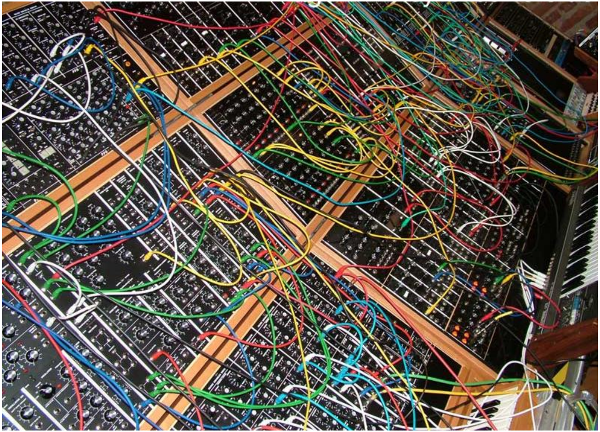
The Faust Programming Language

Yann Orlarey
EQUIPE EMERAUDE

INSA 2022

What is Faust?



A programming language (DSL) to build electronic music instruments, audio plugins, signal processing applications, etc.

Music Programming Languages

Music Languages

Music Languages

Chronology

- 1960 : Music III (Max Mathews) Unit Generators ;
- 1968 : Music V written in Fortran ;
- 1985 : Csound (Barry Vercoe) a port of Music 11 ;
- 1987 : Max (Miller Puckette) visual programming ;
- 1991 : Max/MSP (Miller Puckette) signal processing ;
- 1996 : Supercollider (John McCartney) OOP, client-server ;
- 1996 : Pure Data (Miller Puckette) open Source ;
- 2002 : Faust (Yann Orlarey) compiled, multi targets ;
- 2003 : Chuck (Ge Wang, Perry Cook) Live Coding ;
- 2011 : Gen (Graham Wakefield) compiled, multi targets.
- 2022 : RNBO (Cycling 74) compiled, multi targets.

Music Languages

Some examples

- 4CED
- Adagio
- AML
- AMPLE
- Antescofo
- Arctic
- Autoklang
- Bang
- Canon
- CHANT
- Chuck
- CLCE
- C-major
- 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
- Kronos
- 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
- RNBO
- SALIERI
- SCORE
- ScoreFile
- SCRIPT
- SIREN
- SMDL
- SMOKE
- SOUL
- SSSP
- ST
- Supercollider
- Symbolic Composer
- Tidal

Overview of Faust

What is Faust used for?

- Faust is used on stage for concerts and artistic productions, for education and research, for open sources projects and commercial applications :
- 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.).

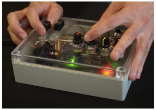
Faust is fully compiled

- Fully compiled to native code
- Sample level semantics
- Multiple backends: C++, WebAssembly, Rust, LLVM, etc.
- Code runs on most platforms: embedded systems, web pages, mobile devices, plug-ins, standalone applications, (fpga), etc.









Tutorial Part

Using the Faust IDE

```
import("stdfaust.lib");
process = button("play") : pm.djembe(60,0.3,0.4,1);
```

https://faustide.grame.fr

The Design of Faust

Design Choices

- Purely functional approach focused on signal processing (λC)
- Programming by composition (FP, CL)
- A Compiled high-level specification language for end-users
- Well-defined preservable formal semantics
- Easy deployment

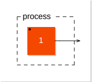
Purely Functional Approach

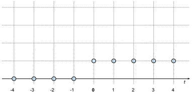
- Signals are functions: $\mathbb{S} = \mathbb{Z} \rightarrow \mathbb{R}$,
- Faust primitives are signal processors: $\mathbb{P} = \mathbb{S}^m \rightarrow \mathbb{S}^n$,
- Faust composition operations (`<:  :>  :  ,  ~`) are binary functions on signal processors: $\mathbb{A} = \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$,
- User defined functions are higher order functions on signal processors: $\mathbb{U} = \mathbb{P}^n \rightarrow \mathbb{P}$,
- A Faust program denotes a signal processor.

