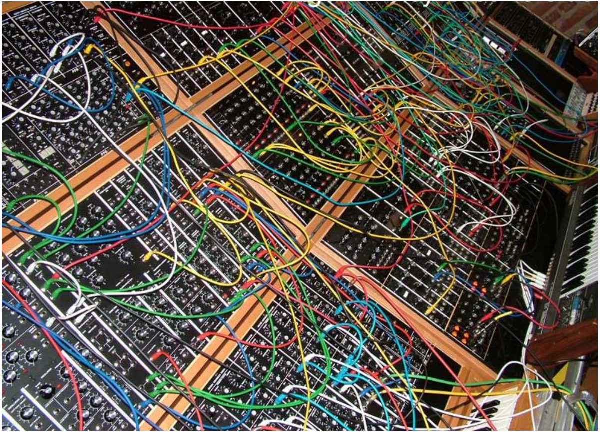
FAUST Tutorial for Functional Programmers

Y. Orlarey, S. Letz, D. Fober, R. Michon

ICFP 2017 / FARM 2017

What is Faust ?

What is Faust?



A programming language (DSL) to build electronic music instruments

Some Music Programming Languages

- 4CED
- Adagio
- AML
- AMPLE
- Antescofo
- Arctic
- Autoklang
- Bang
- Canon
- CHANT
- Chuck
- CLCE
- CMIX
- Cmusic
- CMUSIC
- Common Lisp Music
- Common Music
- Common Music Notation
- Csound
- CyberBand
- DARMS
- DCMP
- DMIX
- Elody
- EsAC
- Euterpea
- Extempore
- Faust
- Flavors Band
- Fluxus
- FOIL
- FORMES
- FORMULA
- Fugue
- Gibber
- GROOVE
- GUIDO
- HARP
- Haskore
- HMSL
- INV
- invokator
- KERN
- Keynote
- Kyma
- LOCO
- LPC
- Mars
- MASC
- Max
- MidiLisp
- MidiLogo
- MODE
- MOM
- Moxc
- MSX
- MUS10
- MUS8
- MUSCMP
- MuseData
- MusES
- MUSIC 10
- MUSIC 11
- MUSIC 360
- MUSIC 4B
- MUSIC 4BF
- MUSIC 4F
- MUSIC 6
- MCL
- MUSIC III/IV/V
- MusicLogo
- Music1000
- MUSIC7
- Musictex
- MUSIGOL
- MusicXML
- Musixtex
- NIFF
- NOTELIST
- Nyquist
- OPAL
- OpenMusic
- Organum1
- Outperform
- Overtone
- PE
- Patchwork
- PILE
- Pla
- PLACOMP
- PLAY1
- PLAY2
- PMX
- POCO
- POD6
- POD7
- PROD
- Puredata
- PWGL
- Ravel
- SALIERI
- SCORE
- ScoreFile
- SCRIPT
- SIREN
- SMDL
- SMOKE
- SSP
- SSSP
- ST
- Supercollider
- Symbolic Composer
- Tidal

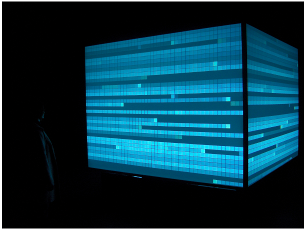
Brief Overview to Faust

- Faust offers end-users a high-level alternative to C to develop audio applications for a large variety of platforms.
- The role of the Faust compiler is to synthesize the most efficient implementations for the target language (C, C++, LLVM, Javascript, etc.).
- Faust is used on stage for concerts and artistic productions, for education and research, for open sources projects and commercial applications :

What Is Faust Used For ?

Artistic Applications

Sonik Cube (Trafik/Orlarey), Smartfaust (Gracia), etc.



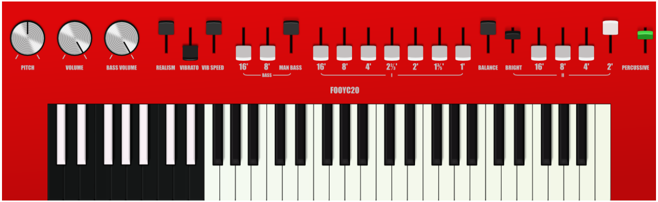
Open-Source projects

Guitarix: Hermann Meyer

WebAudio Applications

YC20 Emulator

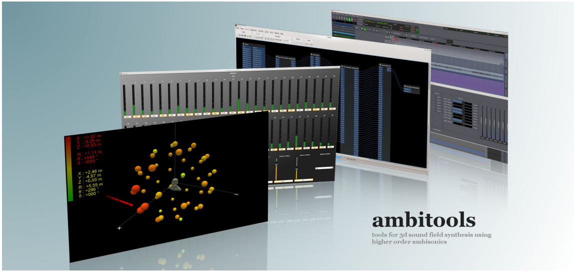
Thanks to the HTML5/WebAudio API and Asm.js it is now possible to run synthesizers and audio effects from a simple web page !



Sound Spatialization

Ambitools: Pierre Lecomte, CNAM

Ambitools (Faust Award 2016), 3-D sound spatialization using Ambisonic techniques.



Medical Applications

Brain Stethoscope: Chris Chafe, CCRMA-Stanford

Brain stethoscope turns seizures into music in hopes of giving the listener an empathetic and intuitive understanding of the neural activity.

