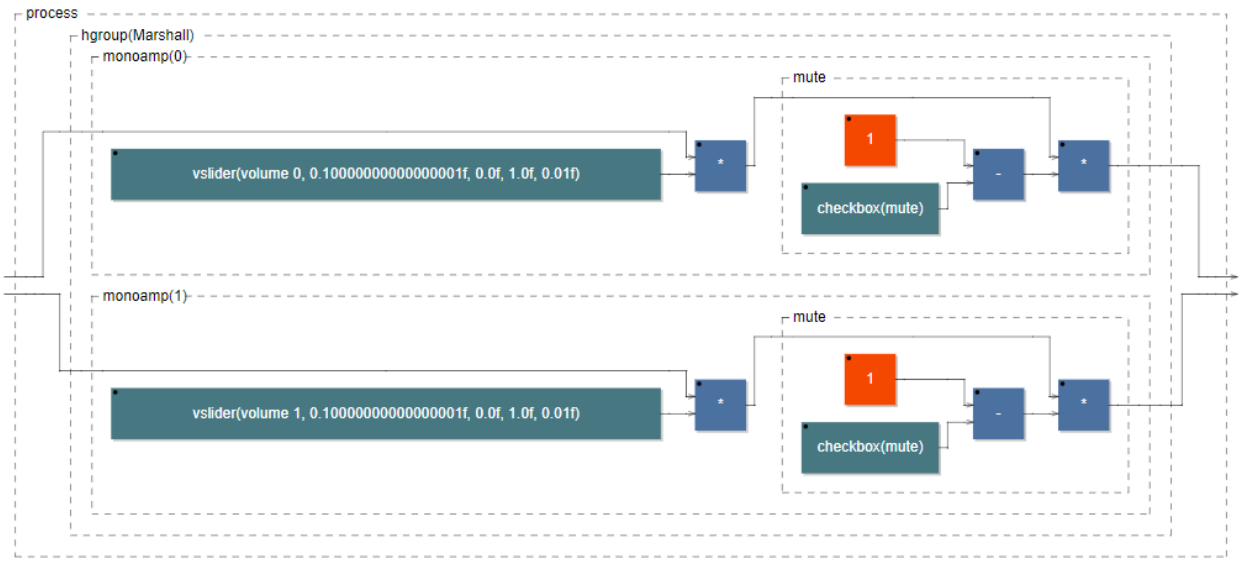

```
mute = *(1-checkbox("mute"));

monoamp = *(vslider("volume[style:knob]", 0.1, 0, 1, 0.01)) : mute;

stereoamp = hgroup("Marshall", monoamp,monoamp);

process = stereoamp;
```

Pruébalo aquí >>

Ejemplo 14: diferenciar el volumen de los dos canales

Para diferenciar el control de volumen de nuestros dos canales, ***parametrizaremos***, es decir, crearemos parámetros en monoamp. Específicamente crearemos un parámetro llamado `c` que representará el número de canal, y que se utilizará para diferenciar el nombre de cada control de volumen. Tener en cuenta que el nombre `c` del parámetro solo debe tener una letra para que se interprete correctamente en el nombre del control deslizante `"volume %c[style:knob]"`.

Este parámetro, lo crearemos indicando su nombre entre paréntesis tras el nombre de la definición que lo poseerá, en este caso `monoamp(c)`. Podemos incluirlo dentro del nombre de la perilla utilizando `%c`. Finalmente, cuando solicitemos utilizar dos monoamp dentro de nuestro stereoamp, le indicaremos este parámetro nuevamente entre paréntesis.




```
mute = *(1-checkbox("mute"));

monoamp(c) = *(vslider("volume %c[style:knob]", 0.1, 0, 1, 0.01)) : mute;

stereoamp = hgroup("Marshall", monoamp(0),monoamp(1));

process = stereoamp;
```

