FAUST
Functional Programming for Signal Processing

Yann Orlarey, Dominique Fober,
Stéphane Letz

Grame, Centre National de Création Musicale
9 rue du Garet, BP 1185
69202 Lyon Cedex 01, France

Part 1

Overview of the FAUST project

The FAUST project
http://faudiostream.sourceforge.net

1. A formal specification language for real-time signal processing and sound synthesis

2. A compiler generating efficient C/C++ code comparable to hand-written code

3. Multiple implementations from one specification

Faust Language :

Block-diagrams + functional programming

1. Very powerful “glues” :
   - *higher order functions*
   - *lazy evaluation*
   - *partial application*
2. Simple and modular semantic
3. Adequate modeling of signals and signal processors

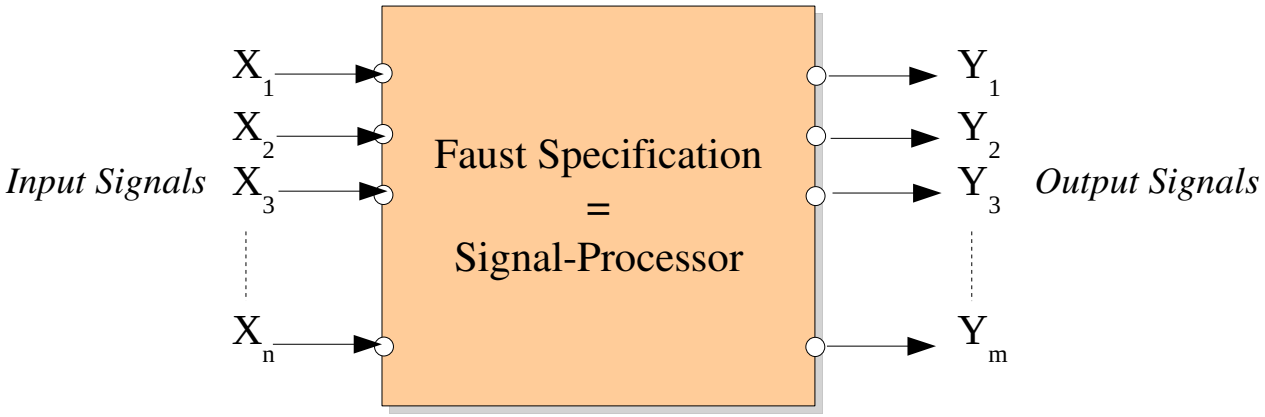
Functional Programming :

Adequate modeling of signals and signal processors

1. Audio signals are functions of time
2. Signal processors are functions of signals
3. Block-diagram operators are functions of signal processors

Faust Specification :

A set of definitions that defines a Signal Processor



Signal

$s:\mathbb{N}\rightarrow\mathbb{R}$

$\mathbb{S} = \mathbb{N}\rightarrow\mathbb{R}$

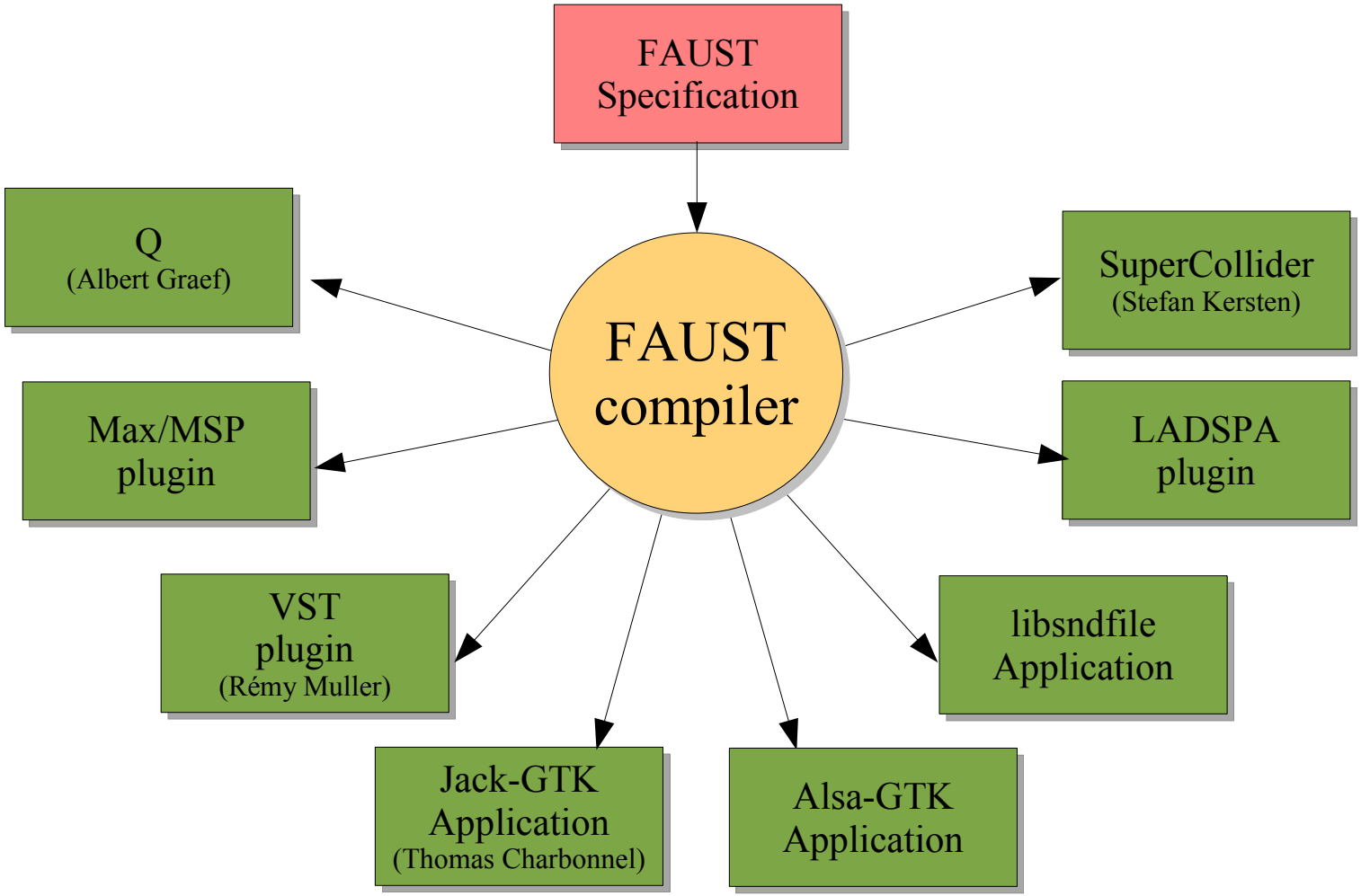
Signal Processor

$p : \mathbb{S}^n \rightarrow \mathbb{S}^m$

$\mathbb{P} = \bigcup_{n,m} \mathbb{S}^n \rightarrow \mathbb{S}^m$

The FAUST compiler

one specification ➜ multiple implementations



C++ Code Generation

```
process = *( hslider("volume", 0.5, 0, 1, 0.01) );
```

```
class mydsp : public dsp {
  private:
    float   fslider0;
  public:
    virtual int getNumInputs()    { return 1; }
    virtual int getNumOutputs()   { return 1; }
    virtual void init(int samplingRate) {
            fslider0 = 0.5;
    }
    virtual void buildUserInterface(UI* interface) {
            interface->openVerticalBox("");
            interface->addHorizontalSlider("volume", &fslider0,
                                           0.5, 0.0, 1.0, 0.01);
            interface->closeBox();
    }
    virtual void compute(int count, float** input, float** output) {
            float* input0 = input[0];
            float* output0 = output[0];
            float ftemp0 = fslider0;
            for (int i=0; i<count; i++) {
                output0[i] = (input0[i] * ftemp0);
            }
    }
};
```