Simulation Applications

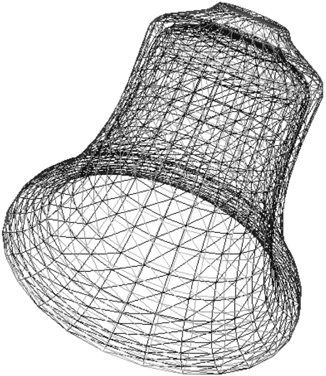
Stanford Car Simulator: Romain Michon, CCRMA-Stanford

Stanford Car Simulator, simulation of the sound of a car engine in Faust.

Simulation Applications

Bell simulations: Romain Michon and Chris Chafe, CCRMA-Stanford

CAD description of a bell turned into a procedural audio simulation in Faust



Hearing Aids Applications
Soniccloud, USA

Soniccloud: every cell phone call can be perfectly calibrated for an individual's unique Hearing Fingerprint – across 10 sonic dimensions.

The team at SonicCloud has had an outstanding experience working with Faust. Specifically, we have been able to optimize code to run our DSP algorithms in real-time without having to hand-optimize C/C++ code or write assembler. (Soniccloud)

Musical Instruments

Moforte, USA

Moforte (USA) designs musical instruments for iPad and iOS using Faust

Exercice 1: Djembe

Exercise 1: Djembe

Faust Online Compiler:

- https://faust.grame.fr/onlinecompiler

Faust Libraries Documentation:

- http://faust.grame.fr/libraries.html

Faust Code:

```
import("stdfaust.lib");

process = button("play")
        : pm.djembe(330,0.8,0.5,1);
```

Faust Signals and Time Model

Faust Signals and Time Model

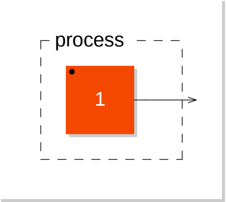
Faust programs operate on periodically sampled *signals*.

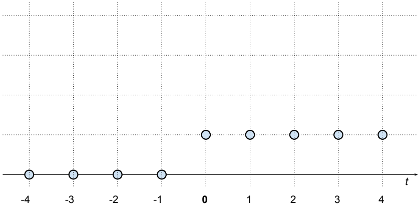
- A *signal* $s \in \mathbb{S}$ is a *time* to *sample* function.
- Two kinds of signals: $\mathbb{S} = \mathbb{S}_{\mathbb{Z}} \cup \mathbb{S}_{\mathbb{R}}$
  - Integer signals: $\mathbb{S}_{\mathbb{Z}} = \mathbb{Z} \rightarrow \mathbb{Z}$
  - Floating-point signals: $\mathbb{S}_{\mathbb{R}} = \mathbb{Z} \rightarrow \mathbb{R}$
- The value of a Faust signal is always 0 before time 0 :
  - $\forall s \in \mathbb{S}, s(t < 0) = 0$
- A Faust program denotes a *signal processor* $p \in \mathbb{P}$, a (continuous) function that maps a group of n input *signals* to a group of m output *signals* :
  - $\mathbb{P} = \mathbb{S}^n \rightarrow \mathbb{S}^m$

Faust Signals and Time Model

Example, a "constant" signal

```
process = 1;
```





$$y(t) = \begin{cases} 1 & t \geq 0 \\ 0 & t < 0 \end{cases}$$

Primitive Signal Processors

Faust Primitives

Arithmetic operations

Syntax	Type	Description
`+`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	addition: $y(t) = x_1(t) + x_2(t)$
`-`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	subtraction: $y(t) = x_1(t) - x_2(t)$
`*`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	multiplication: $y(t) = x_1(t) * x_2(t)$
$\wedge$	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	power: $y(t) = x_1(t)^{x_2(t)}$
`/`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	division: $y(t) = x_1(t)/x_2(t)$
`%`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	modulo: $y(t) = x_1(t)\%x_2(t)$
`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: $y(t) = (float)x(t)$

Faust Primitives

Bitwise operations

Syntax	Type	Description
`&`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	logical AND: $y(t) = x_1(t) \& x_2(t)$
`|`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	logical OR: $y(t) = x_1(t) \vert x_2(t)$
`xor`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	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: $y(t) = x_1(t) >> x_2(t)$

Faust Primitives

Comparison operations

Syntax	Type	Description
<	$\mathbb{S}^2 \to \mathbb{S}^1$	less than: $y(t) = x_1(t) < x_2(t)$
<=	$\mathbb{S}^2 \to \mathbb{S}^1$	less or equal: $y(t) = x_1(t) <= x_2(t)$
>	$\mathbb{S}^2 \to \mathbb{S}^1$	greater than: $y(t) = x_1(t) > x_2(t)$
>=	$\mathbb{S}^2 \to \mathbb{S}^1$	greater or equal: $y(t) = x_1(t) >= x_2(t)$
==	$\mathbb{S}^2 \to \mathbb{S}^1$	equal: $y(t) = x_1(t) == x_2(t)$
!=	$\mathbb{S}^2 \to \mathbb{S}^1$	different: $y(t) = x_1(t)! = x_2(t)$