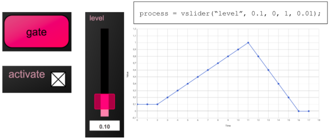
Faust Primitives

Generators: $\mathbb{S}^0 \rightarrow \mathbb{S}^1$

```
process = 1;
```





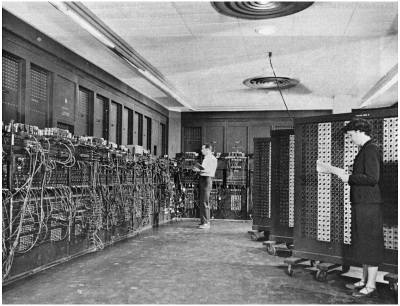


Operations: $\mathbb{S}^n \rightarrow \mathbb{S}^m$

- Arithmetic: `+`, `-`,`*`, `/`,...
- Comparison: `<`, `<=`,`!=`,...
- Trigonometric: `sin`, `cos`,...
- Log and Co.: `log`, `exp`,...
- Min, Max: `min`, `max`,...
- Selectors: `select2`,...
- Delays and Tables: `@`,...
- GUI: `button("...")`,...

Block-Diagram Algebra

Programming by patching is familiar to musicians :



Block-Diagram Algebra

Today programming by patching is widely used in Visual Programming Languages like Max/MSP:

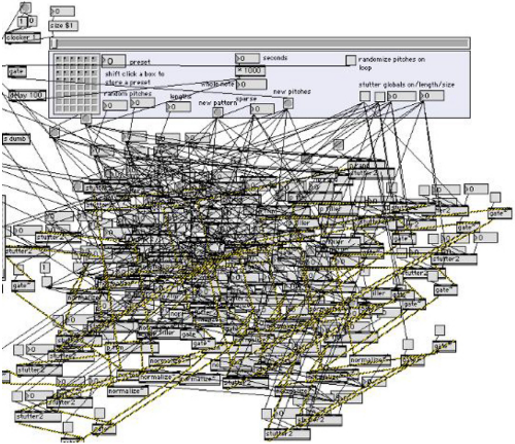


Figure: Block-diagrams can be a mess

Block-Diagram Algebra

Faust allows structured block-diagrams

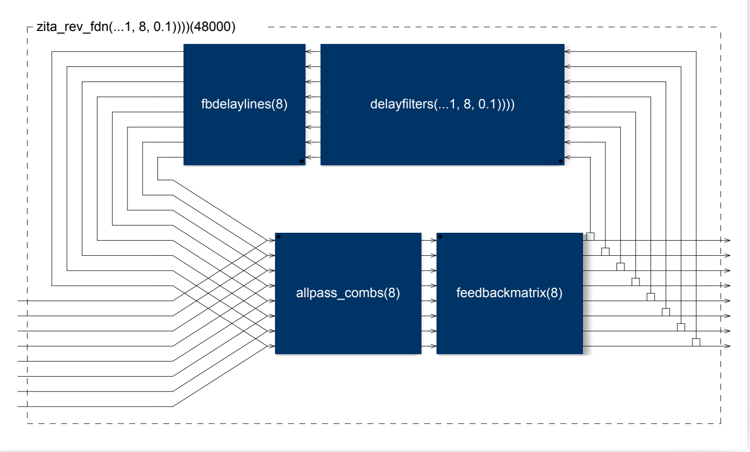


Figure: A complex but structured block-diagram

Block-Diagram Algebra

Faust syntax is based on a *block diagram algebra*

5 Composition Operators

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

2 Constants

- `!` cut
- `_` wire

Block-Diagram Algebra
Parallel Composition

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

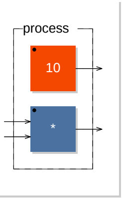


Figure: Example of parallel composition `(10,*)`

Block-Diagram Algebra

Sequential Composition

The *sequential composition* $(A : B)$ connects the outputs of A to the inputs of B. $A[0]$ is connected to $[0]B$, $A[1]$ is connected to $[1]B$, and so on.