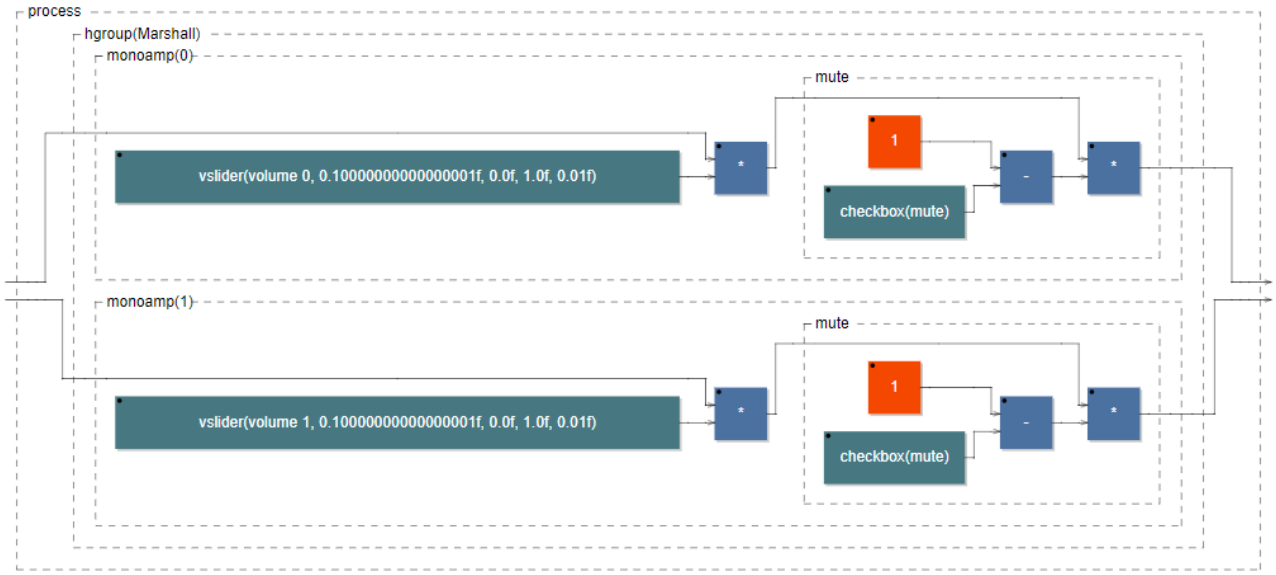
Pruébalo aquí >>

Ejemplo 15: tener muchos canales

Hemos construido un amplificador estéreo, pero supongamos que queremos generalizar esta construcción a un número mayor de canales. Para hacerlo, introduciremos la construcción paralela `par(i, N, ...)`, que nos permite poner varias veces la misma expresión en paralelo. En cierto modo, es el equivalente del bucle for () de un lenguaje de programación clásico.

En nuestro caso, queremos indicar, por ejemplo la cantidad de canales de nuestro amplificador.

La construcción paralela requiere tres argumentos, separados por comas: `par(i, N, elemento)` donde `i` será un valor que se ***incrementará automáticamente*** desde 0 hasta `N`. Este último, `N`, será entonces el ***número de veces*** que queramos que se repita nuestro `elemento`, que en principio será alguna ***definición*** que hayamos creado.




```
mute = *(1-checkbox("mute"));

monoamp(c) = *(vslider("volume %c[style:knob]", 0.1, 0, 1, 0.01)) : mute;

multiamp(N) = hgroup("Marshall", par(i, N, monoamp(i)));

process = multiamp(2); // probar multiamp(4) u otros valores (solo enteros positivos)
```

Pruébalo aquí >>

Hasta aquí hemos visto los elementos básicos de Faust para comenzar a experimentar con el lenguaje. En las siguientes partes de este tutorial veremos cómo trabajar con retardos, filtros, osciladores y otros elementos que nos permitirán crear proyectos más interesantes.