Faust Primitives

Trigonometric functions

Syntax	Type	Description
`acos`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	arc cosine: $y(t) = \mathrm{acosf}(x(t))$
`asin`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	arc sine: $y(t) = \mathrm{asinf}(x(t))$
`atan`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	arc tangent: $y(t) = \mathrm{atanf}(x(t))$
`atan2`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	arc tangent of 2 signals: $y(t) = \mathrm{atan2f}(x_1(t), x_2(t))$
`cos`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	cosine: $y(t) = \mathrm{cosf}(x(t))$
`sin`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	sine: $y(t) = \mathrm{sinf}(x(t))$
`tan`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	tangent: $y(t) = \mathrm{tanf}(x(t))$

Faust Primitives

Other Math operations

Syntax	Type	Description
`exp`	$\mathbb{S}^1 \to \mathbb{S}^1$	base-e exponential: $y(t) = \mathrm{expf}(x(t))$
`log`	$\mathbb{S}^1 \to \mathbb{S}^1$	base-e logarithm: $y(t) = \mathrm{logf}(x(t))$
`log10`	$\mathbb{S}^1 \to \mathbb{S}^1$	base-10 logarithm: $y(t) = \mathrm{log10f}(x(t))$
`pow`	$\mathbb{S}^2 \to \mathbb{S}^1$	power: $y(t) = \mathrm{powf}(x_1(t), x_2(t))$
`sqrt`	$\mathbb{S}^1 \to \mathbb{S}^1$	square root: $y(t) = \mathrm{sqrtf}(x(t))$
`abs`	$\mathbb{S}^1 \to \mathbb{S}^1$	absolute value (int): $y(t) = \mathrm{abs}(x(t))$
		absolute value (float): $y(t) = \mathrm{fabsf}(x(t))$
`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: $y(t) = \mathrm{fmodf}(x_1(t), x_2(t))$
`remainder`	$\mathbb{S}^2 \to \mathbb{S}^1$	float remainder: $y(t) = \mathrm{remainderf}(x_1(t), x_2(t))$
`floor`	$\mathbb{S}^1 \to \mathbb{S}^1$	largest int $\leq$: $y(t) = \mathrm{floorf}(x(t))$
`ceil`	$\mathbb{S}^1 \to \mathbb{S}^1$	smallest int $\geq$: $y(t) = \mathrm{ceilf}(x(t))$
`rint`	$\mathbb{S}^1 \to \mathbb{S}^1$	closest int: $y(t) = \mathrm{rintf}(x(t))$

Faust Primitives

Delays and Tables

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

Faust Primitives

User Interface Primitives

Syntax	Example
`button(`*str*`)`	`button("play")`
`checkbox(`*str*`)`	`checkbox("mute")`
`vslider(`*str*`,`*cur*`,`*min*`,`*max*`,`*inc*`)`	`vslider("vol",50,0,100,1)`
`hslider(`*str*`,`*cur*`,`*min*`,`*max*`,`*inc*`)`	`hslider("vol",0.5,0,1,0.01)`
`nentry(`*str*`,`*cur*`,`*min*`,`*max*`,`*inc*`)`	`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)`

Exercice 2: Adding rhythm and sliders to the Djembe

Exercise 2: Adding rhythm to the Djembe

Faust Code:

```
import("stdfaust.lib");

process = button("play"), // try checkbox("play")
          ba.pulsen(100, 4800) : *
        : pm.djembe(330,0.8,0.5,1);
```

Exercise 3: Adding sliders to the Djembe

Faust Code:

```
import("stdfaust.lib");

process = checkbox("play"),
          ba.pulsen(100, 4800) : *
        : pm.djembe(
            hslider("freq", 300, 100, 1000, 1),
            hslider("position", 0.8, 0, 1, 0.1),
            hslider("sharpness", 0.5, 0, 1, 0.1),
            hslider("gain",0.5,0,1,0.1)
          );
```

Exercise 4: Adding an echo to the Djembe

Faust Code:

```
import("stdfaust.lib");

echo = +~(@(22100):*(hslider("fb",0,0,1,0.01)));

process = checkbox("play"),
          ba.pulsen(100, 4800) : *
        : pm.djembe(
            hslider("freq", 300, 100, 1000, 1),
            hslider("position", 0.8, 0, 1, 0.1),
            hslider("sharpness", 0.5, 0, 1, 0.1),
            hslider("gain",0.5,0,1,0.1)
          )
        : echo;
```

Programming by composition

Programming by Composition

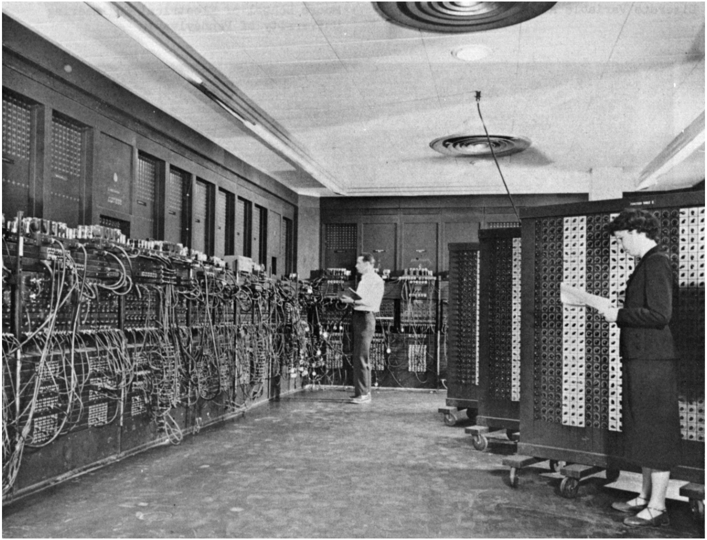
Block-Diagram Algebra

Programming by patching is familiar to musicians :