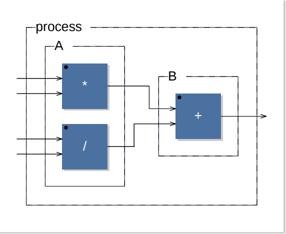


Figure: Example of sequential composition `((*,/):+)`

Note that the number of outputs of A must be equal to the number of inputs of B.

Block-Diagram Algebra

Split Composition

The *split composition* $(A <: B)$ operator is used to distribute A outputs to B inputs.

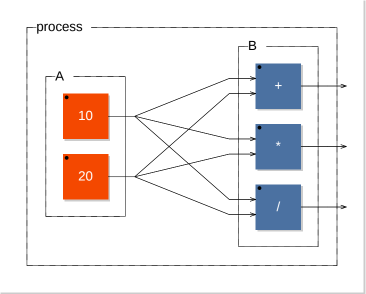


Figure: example of split composition `((10,20) <: (+,*,/))`

Block-Diagram Algebra

Merge Composition

The *merge composition* $(A :> B)$ is used to connect several outputs of A to the same inputs of B.

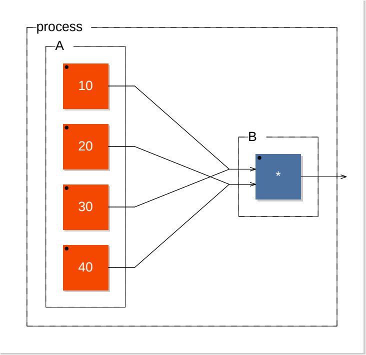


Figure: example of merge composition `((10,20,30,40) :> *)`

Block-Diagram Algebra

Recursive Composition

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

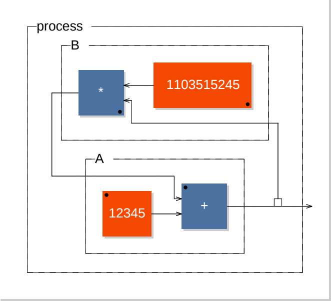


Figure: example of recursive composition `+(12345) ~ *(1103515245)`

Expressivity Quest

Language Expressivity

- Function Composition
- Anonymity
- Faust programs as components
- Partial application
- Lexical environments as first class citizen
- Pattern Matching

Language Expressiveness

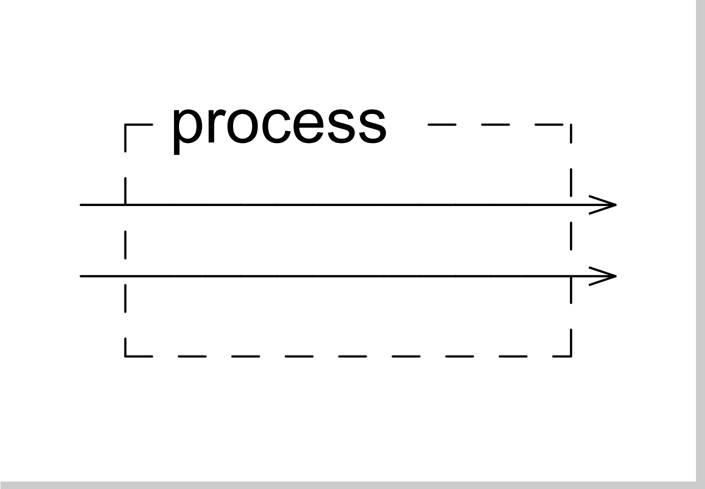
Fast Fourier Transform

```
fft(N) = si.cbus(N) : an.c_bit_reverse_shuffle(N) : fftb(N)
with {
    fftb(1) = _,_;
    fftb(N) = si.cbus(N)
            : (fftb(No2)<:(si.cbus(No2), si.cbus(No2))),
            (fftb(No2) <: (si.cbus(N):twiddleOdd(N)))
            :> si.cbus(N)
        with {
        No2 = int(N)>>1;
        twiddleOdd(N) = par(k,N,si.cmul(cos(w(k)),0-sin(w(k))));
        w(k) = 2.0*ma.PI*float(k)/float(N);
        };
};
```