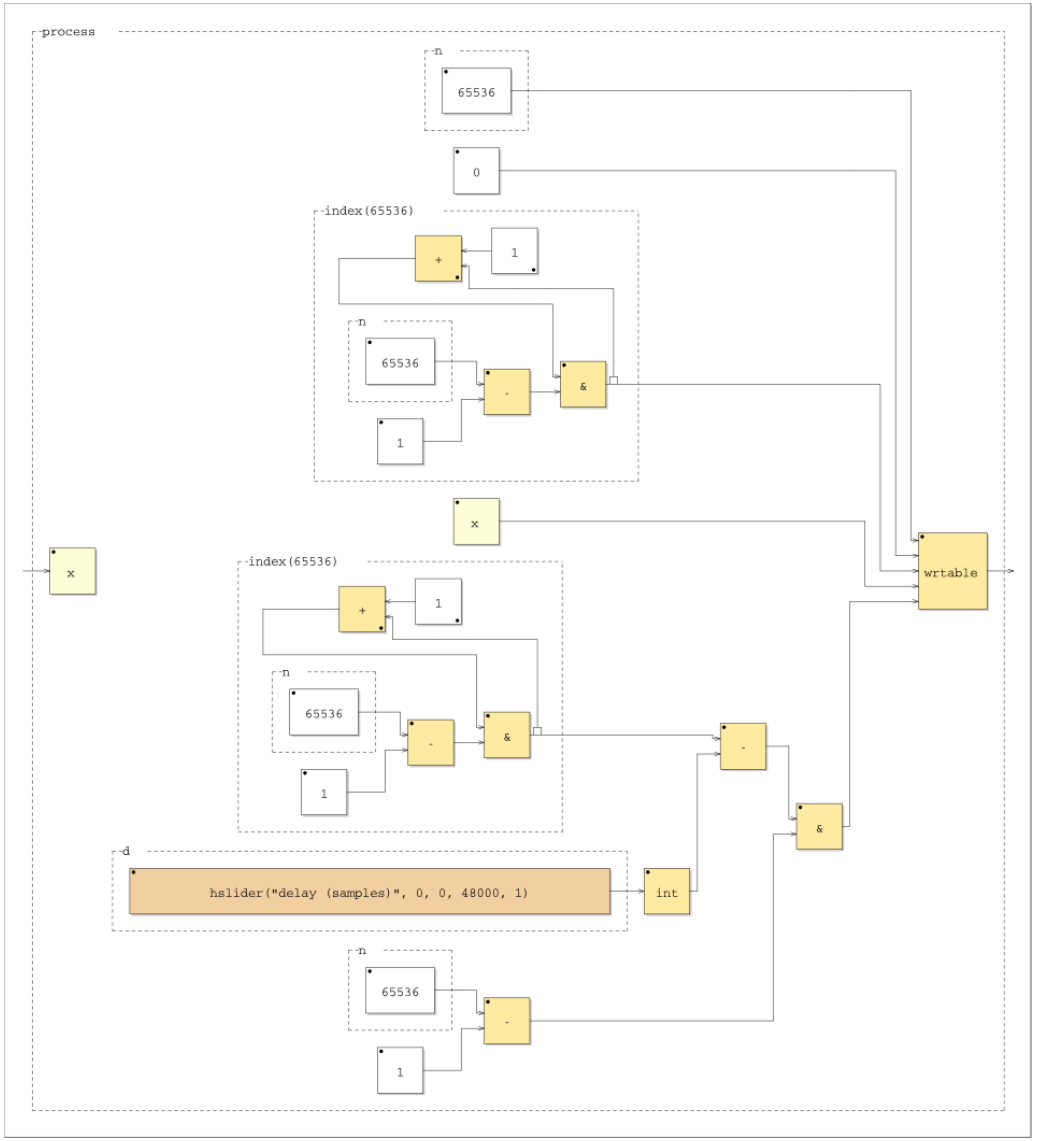
UI abstract class

```
class UI
{
 public:
    virtual ~UI() {}

    virtual void addButton(char* label, float* zone) = 0;
    virtual void addToggleButton(char* label, float* zone) = 0;
    virtual void addCheckButton(char* label, float* zone) = 0;

    virtual void addVerticalSlider(char* label, float* zone,
                float init, float min, float max, float step) = 0;
    virtual void addHorizontalSlider(char* label, float* zone,
                float init, float min, float max, float step) = 0;
    virtual void addNumEntry(char* label, float* zone,
                float init, float min, float max, float step) = 0;

    virtual void openVerticalBox(char* label) = 0;
    virtual void openHorizontalBox(char* label) = 0;
    virtual void openTabBox(char* label) = 0;
    virtual void closeBox() = 0;

};
```

Part 2

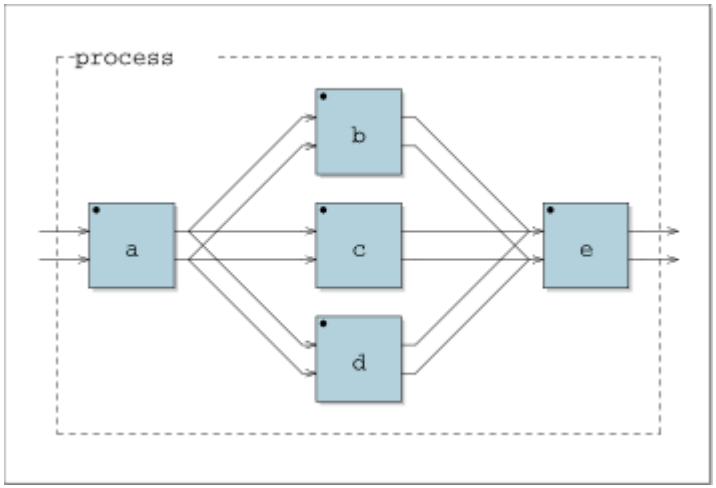
The Block-Diagram Algebra

Graphic block-diagram



Example

Graph representation



Algebraic representation

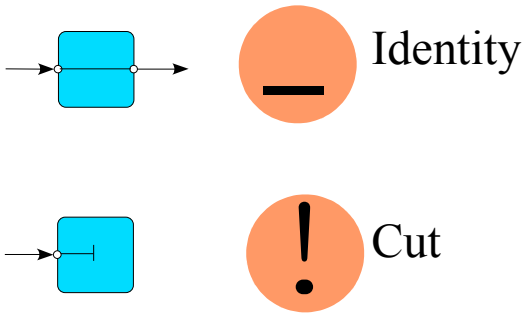
```
a <: b,c,d :> e
```

Advantages of the BDA

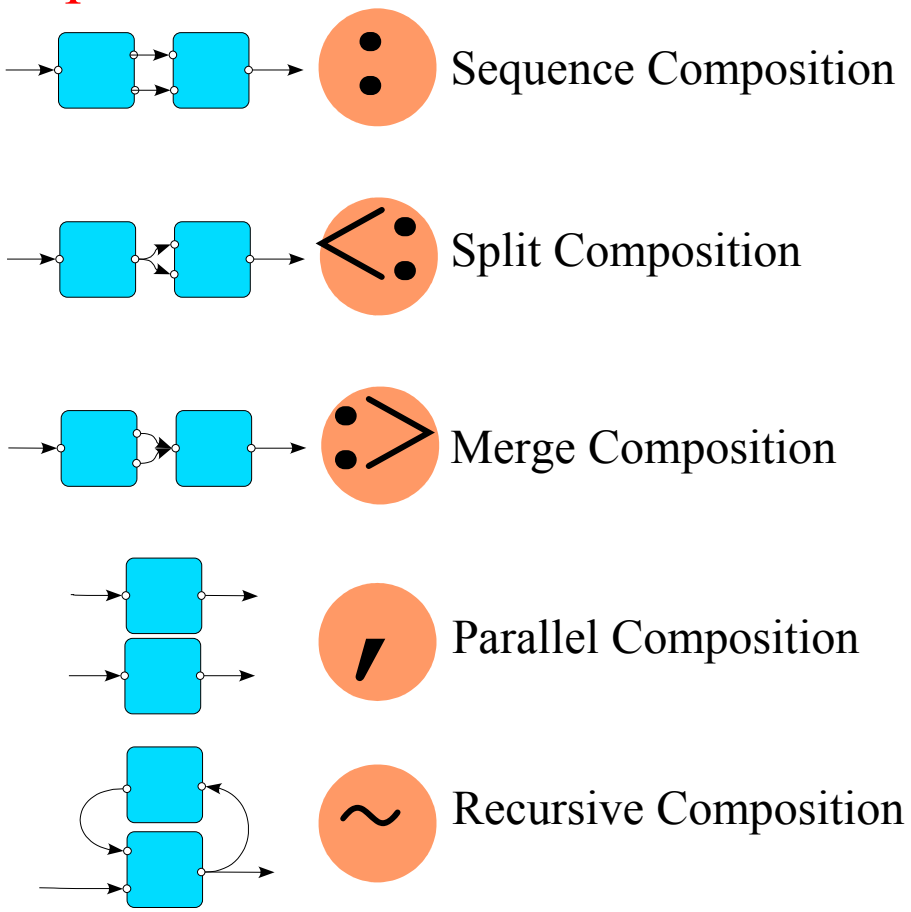
1. Powerful enough to represent any block diagram
2. Concise and adequate textual syntax for block diagram languages
3. Useful to formalize the semantic of block diagram languages
4. Suitable for formal manipulations : lambda-calculus, partial evaluation, compilation...

Overview of the BDA

Constants

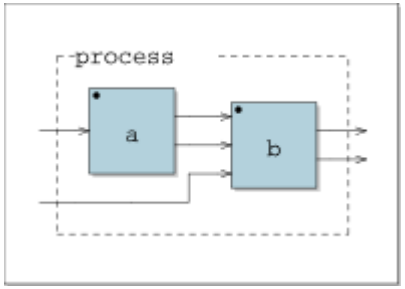


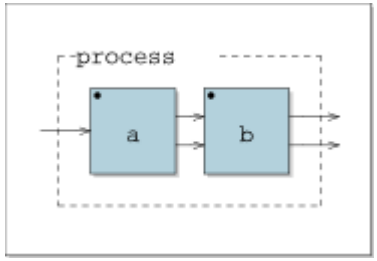
Operators

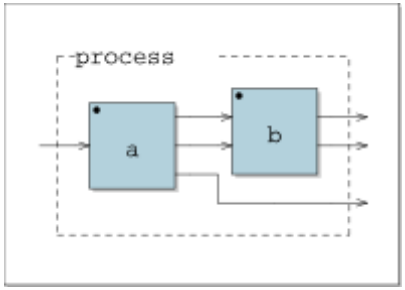


Sequence Composition :

```
a : b
```







Outs(a) < Ins(b)

Ins(a:b) = Ins(a)+Ins(b)-Outs(a)

Outs(a:b) = Outs(b)

Outs(a) = Ins(b)

Ins(a:b) = Ins(a)

Outs(a:b) = Outs(b)

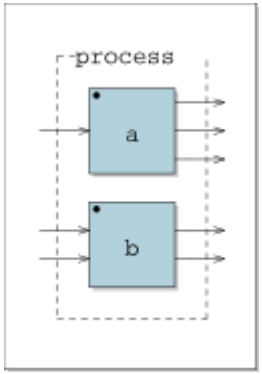
Outs(a) > Ins(b)

Ins(a:b) = Ins(a)

Outs(a:b) = Outs(b)+Outs(a)-Ins(b)

Parallel Composition ,

```
a , b
```

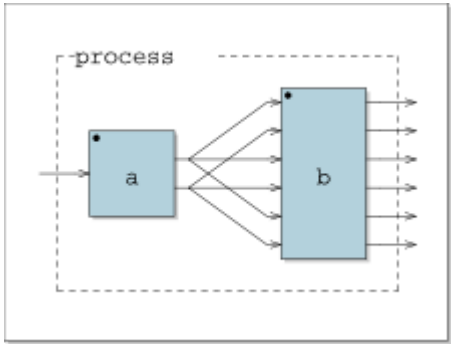


Ins(a,b) = Ins(a)+Ins(b)

Outs(a,b) = Outs(a)+Outs(b)