Programming by Composition

Block-Diagram Algebra

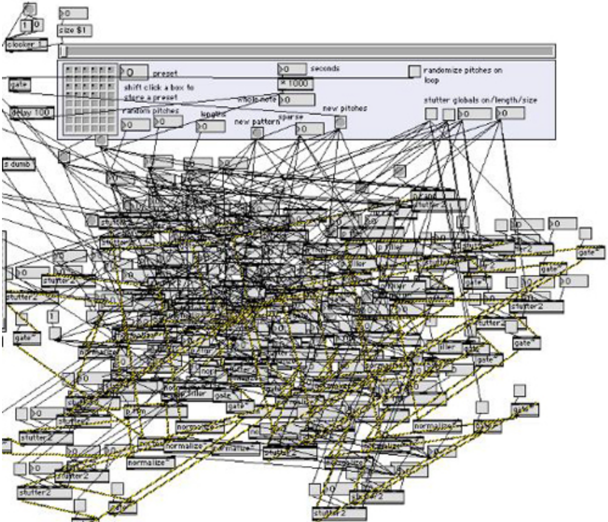
Programming by patching, the ENIAC computer :



Programming by Composition

Block-Diagram Algebra

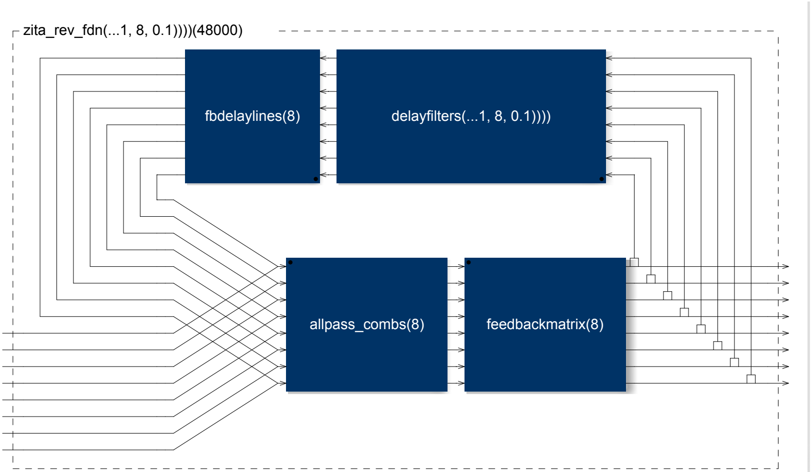
Block-diagrams are widely used in Visual Programming Languages like Max/MSP:



Programming by Composition

Block-Diagram Algebra

Faust allows structured block-diagrams, here part of the zita reverb.



Programming by Composition

Composition Operations

- `(A<:B)` split composition (associative, priority 1)
- `(A:>B)` merge composition (associative, priority 1)
- `(A:B)` sequential composition (associative, priority 2)
- `(A,B)` parallel composition (associative, priority 3)
- `(A~B)` recursive composition (priority 4)

Programming by Composition

Same Expression in Lambda-Calculus, FP and Faust

Lambda-Calculus

```
\x.\y.(x+y,x*y) 2 3
```

FP/FL (John Backus)

```
[+,*]:<2,3>
```

Faust

```
2,3 <: +,*
```

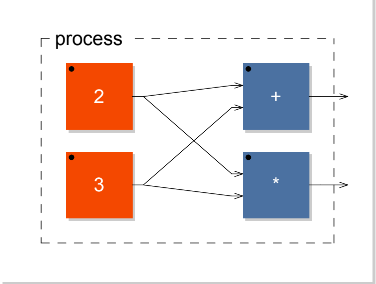


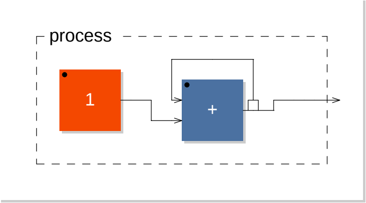
Figure: block-diagram of `2,3 <: +,*`

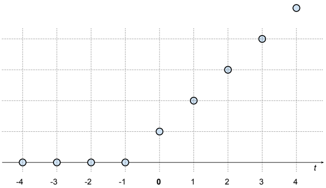
Programming by Composition

A Very Simple Example

```
process = 1 : +~_;
```

$$y(t) = \begin{cases} 0 & t < 0 \\ 1 + y(t-1) & t \geq 0 \end{cases}$$





Programming by Composition

Parallel Composition (associative, priority 3)

The *parallel composition* `(A,B)` is probably the simplest one. It places the two block-diagrams one on top of the other, without connections.