Language Expressiveness

Fast Fourier Transform

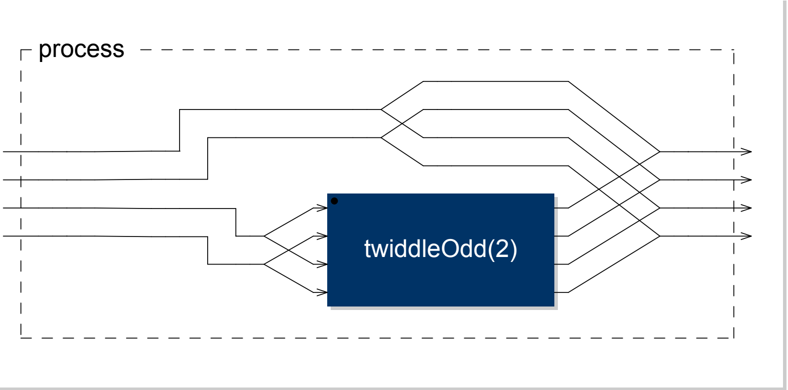
```
fft(1)
```



Language Expressiveness

Fast Fourier Transform

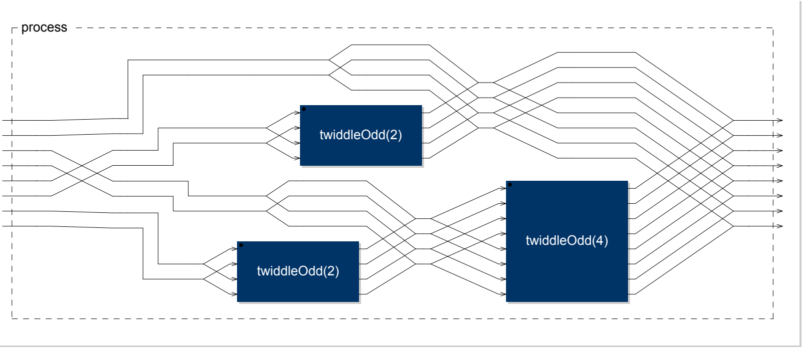
```
fft(2)
```



Language Expressiveness

Fast Fourier Transform

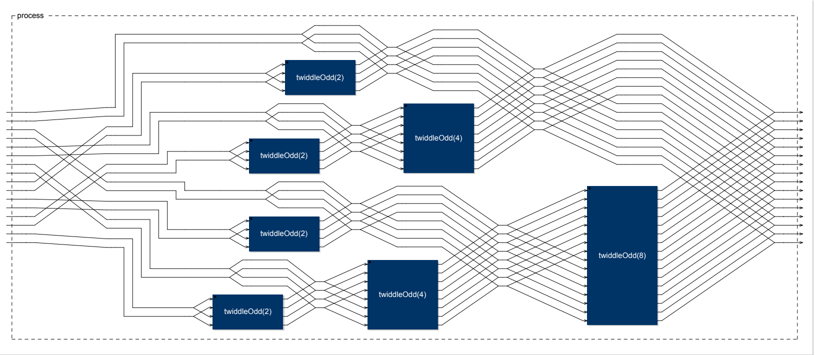
```
    fft(4)
```



Language Expressiveness

Fast Fourier Transform

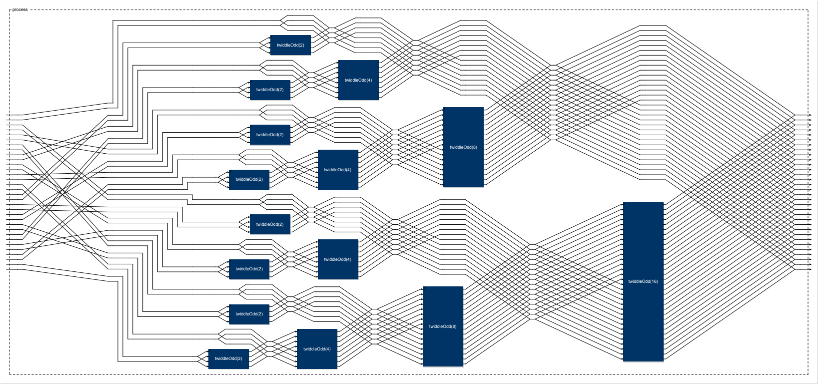
```
fft(8)
```



Language Expressiveness

Fast Fourier Transform

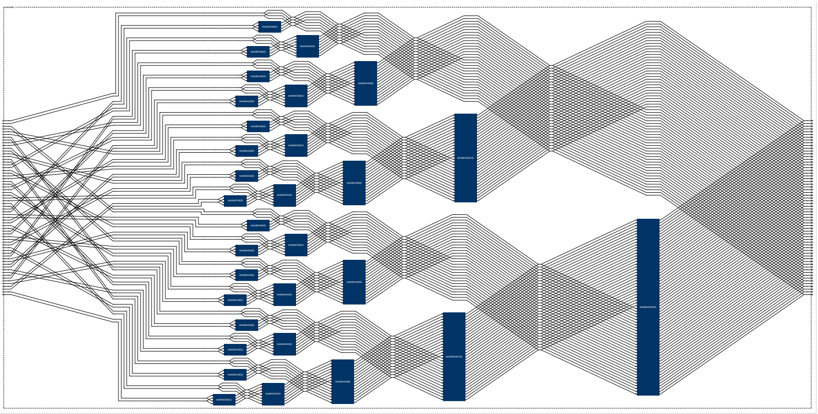
```
fft(16)
```



Language Expressiveness

Fast Fourier Transform