Split Composition

```
a <: b
```



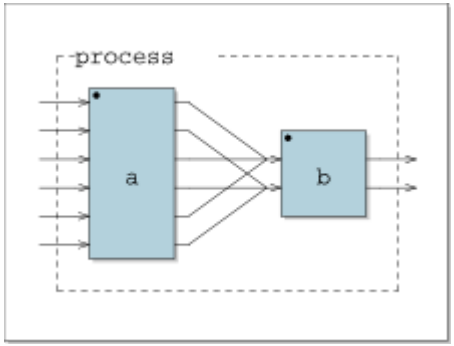
Outs(a)*k = Ins(b)

Ins(a<:b) = Ins(a)

Outs(a<:b) = Outs(b)

Merge Composition 

```
a :> b
```



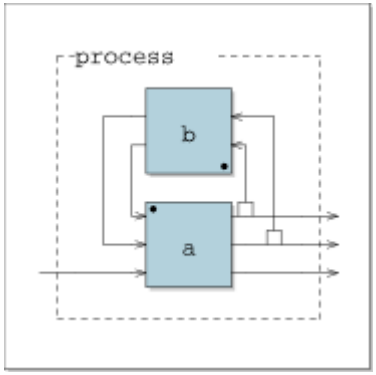
Outs(a) = k*Ins(b)

Ins(a:>b) = Ins(a)

Outs(a:>b) = Outs(b)

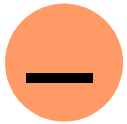
Recursive Composition ~

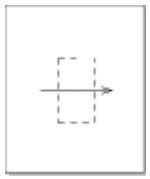
```
a ~ b
```



Outs(a)>=Ins(b) &Ins(a)>=Outs(b)

Ins(a~b) = Ins(a)-Outs(b)
Outs(a~b) = Outs(a)

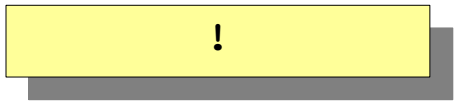
Identity 

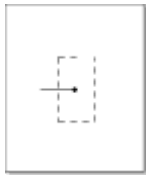


Ins(_) = 1

Outs(_) = 1

Cut !





Ins(!) = 1

Outs(!) = 0

Part 3

The Building Blocks

Operations on signals

Syntax	Type	Description
n	$\mathbb{S}^0 \rightarrow \mathbb{S}^1$	integer number: $y(t) = n$
$n.m$	$\mathbb{S}^0 \rightarrow \mathbb{S}^1$	floating point number: $y(t) = n.m$
_	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	identity function: $y(t) = x(t)$
!	$\mathbb{S}^1 \rightarrow \mathbb{S}^0$	cut function: $\forall x \in \mathbb{S}, (x) \rightarrow ()$
int	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	cast into an int signal: $y(t) = (int)x(t)$
float	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	cast into an float signal: $y(t) = (float)x(t)$
+	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	addition: $y(t) = x_1(t) + x_2(t)$
-	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	substraction: $y(t) = x_1(t) - x_2(t)$
*	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	multiplication: $y(t) = x_1(t) * x_2(t)$
/	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	division: $y(t) = x_1(t)/x_2(t)$
%	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	modulo: $y(t) = x_1(t)\%x_2(t)$
&	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	logical AND: $y(t) = x_1(t)\&x_2(t)$
|	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	logical OR: $y(t) = x_1(t)\|x_2(t)$
$\wedge$	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	logical XOR: $y(t) = x_1(t) \wedge x_2(t)$
<<	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	arith. shift left: $y(t) = x_1(t) << x_2(t)$
>>	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	arith. shift right: $y(t) = x_1(t) >> x_2(t)$
<	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	less than: $y(t) = x_1(t) < x_2(t)$
<=	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	less or equal: $y(t) = x_1(t) <= x_2(t)$
>	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	greater than: $y(t) = x_1(t) > x_2(t)$
>=	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	greater or equal: $y(t) = x_1(t) >= x_2(t)$
==	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	equal: $y(t) = x_1(t) == x_2(t)$
!=	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	different: $y(t) = x_1(t)! = x_2(t)$

Mathematical functions on signals

Syntax	Type	Description
acos	$\mathbb{S}^1 \to \mathbb{S}^1$	arc cosine: $y(t) = \text{acosf}(x(t))$
asin	$\mathbb{S}^1 \to \mathbb{S}^1$	arc sine: $y(t) = \text{asinf}(x(t))$
atan	$\mathbb{S}^1 \to \mathbb{S}^1$	arc tangent: $y(t) = \text{atanf}(x(t))$
atan2	$\mathbb{S}^2 \to \mathbb{S}^1$	arc tangent of 2 signals: $y(t) = \text{atan2f}(x_1(t), x_2(t))$
cos	$\mathbb{S}^1 \to \mathbb{S}^1$	cosine: $y(t) = \text{cosf}(x(t))$
sin	$\mathbb{S}^1 \to \mathbb{S}^1$	sine: $y(t) = \text{sinf}(x(t))$
tan	$\mathbb{S}^1 \to \mathbb{S}^1$	tangent: $y(t) = \text{tanf}(x(t))$
exp	$\mathbb{S}^1 \to \mathbb{S}^1$	base-e exponential: $y(t) = \text{expf}(x(t))$
log	$\mathbb{S}^1 \to \mathbb{S}^1$	base-e logarithm: $y(t) = \text{logf}(x(t))$
log10	$\mathbb{S}^1 \to \mathbb{S}^1$	base-10 logarithm: $y(t) = \text{log10f}(x(t))$
pow	$\mathbb{S}^2 \to \mathbb{S}^1$	power: $y(t) = \text{powf}(x_1(t), x_2(t))$
sqrt	$\mathbb{S}^1 \to \mathbb{S}^1$	square root: $y(t) = \text{sqrtf}(x(t))$
abs	$\mathbb{S}^1 \to \mathbb{S}^1$	absolute value (int): $y(t) = \text{abs}(x(t))$
		absolute value (float): $y(t) = \text{fabsf}(x(t))$
min	$\mathbb{S}^2 \to \mathbb{S}^1$	minimum: $y(t) = \min(x_1(t), x_2(t))$
max	$\mathbb{S}^2 \to \mathbb{S}^1$	maximum: $y(t) = \max(x_1(t), x_2(t))$
fmod	$\mathbb{S}^2 \to \mathbb{S}^1$	float modulo: $y(t) = \text{fmodf}(x_1(t), x_2(t))$
remainder	$\mathbb{S}^2 \to \mathbb{S}^1$	float remainder: $y(t) = \text{remainderf}(x_1(t), x_2(t))$
floor	$\mathbb{S}^1 \to \mathbb{S}^1$	largest int $\leq$: $y(t) = \text{floorf}(x(t))$
ceil	$\mathbb{S}^1 \to \mathbb{S}^1$	smallest int $\geq$: $y(t) = \text{ceilf}(x(t))$
rint	$\mathbb{S}^1 \to \mathbb{S}^1$	closest int: $y(t) = \text{rintf}(x(t))$

Delays and Tables

Syntax	Type	Description
`mem`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	1-sample delay: $y(t+1) = x(t), y(0) = 0$
`prefix`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	1-sample delay: $y(t+1) = x_2(t), y(0) = x_1(0)$
`@`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	fixed delay: $y(t + x_2(t)) = x_1(t), y(t < x_2(t)) = 0$
`rdtable`	$\mathbb{S}^3 \rightarrow \mathbb{S}^1$	read-only table: $y(t) = T[r(t)]$
`rwtable`	$\mathbb{S}^5 \rightarrow \mathbb{S}^1$	read-write table: $T[w(t)] = c(t); y(t) = T[r(t)]$
`select2`	$\mathbb{S}^3 \rightarrow \mathbb{S}^1$	select between 2 signals: $T[] = \{x_0(t), x_1(t)\}; y(t) = T[s(t)]$
`select3`	$\mathbb{S}^4 \rightarrow \mathbb{S}^1$	select between 3 signals: $T[] = \{x_0(t), x_1(t), x_2(t)\}; y(t) = T[s(t)]$

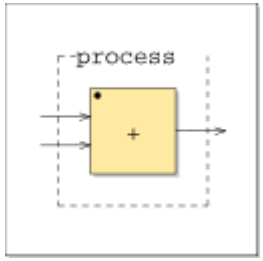
Graphic User Interface

Syntax	Example
`button(`*str*`)`	`button("play")`
`checkbox(`*str*`)`	`checkbox("mute")`
`vslider(`*str*`,`*cur*`,`*min*`,`*max*`,`*step*`)`	`vslider("vol",50,0,100,1)`
`hslider(`*str*`,`*cur*`,`*min*`,`*max*`,`*step*`)`	`hslider("vol",0.5,0,1,0.01)`
`nentry(`*str*`,`*cur*`,`*min*`,`*max*`,`*step*`)`	`nentry("freq",440,0,8000,1)`
`vgroup(`*str*`,`*block-diagram*`)`	`vgroup("reverb", ...)`
`hgroup(`*str*`,`*block-diagram*`)`	`hgroup("mixer", ...)`
`tgroup(`*str*`,`*block-diagram*`)`	`vgroup("parametric", ...)`
`vbargraph(`*str*`,`*min*`,`*max*`)`	`vbargraph("input",0,100)`
`hbargraph(`*str*`,`*min*`,`*max*`)`	`hbargraph("signal",0,1.0)`