`(A,B)`: $(\mathbb{S}^n \to \mathbb{S}^m) \to (\mathbb{S}^{n'} \to \mathbb{S}^{m'}) \to (\mathbb{S}^{n+n'} \to \mathbb{S}^{m+m'})$

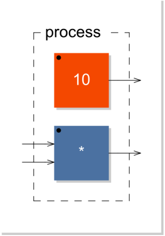


Figure: Example of parallel composition `(10,*)`

Programming by Composition

Sequential Composition (associative, priority 2)

The *sequential composition* `(A:B)` connects the outputs of `A` to the corresponding inputs of `B`.

`(A:B)`: $(\mathbb{S}^n \rightarrow \mathbb{S}^m) \rightarrow (\mathbb{S}^m \rightarrow \mathbb{S}^p) \rightarrow (\mathbb{S}^n \rightarrow \mathbb{S}^p)$

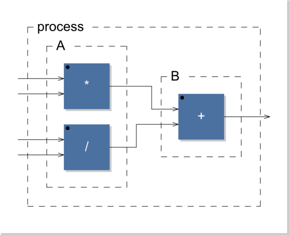


Figure: Example of sequential composition `((*,/):+)`

Programming by Composition

Split Composition (associative, priority 1)

The *split composition* `(A<:B)` operator is used to distribute the outputs of `A` to the inputs of `B`

`(A<:B)`: $(\mathbb{S}^n \rightarrow \mathbb{S}^m) \rightarrow (\mathbb{S}^{k.m} \rightarrow \mathbb{S}^p) \rightarrow (\mathbb{S}^n \rightarrow \mathbb{S}^p)$

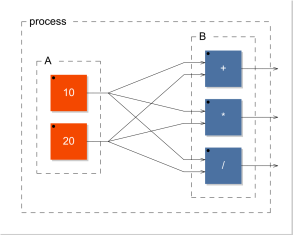


Figure: example of split composition `((10,20) <: (+,*,/))`

Programming by Composition

Merge Composition (associative, priority 1)

The *merge composition* `(A:>B)` is used to connect several outputs of `A` to the same inputs of `B`. Signals connected to the same input are added.

`(A:>B)`: $(\mathbb{S}^n \to \mathbb{S}^{k.m}) \to (\mathbb{S}^m \to \mathbb{S}^p) \to (\mathbb{S}^n \to \mathbb{S}^p)$

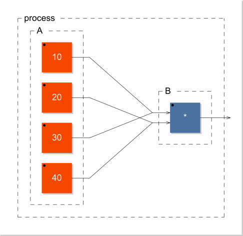


Figure: example of merge composition `((10,20,30,40) :> *)`

Programming by Composition

Recursive Composition (priority 4)

The *recursive composition* `(A~B)` is used to create cycles in the block-diagram in order to express recursive computations.

`(A~B)`: $(\mathbb{S}^{n+n'} \rightarrow \mathbb{S}^{m+m'}) \rightarrow (\mathbb{S}^{m'} \rightarrow \mathbb{S}^{n'}) \rightarrow (\mathbb{S}^{n} \rightarrow \mathbb{S}^{m+m'})$

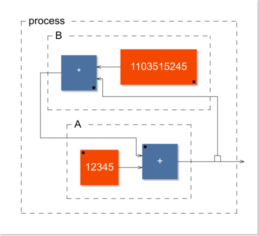
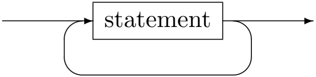


Figure: example of recursive composition `+(12345) ~ *(1103515245)`

Faust Program

Faust Program

program



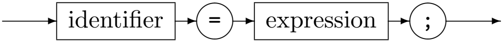
- A Faust program is essentially a list of *statements*. These statements can be :
  - metadata *declarations*,
  - file *imports*
  - *definitions*
- Example :

```
declare name        "noise";
declare copyright   "(c)GRAME␣2006";
import("music.lib");
process = noise * vslider("volume", 0, 0, 1, 0.1);
```

Definitions

Simple Definitions

definition



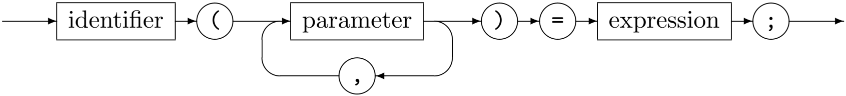
- A *definition* associates an identifier with an expression it stands for. Example :

```
random = +(12345) ~ *(1103515245);
```

Definitions

Functions' definitions

definition



- Definitions with formal parameters correspond to functions' definitions. Example :

```
linear2db(x) = 20*log10(x);
```

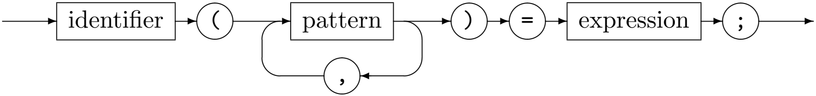
- Alternative notation using a *lambda-abstraction*:

```
linear2db = \(x).(20*log10(x));
```

Definitions

Pattern Matching Definitions

definition



- Formal parameters can also be full expressions representing patterns. Example :

```
duplicate(1,exp) = exp;
duplicate(n,exp) = exp, duplicate(n-1,exp);
```

- Alternative notation :

```
duplicate = case {
              (1,exp) => exp;
              (n,exp) => duplicate(n-1,exp);
            };
```