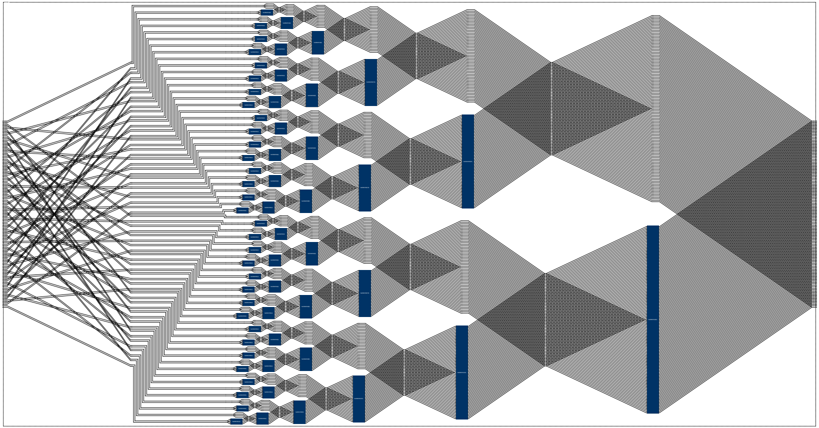
```
fft(32)
```



Language Expressiveness

Fast Fourier Transform

```
fft(64)
```



Performance Quest

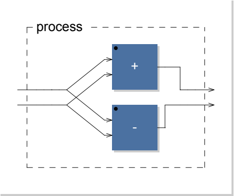
- Fully compiled to native code
- Sample level semantics
- Specification language
- Automatic parallelization

Fully compiled to native code

Faust code:

```
process = _,_ <: +,-;
```

Block-diagram:



C++ translation:

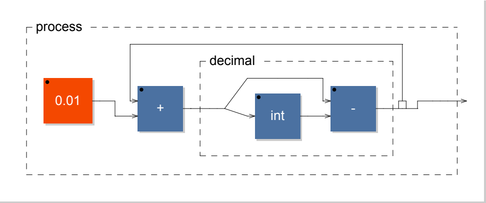
```
for (int i = 0; (i < count); i = (i + 1)) {
    float fTemp0 = input0[i];
    float fTemp1 = input1[i];
    output0[i] = fTemp0 + fTemp1;
    output1[i] = fTemp0 - fTemp1;
}
```