Part 4

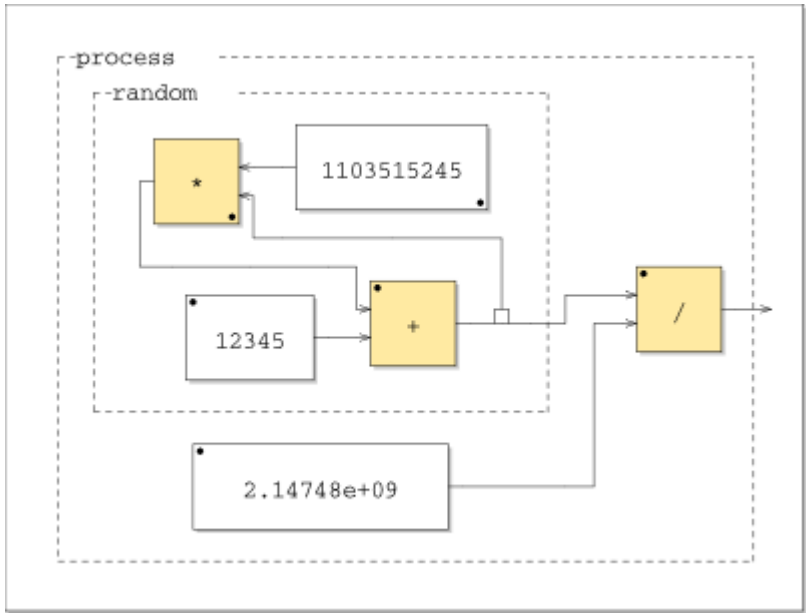
Examples

A Very Simple Example



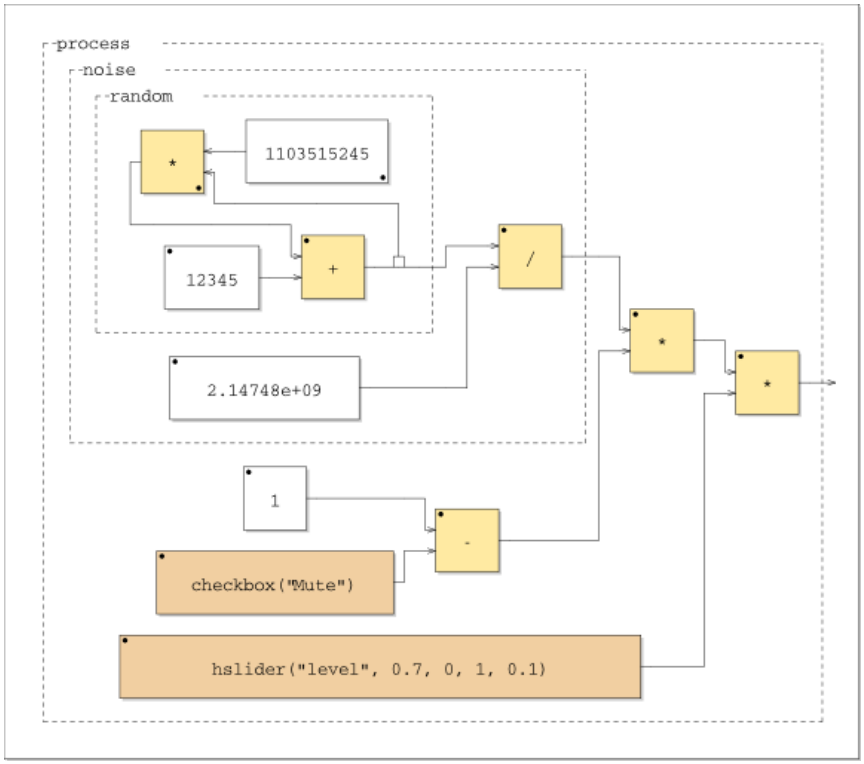

```
//-----------------------------------------------------
//
//   Stereo to mono conversion : a very simple
//   example of Faust program
//
//-----------------------------------------------------

process   = +;
```

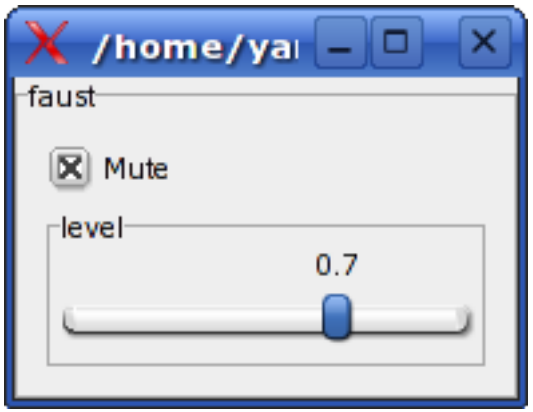
A Simple Noise Generator




```
//===============================================
//   Noise generator
//===============================================

random    = +(12345) ~ *(1103515245);
process   = random/2147483647.0;
```

A Noise Generator with GUI

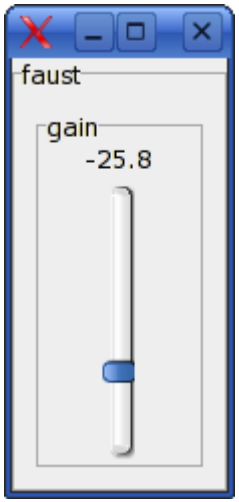


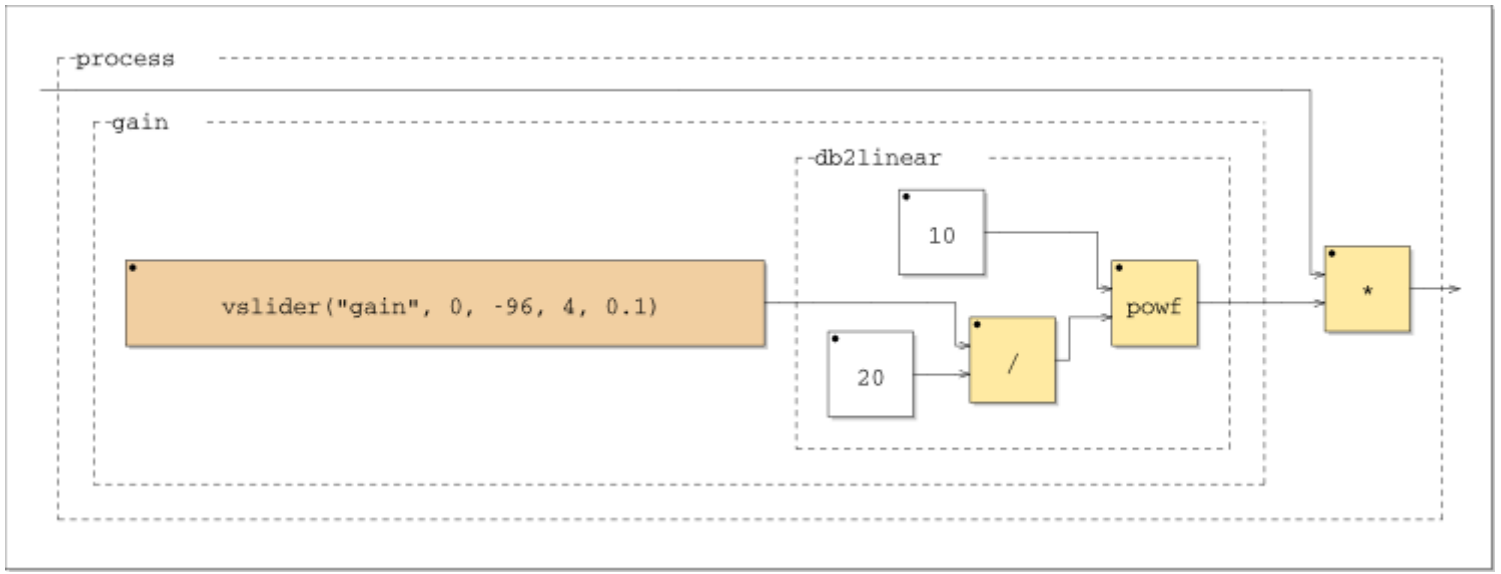



```
//=================================================
//   Noise generator
//=================================================

random    = +(12345) ~ *(1103515245);
noise     = random/2147483647.0;
process   = noise * (1-checkbox("Mute"))
                  * hslider("level", 0.7, 0, 1, 0.1);
```

Volume Control

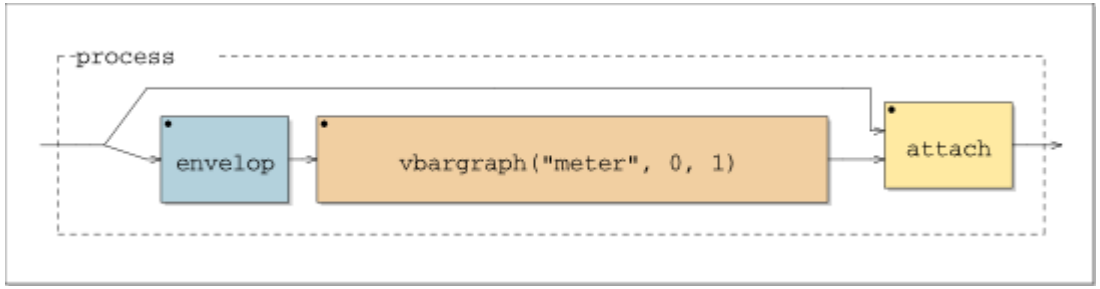
```
//-----------------------------------------------
//              Volume control in dB
//-----------------------------------------------


db2linear = pow(10, /(20.0));

gain      = vslider("gain", 0, -96, 4, 0.1) : db2linear;

process   = *(gain);
```

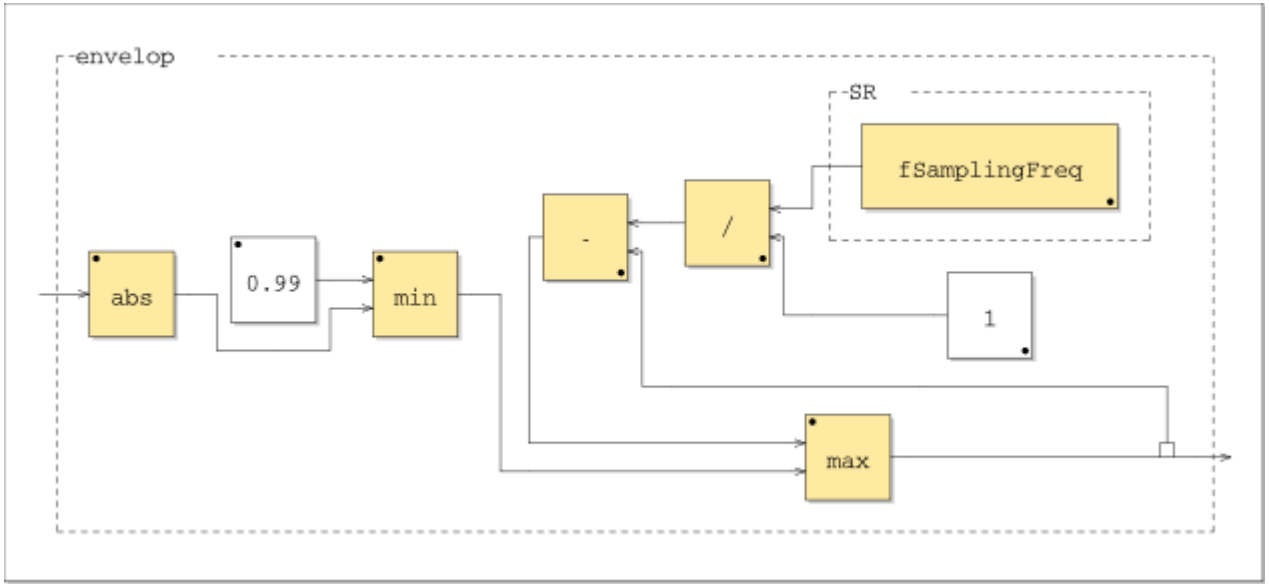




```
//-----------------------------------------------
// Simple vumeter
//-----------------------------------------------
import("music.lib");

envelop   = abs : min(0.99) : max ~ -(1.0/SR);
vumeter   = _ <: attach(_, envelop : vbargraph("meter", 0, 1));

process   = vumeter;
```