Statement

Import file

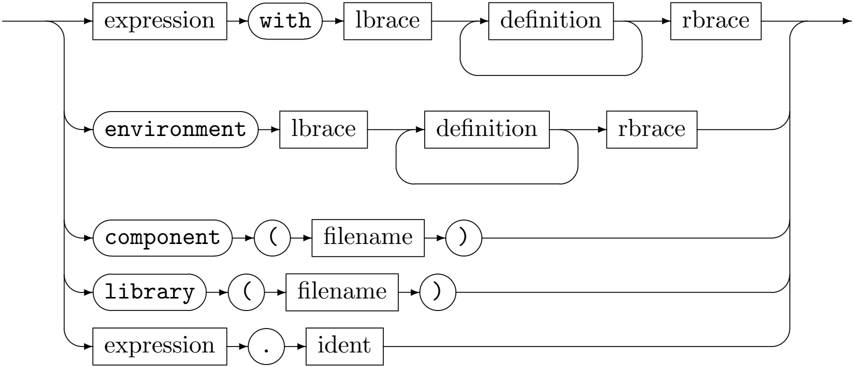
fileimport

- allows to import definitions from other source files.
- for example `import("math.lib");` imports the definitions from `"math.lib"` file, a set of additional mathematical functions provided as foreign functions.

Expressions

Environments

envexp

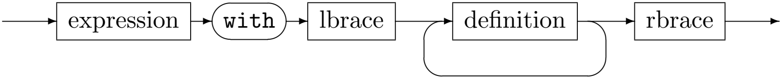


- Each Faust expression has an associated *lexical environment*

Environments

With Expression

withexpression



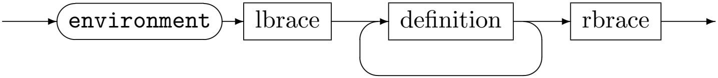
- With expression allows to specify a *local environment*, a private list of definitions that will be used to evaluate the left hand expression
- example pink noise filter :

```
pink = f : + ~ g with {
     f(x) = 0.04957526213389*x
          - 0.06305581334498*x@1
          + 0.01483220320740*x@2;
     g(x) = 1.80116083982126*x
          - 0.80257737639225*x@1;
    };
```

Environments

Environment

environment



- an `environment` is used to group together related definitions :

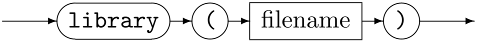
```
constant = environment {
    pi = 3.14159;
     e = 2,718 ;
     ....
   };
```

- definitions of an environment can be easily accessed : `constant.pi`

Environments

Library

library

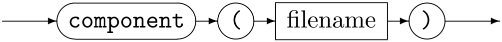


- allows to create an environment by reading the definitions from a file.
- example : `library("filter.lib")`
- definitions are accesed like this : `library("filter.lib").smooth`

Environments

Component

component



- allows to reuse a full Faust program as a simple expression.
- example :

```
component("osc.dsp")<:component("freeverb.dsp")
```

- equivalence between :

```
component("freeverb.dsp")
```

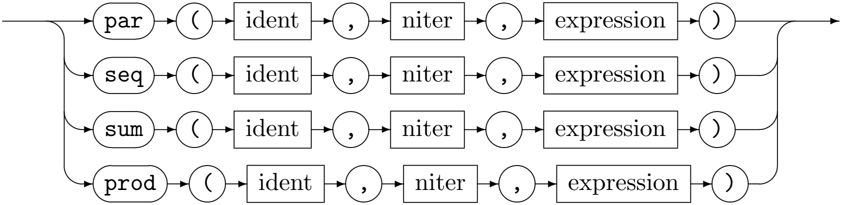
and

```
library("freeverb.dsp").process
```

Expressions

Iterations

diagiteration



- Iterations are analog to `for(...)` loops
- provide a convenient way to automate some complex block-diagram constructions.

Expressions

Iterations

The following example shows the use of `seq` to create a 10-bands filter:

```
process  =      seq(i, 10,
                vgroup("band␣%i",
                   bandfilter( 1000*(1+i) )
                )
            );
```

Exercise 5: Djembe automation

Exercise 5: Djembe automation

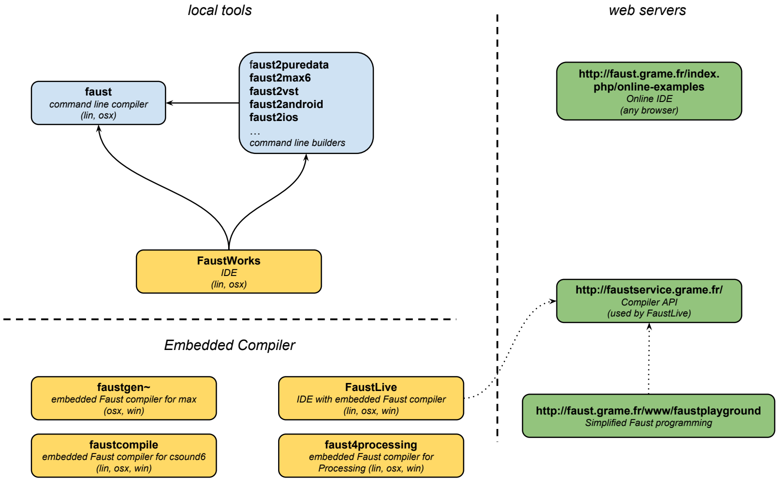
Faust Code:

```
import("stdfaust.lib");

saw(f) = f/ma.SR : (+,1:fmod)~_;

process = checkbox("play"),
           ba.pulsen(100, 4800) : *
        : pm.djembe(
           hslider("freq", 300, 100, 1000, 1),
           saw(hslider("fpos",1,0.05,20,0.01)),
           saw(hslider("fsharp",1,0.05,20,0.01)),
           saw(hslider("fgain",1,0.05,20,0.01))
          )
        ;
```