Sample level semantics

Sawtooth signal by step of 0.01:

```
decimal = _ <: _, int : -;
process = 0.01 : (+:decimal) ~ _;
```

Block-diagram:



Signal equation:

$$
\begin{aligned}
y(t < 0) &= 0 \\
y(t \geq 0) &= decimal(y(t-1) + 0.01)
\end{aligned}
$$

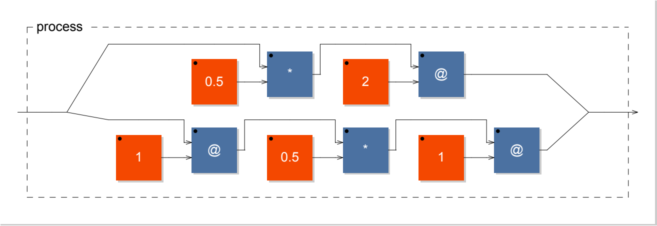
Specification Language

Leave the implementation to the compiler

User's code:

```
process = _<:(*(0.5):@(2)),(@(1):*(0.5):@(1)):>_;
```

Block-diagram:

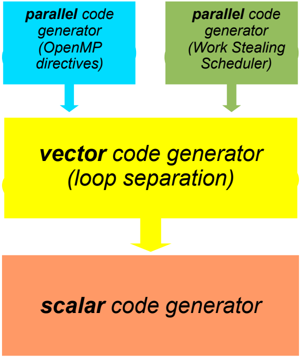


Equivalent, more efficient code

```
process = @(2);
```

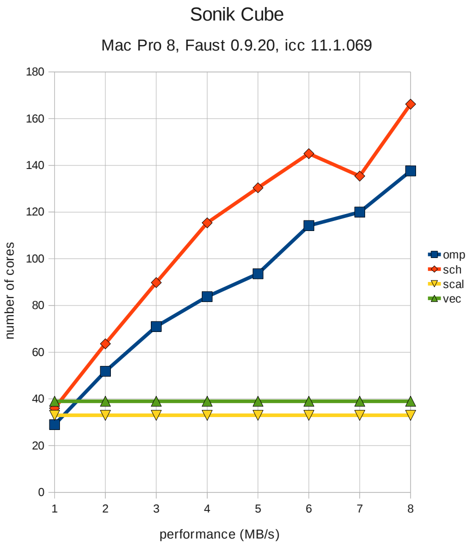
Automatic Parallelization

Code Generators



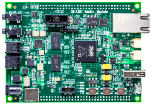
Automatic Parallelization

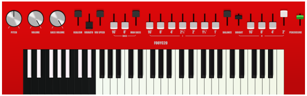
Performances



Easy Deployment Quest

Easy Deployment

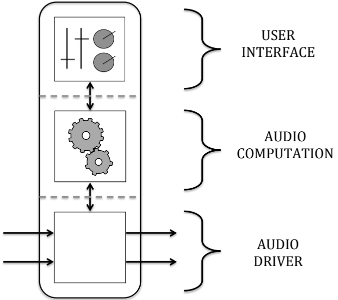


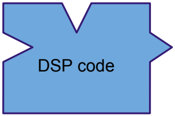


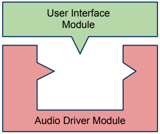
Easy Deployment

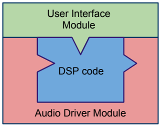
Separation of concern

The *architecture file* describes how to connect the audio computation to the external world.