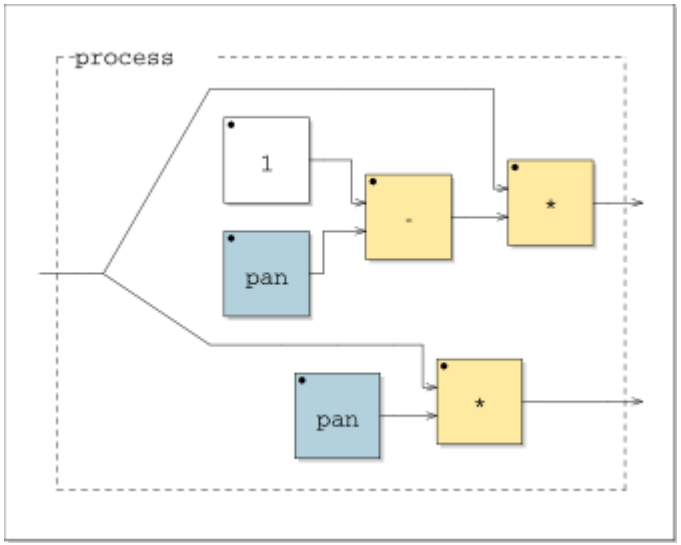
Vumeter

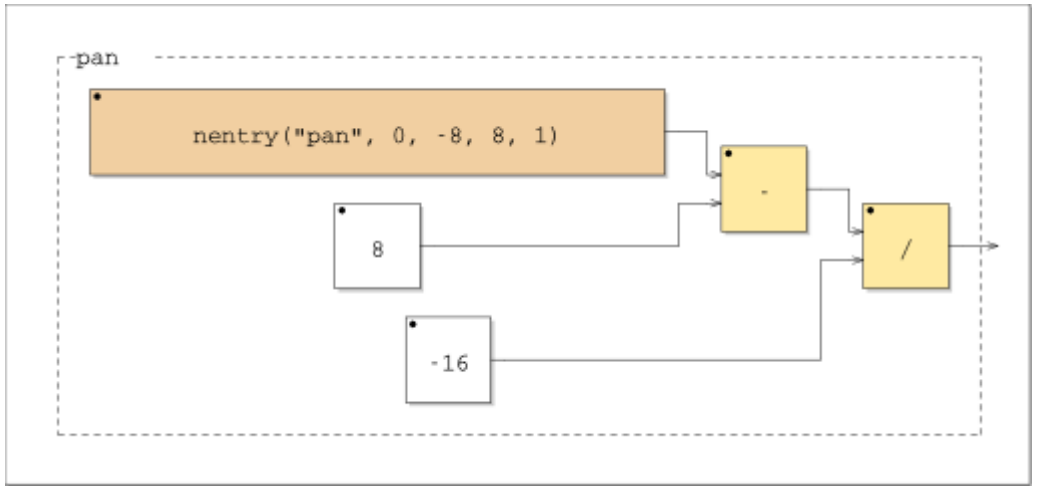




Stereo Pan Control

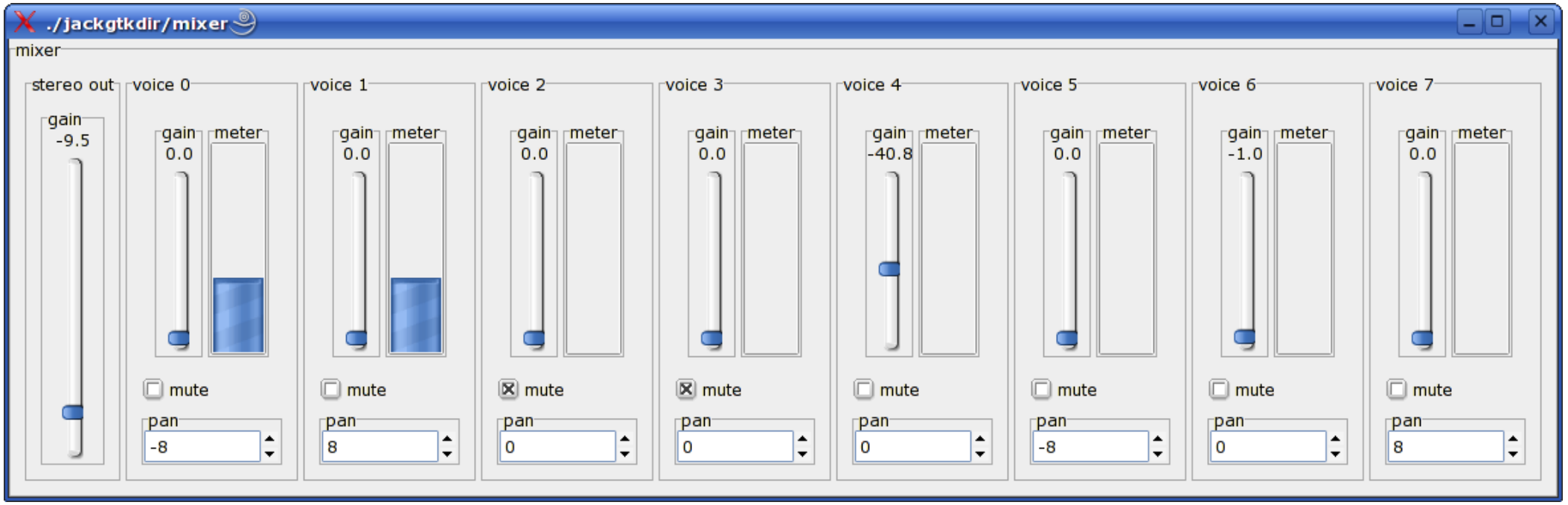
```
//-----------------------------------------------
// Stereo panpot
//-----------------------------------------------

panpot    = _ <:*(1-pan), *(pan)
     with {
          pan  =(nentry("pan",0,-8,8,1)-8)/-16;
     };

process   = panpot;
```





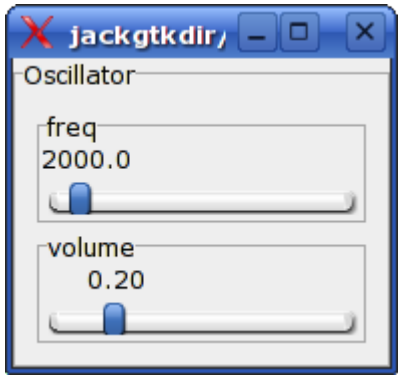
Mixer

```
vol       = component("volume.dsp");
pan       = component("panpot.dsp");
vumeter   = component("vumeter.dsp");
mute      = *(1 - checkbox("mute"));

voice(v)  = vgroup("voice %v",  mute : hgroup("", vol : vumeter) : pan);
stereo    = hgroup("stereo out", vol, vol);

process   = hgroup("mixer", par(i, 8, voice(i)) :> stereo);
```