Faust Ecosystem

Faust Ecosystem

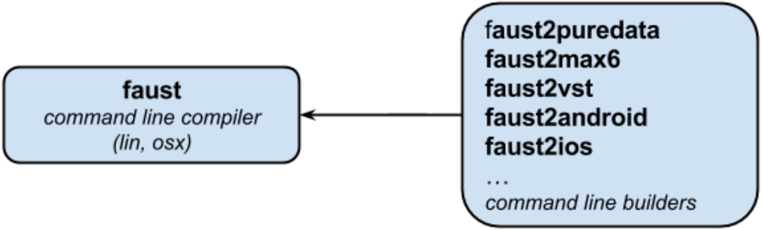
Overview



Faust Ecosystem

Command-line Tools

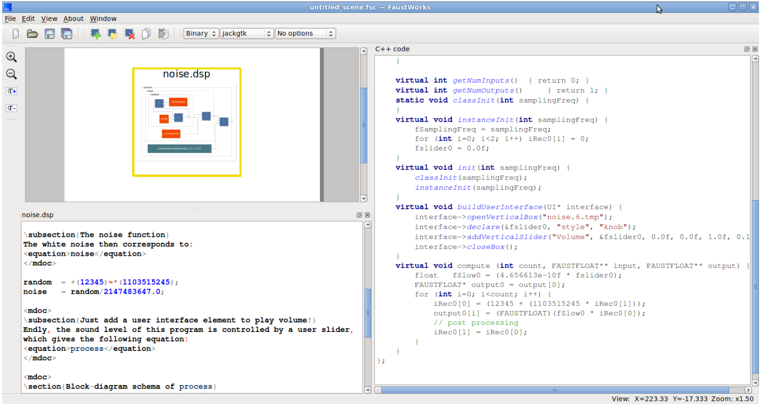
Simplify the compilation workflow : full automated process to build Android and iOS applications, VST plugins, Max/MSP externals, Csound opcodes, etc.



Faust Ecosystem

FaustWorks

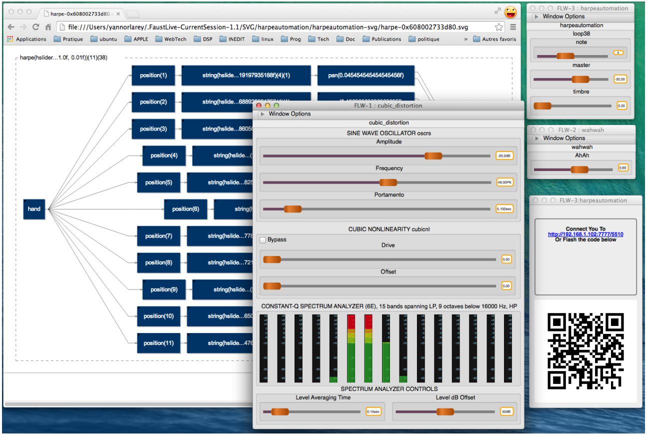
FaustWorks can simplify Faust learning in particular by providing "realtime" code and diagram generation:



Faust Ecosystem

FaustLive

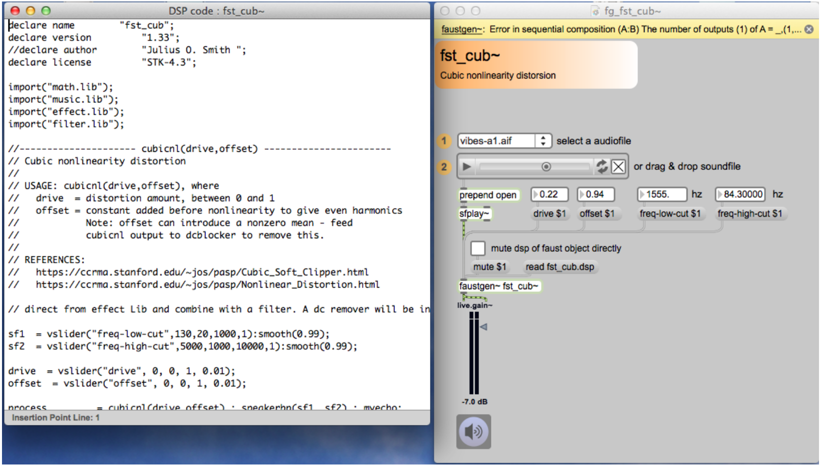
FaustLive speeds up the Edit/Compile/Run. It provides advanced cooperation features :