Easy Deployment

Examples of supported architectures

- Audio plugins :
  - AudioUnit
  - LADSPA
  - DSSI
  - LV2
  - Max/MSP
  - VST
  - PD
  - Csound
  - Supercollider
  - Pure
  - Chuck
  - JUCE
  - Unity
- Devices :
  - OWL
  - MOD
  - BELA
  - SAM
- Audio drivers :
  - Jack
  - Alsa
  - CoreAudio
  - Web Audio API
- Graphic User Interfaces :
  - QT
  - GTK
  - Android
  - iOS
  - HTML5/SVG
- Other User Interfaces :
  - MIDI
  - OSC
  - HTTPD

Ubiquity:
Compiling Everywhere

Compiling Everywhere

Language Backends

- **C++**
- C
- Rust
- Java
- Javascript
- Asm.js
- **LLVM**
- **WebAssembly**
- ...

Compiling Everywhere

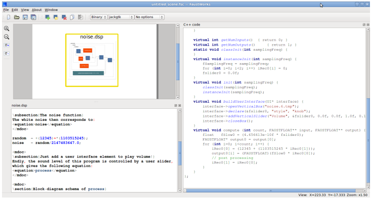
Libfaust

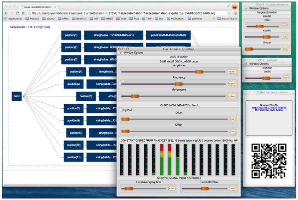
- Libfaust: embeddable version of the Faust compiler coupled with LLVM
- Libfaust.js: embeddable Javascript version of the Faust compiler

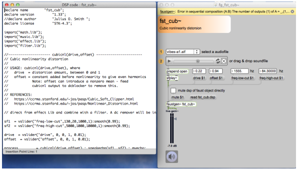
Compiling Everywhere

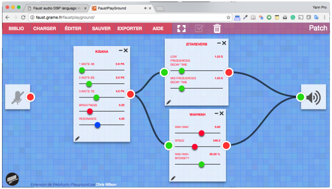
- Command Line Compilers
  - `faust` command line
  - `faust2xxx` command line
  - FaustWorks (IDE)
- Embedded Compilers (libfaust)
  - FaustLive (self contained)
  - Faustgen for Max/MSP
  - Faust for PD
  - Faustcompile, etc. for Csound (V. Lazzarini)
  - Faust4processing
  - Antescofo (IRCAM's score follower)
- Web Based Compilers
  - Faustweb API (`https://faustservice.grame.fr`)
  - Online Development Environment (`https://faustide.grame.fr/`)
  - Online Editor (`https://fausteditor.grame.fr/`)
  - Faustplayground (`https://faustplayground.grame.fr/`)

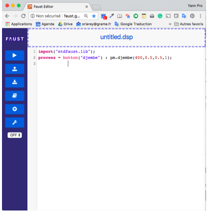
The Faust Ecosystem

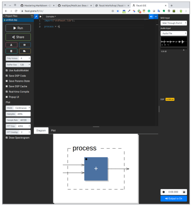












Additional Resources

Where to learn Faust

International:

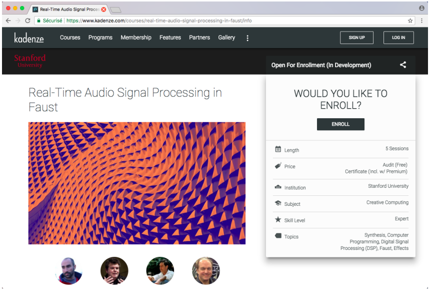
- Stanford U./CCRMA
- Maynooth University
- Louisiana State University
- Aalborg University

France:

- Jean Monnet U., Master RIM
- IRCAM, ATIAM
- PARIS 8

Where to learn Faust

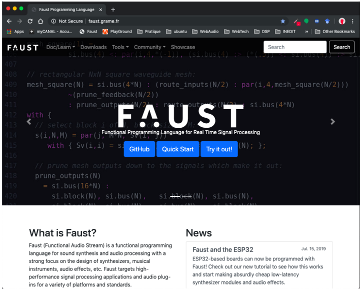
Kadenze course



https://www.kadenze.com/courses/real-time-audio-signal-processing-in-faust/info

Where to learn Faust

Faust website



https://faust.grame.fr