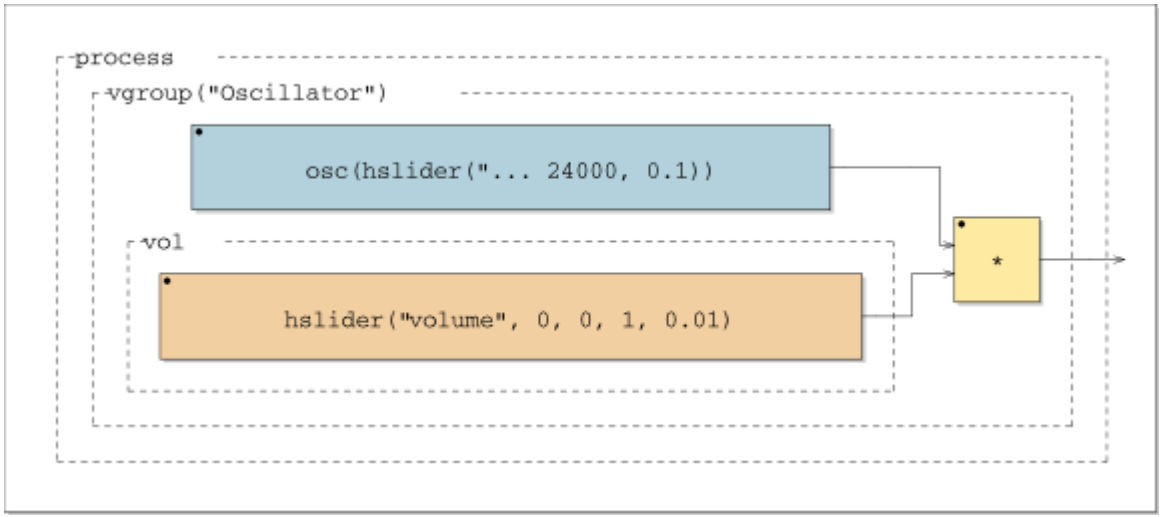


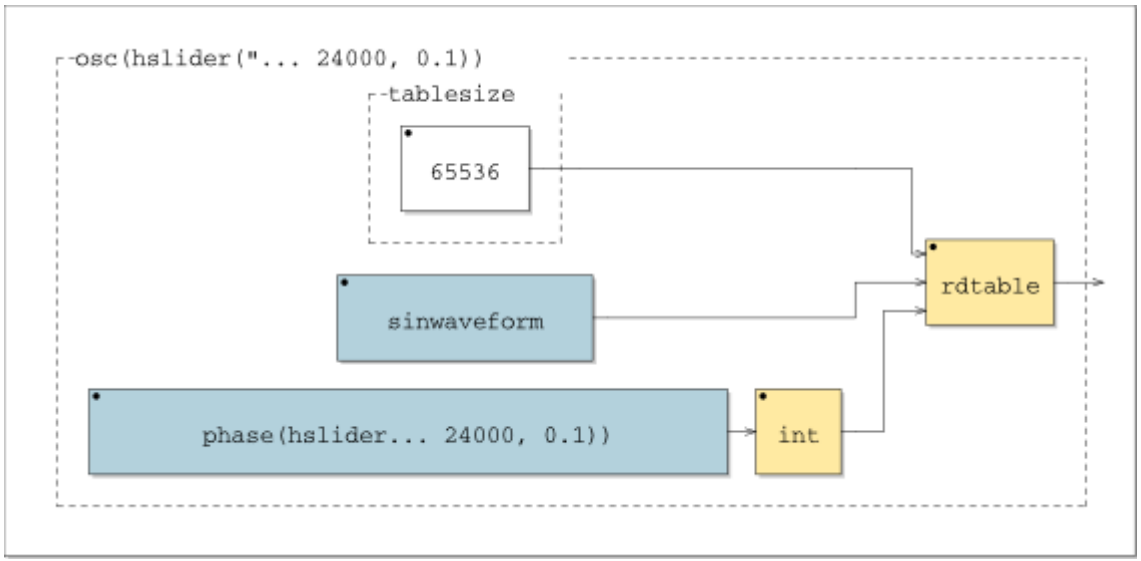
Oscillator (1/3)

```
//-----------------------------------------------
//             Sinusoidal Oscillator
//-----------------------------------------------
import("math.lib");

//--------------------- waveform ------------------------
tablesize      = 65536;
samplingfreq   = fconstant(int fSamplingFreq, <math.h>);
time           = (+(1)~_ ) - 1;              // 0,1,2,3,...
sinwaveform    = float(time)*(2.0*PI)/float(tablesize) : sin;

//-------------------- oscillator ------------------------
decimal(x)     = x - floor(x);
phase(freq)    = freq/float(samplingfreq)
               : (+ : decimal) ~ _
               : *(float(tablesize));
osc(freq)      = rdtable(tablesize,
                         sinwaveform, int(phase(freq)) );

//--------------------- process ------------------------
vol            = hslider("volume", 0, 0, 1, 0.01);
freq           = hslider("freq", 1000, 0, 24000, 0.1);

process        = vgroup("Oscillator", osc(freq) * vol);
```

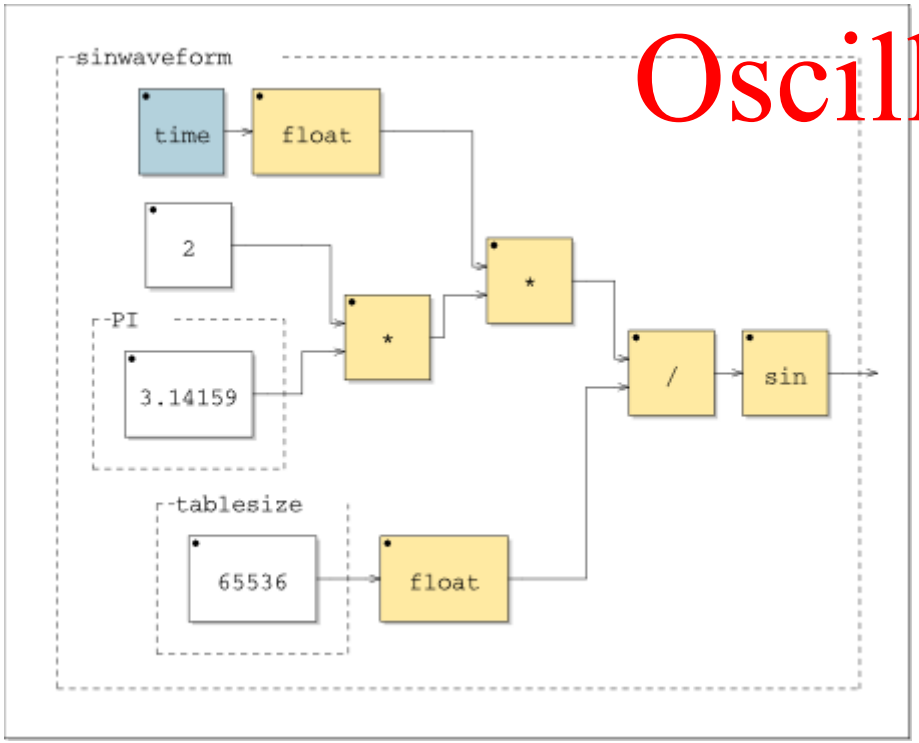


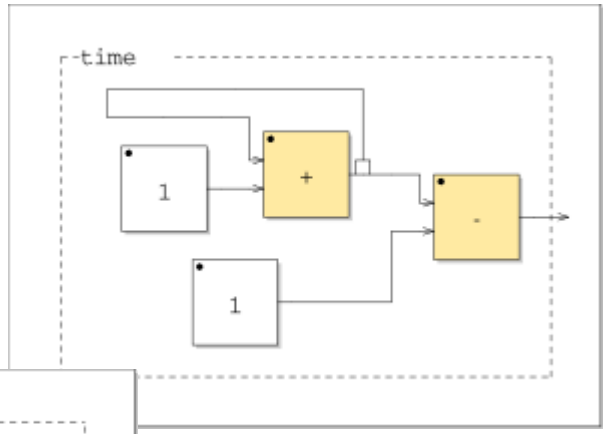
Oscillator (2/3)

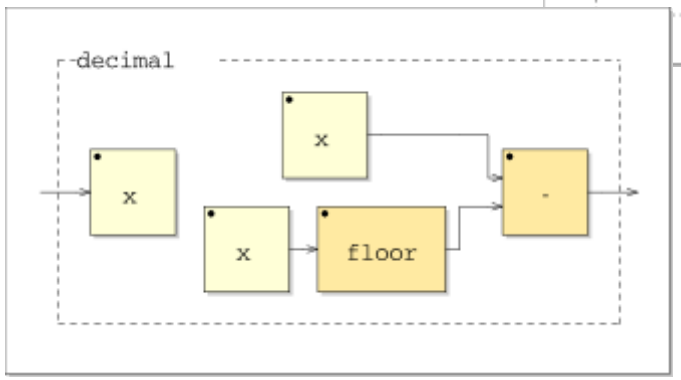


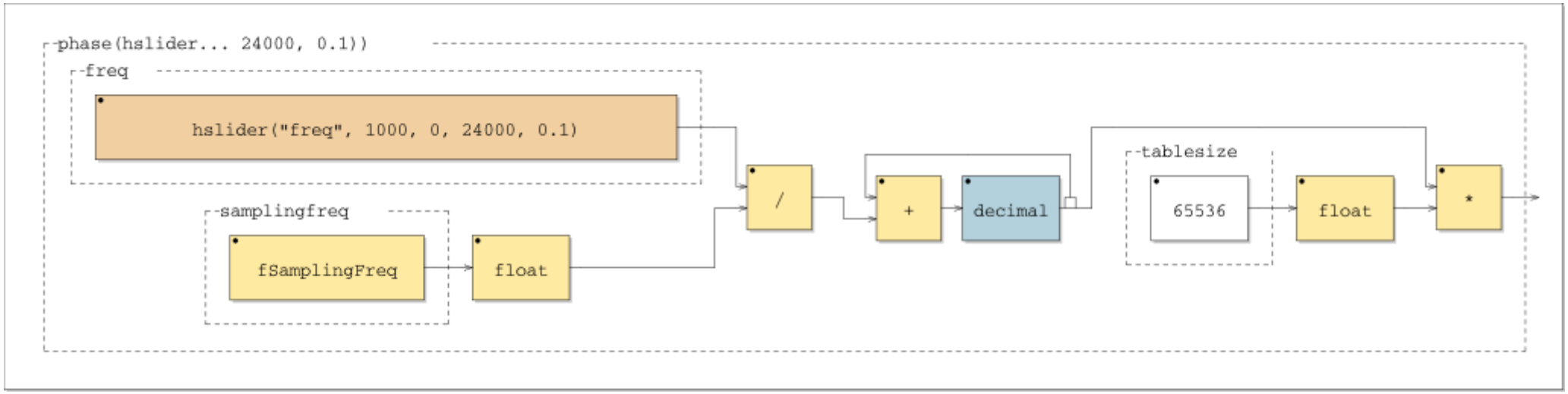


Oscillator (3/3)