Faust Ecosystem

Faustgen

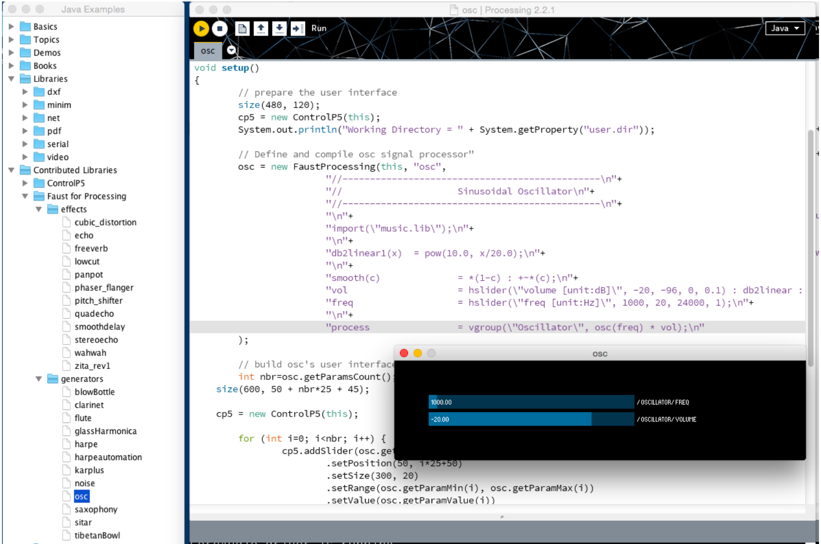
Faustgen speeds up the Edit/Compile/Run within the Max framework:



Faust Ecosystem

Faust4processing

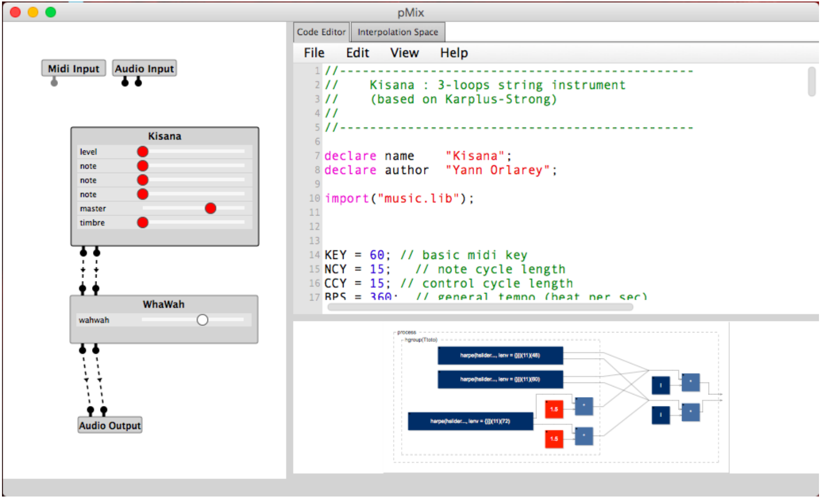
Faust4processing provides an embedded Faust compiler for Processing:



Faust Ecosystem

PMIX (Oliver Larkin)

PMIX speeds up the Edit/Compile/Run within VST host:

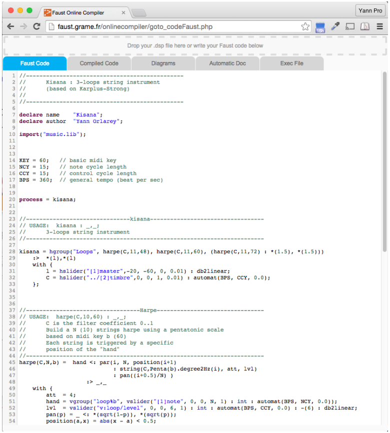


Faust Ecosystem

Online compiler

The Online compiler can be used from a web browser to compile Faust programs for a variety of systems, including the Web.

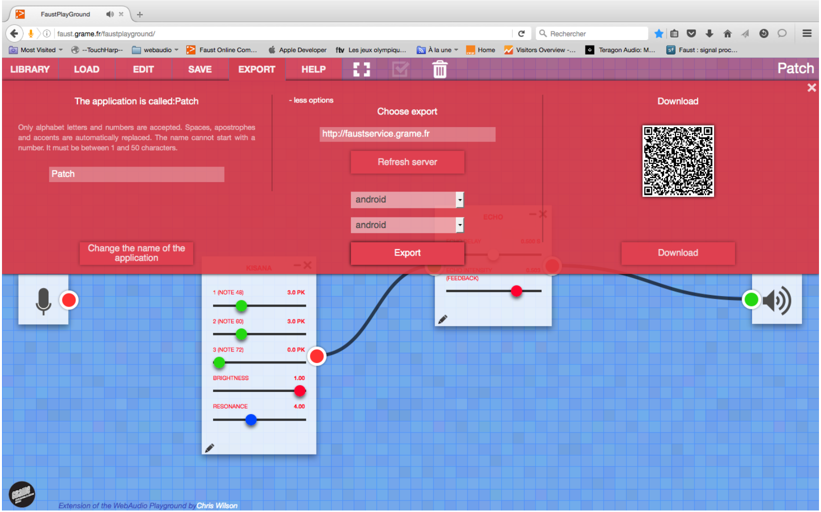
`http://faust.grame.fr/onlinecompiler`



Faust Ecosystem

Faust playground

The Web as a gigantic Lego box to reuse and recompose audio applications. `http://faust.grame.fr/faustplayground`



Thanks! Questions?