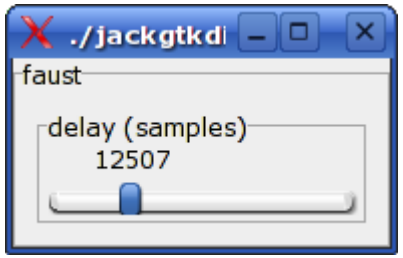






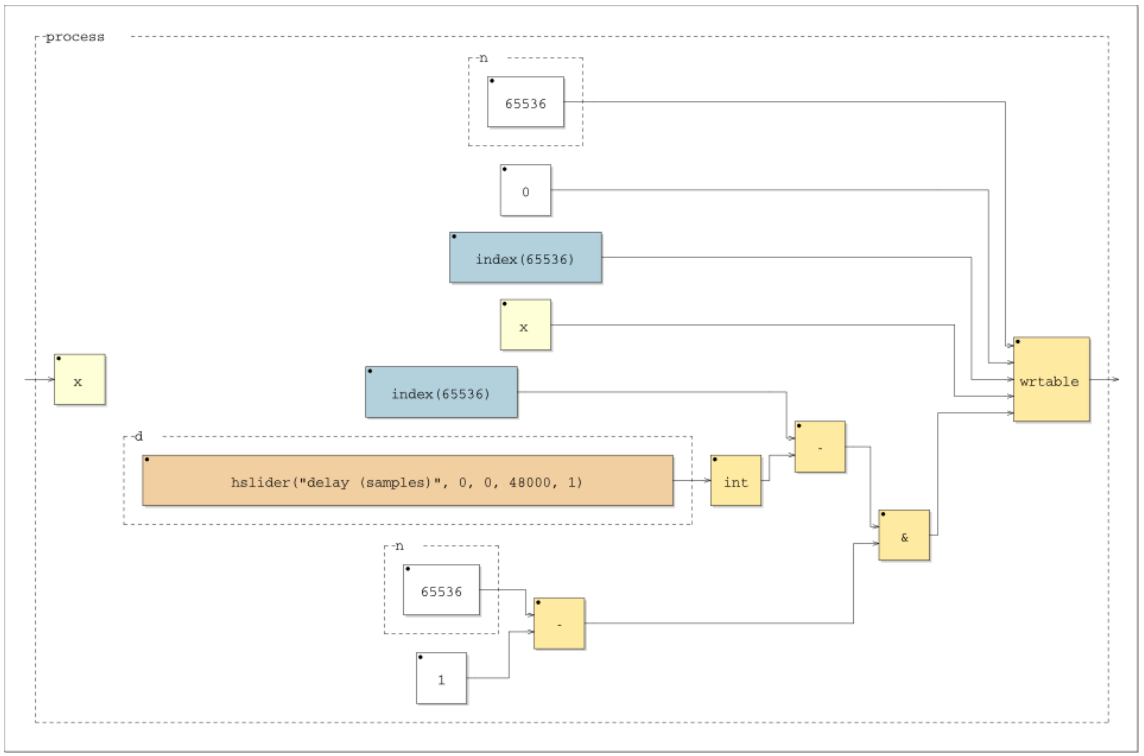


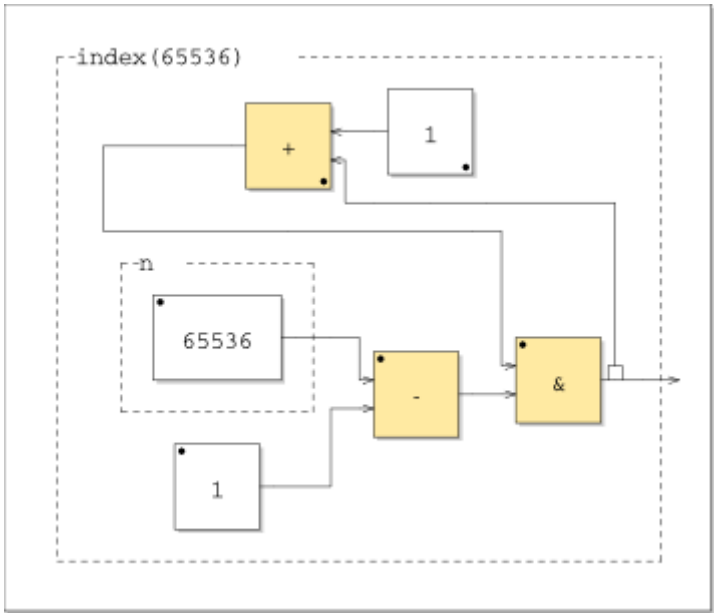
Variable Delay (1/2)




```
//-----------------------------------------------------------------------
//        Simple Delay Line
//-----------------------------------------------------------------------

index(n)        = &(n-1) ~ +(1);          // n = 2**i
delay(n,d,x)    = rwtable(n, 0.0, index(n), x, (index(n)-int(d)) & (n-1));
process         = delay(65536, hslider("delay (samples)", 0, 0, 48000, 1));
```

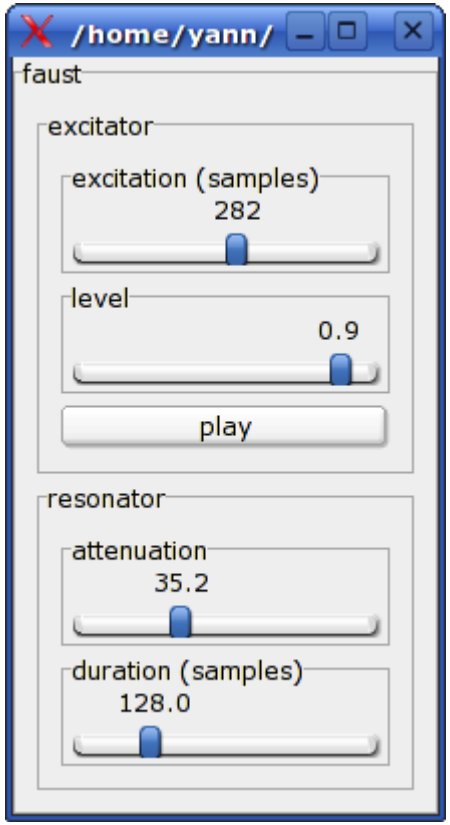
Variable Delay (2/2)



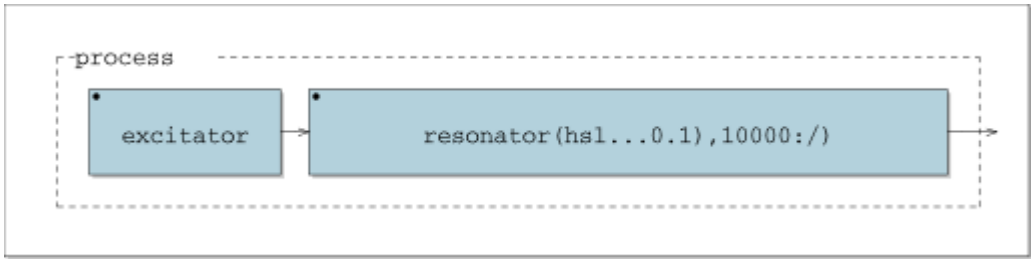


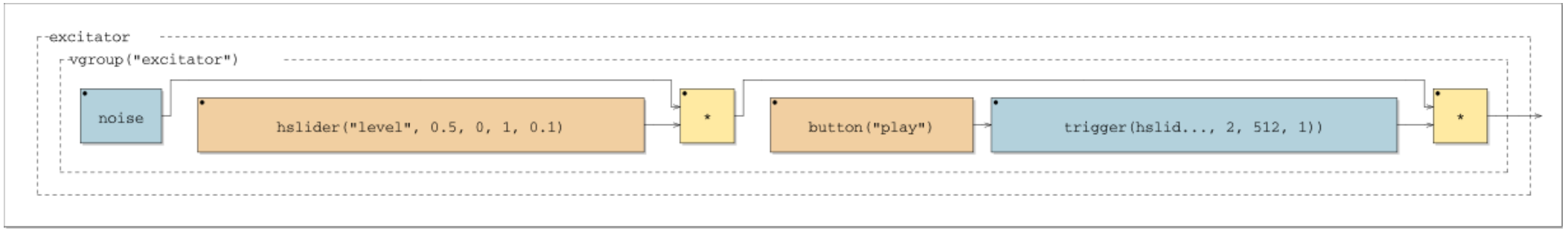
Karplus-Strong String (1/3)

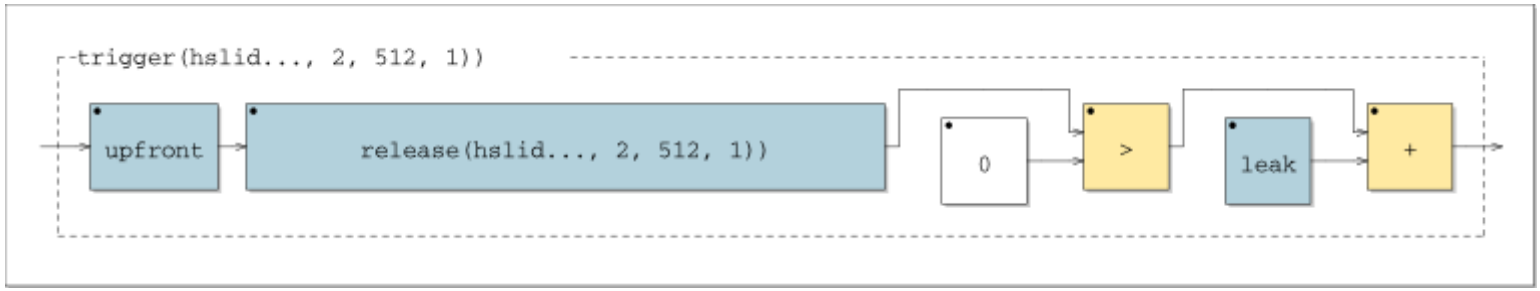
```
import("music.lib");
import("math.lib");

//----------------Excitator-------------------
upfront(x)      = (x-x') > 0.0;
decay(n,x)      = x - (x>0.0)/n;
release(n)      = + ~ decay(n);
trigger(n)      = upfront : release(n) : >(0.0) : +(leak);
leak            = 1.0/655360.0;
excitator       = vgroup("excitator",
                     noise
                     : *(hslider("level", 0.5, 0, 1, 0.1))
                     : *(button("play")
                     : trigger(hslider("excitation (samples)",
                                  128, 2, 512, 1)))
                );

//----------------resonator--------------------
average(x)      = (x+x')/2;

resonator(d,a) = vgroup("resonator",
                    (+ : fdelay1s(d-1.5)) ~ (average : *(1.0-a))
                  ) ;

//-------------process-----------------------
dur       = hslider("duration (samples)", 128, 2, 512, 0.1);
att       = hslider("attenuation", 10, 0, 100, 0.1)/10000;

process = excitator : resonator(dur,att);
```



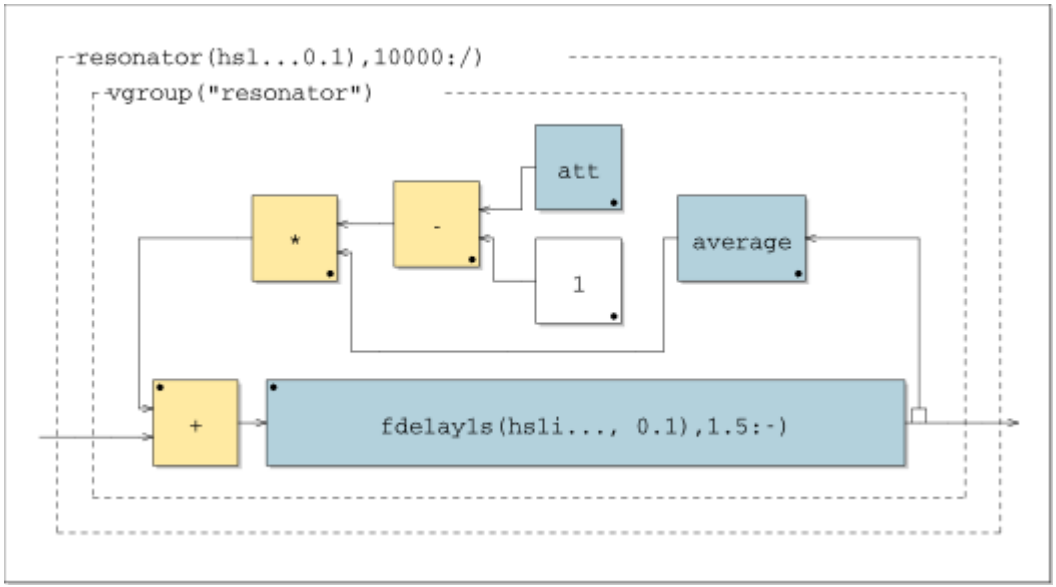
Karplus-Strong String (2/3)







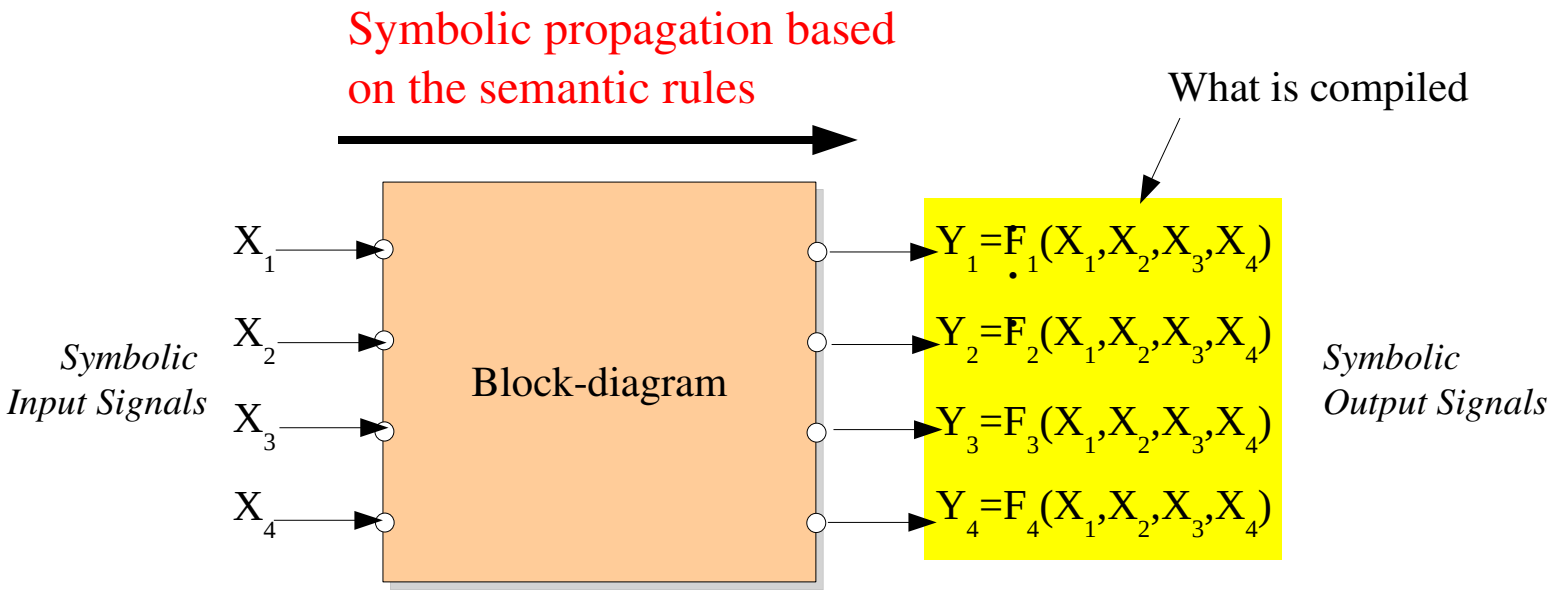
Karplus-Strong String (3/3)



Part 4

Compilation Process

Semantic Based compilation



Compilation Phases

1. Definitions Parsing
2. Evaluation of « process »
3. Symbolic Propagation
4. Symbolic Simplification and Normalization
5. Maximal Sharing of CSE
6. Type Based Optimizations
7. (SIMD) Code Generation
8. Add Target Architecture Code

Type System

1. Nature of the signal :
   integer or ***real***.
2. Computability of the signal :
   compile time, ***init time***, ***real time***
3. Variability of the signal :
   constant, ***control rate***, ***audio rate***
4. ***Scalar*** or ***SIMD*** computation

Part 5

Conclusion

Key elements of the Faust

1. Functional Programming
2. Block Diagram Composition
3. Well defined formal semantic
4. Efficient Semantic Compilation
5. Maximal Sharing of CSE
6. Type based optimizations
7. SIMD code generation
8. One specification, Multiple Implementations

Future Directions

1. Vectors and Matrix Extensions
2. Improved SIMD Code Generation
3. Improved Normal Forms and Symbolic Simplifications
4. User Interface Extensions
5. Improved Diagram Generation
6. Integration in other softwares

Challenges

1. Massively Parallel Systems

2. Long term preservation of technological pieces

Challenge 1:

Massively Parallel Systems

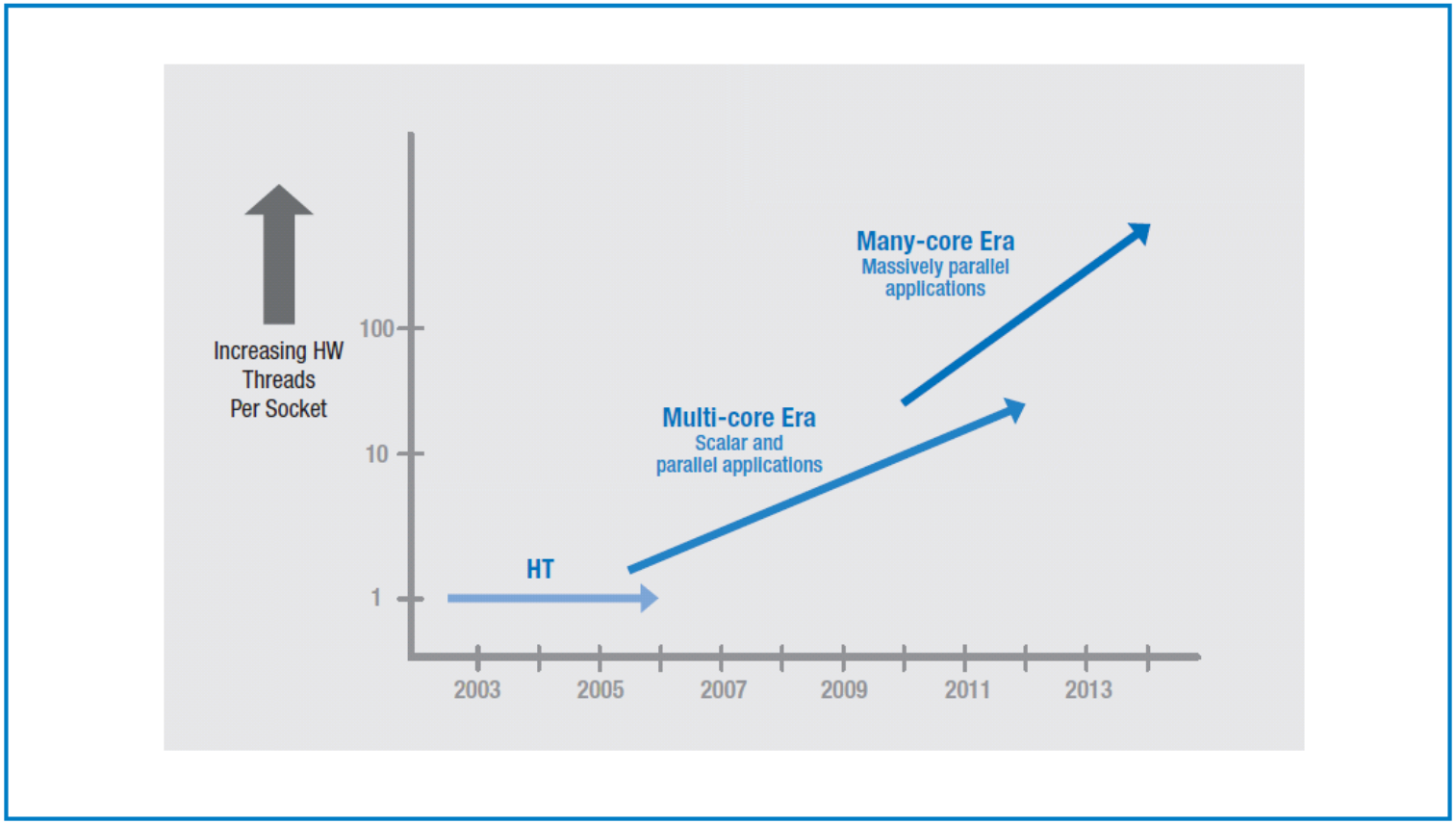


Figure 1: Current and expected eras of Intel® processor architectures

Challenge 2:

Long Term Preservation of Computer Music Programs

How could my technological pieces be played in 2358 ?

END

Questions ?