# Faust

Yann Orlarey

JIM-Montréal, mai 2015

# 1-Faust Overview

# Brief Overview to Faust

## Some Music DSLs

- 4CED
- Adagio
- AML
- AMPLE
- 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

http://faust.grame.fr

- Faust is a *Domain-Specific Language* for real-time signal processing and synthesis (like *Csound*, *Max/MSP*, *Supercollider*, *Puredata*,...).
- A Faust program denotes a *signal processor* : a *(continuous) function* that maps input *signals* to output *signals*.
- Programming in Faust is essentially combining *signal processors* using an algebra of 5 composition operations :

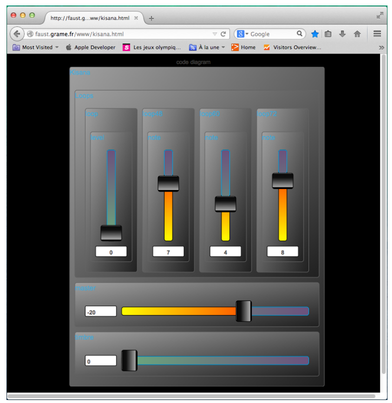
```
process = noise*hslider("level",0,0,1,0.01);
noise = +(12345)~*(1103515245):/(2147483647.0);
```

# Brief Overview to Faust

http://faust.grame.fr

- Faust offers end-users a high-level alternative to C to develop audio applications for a large variety of platforms, from desktop to web applications, from audio plug-ins to embedded systems.
- 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 :





# 2-Programming by Composition

# Programming by Composition

## Faust programs are *signal processors*

- A Faust program denotes a *signal processor* $p : \mathbb{S}^n \rightarrow \mathbb{S}^m$, a (continuous) function that maps a group of $n$ input *signals* to a group of $m$ output *signals*.
- Two kinds of signals :
  - Integer signals : $\mathbb{S}_\mathbb{Z} = \mathbb{Z} \rightarrow \mathbb{Z}$
  - Floating-point signals : $\mathbb{S}_\mathbb{R} = \mathbb{Z} \rightarrow \mathbb{R}$
  - $\mathbb{S} = \mathbb{S}_\mathbb{Z} \cup \mathbb{S}_\mathbb{R}$
- The value of a Faust signal is always 0 before time 0 :
  - $\forall s \in \mathbb{S}, s(t < 0) = 0$
- Programming in Faust is essentially composing signal processors together using an algebra of five composition operations : `<:` `:>` `:` `,` `~`

# Programming by Composition

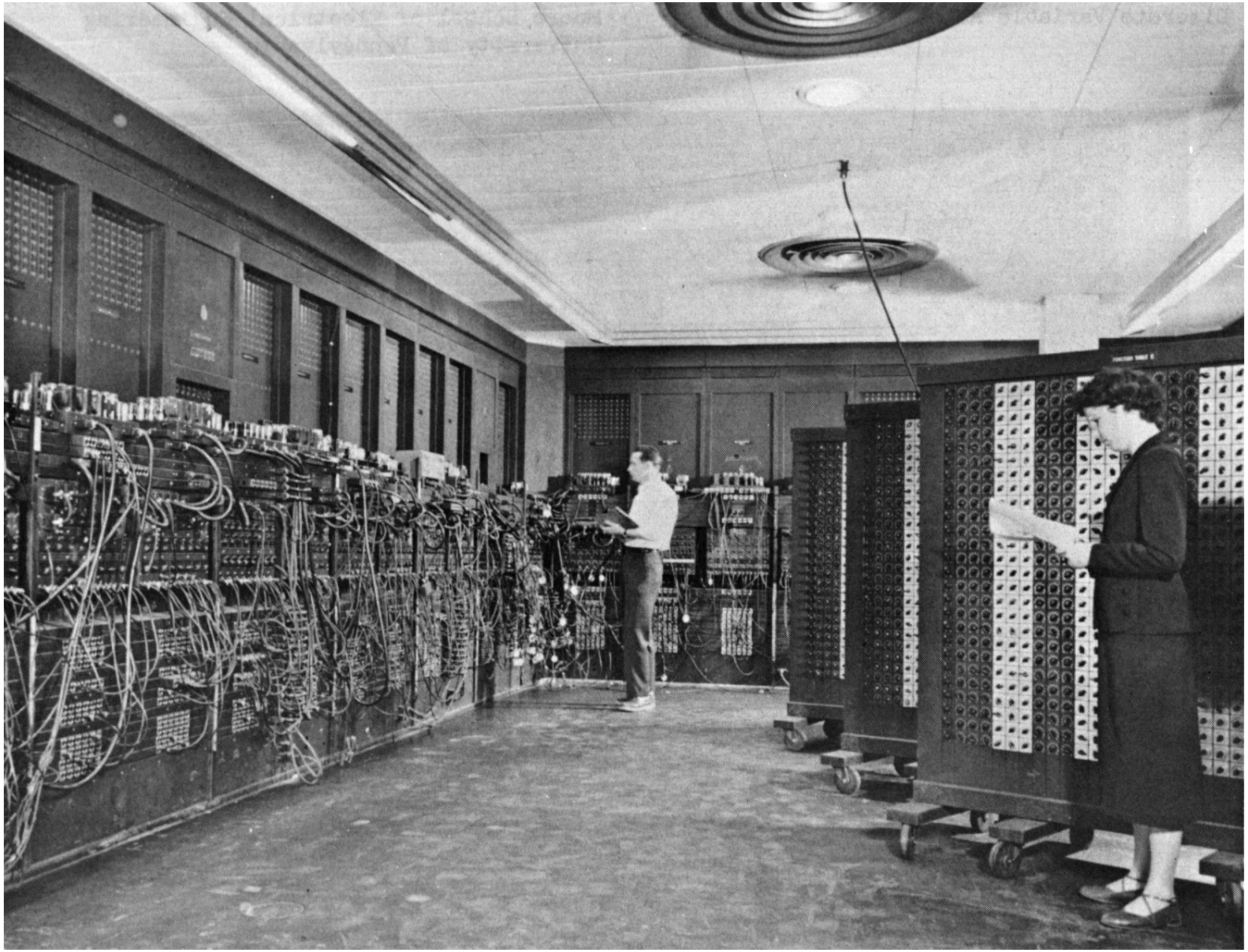
## Block-Diagram Algebra

Programming by patching is familiar to musicians :

# Programming by Composition

## Block-Diagram Algebra

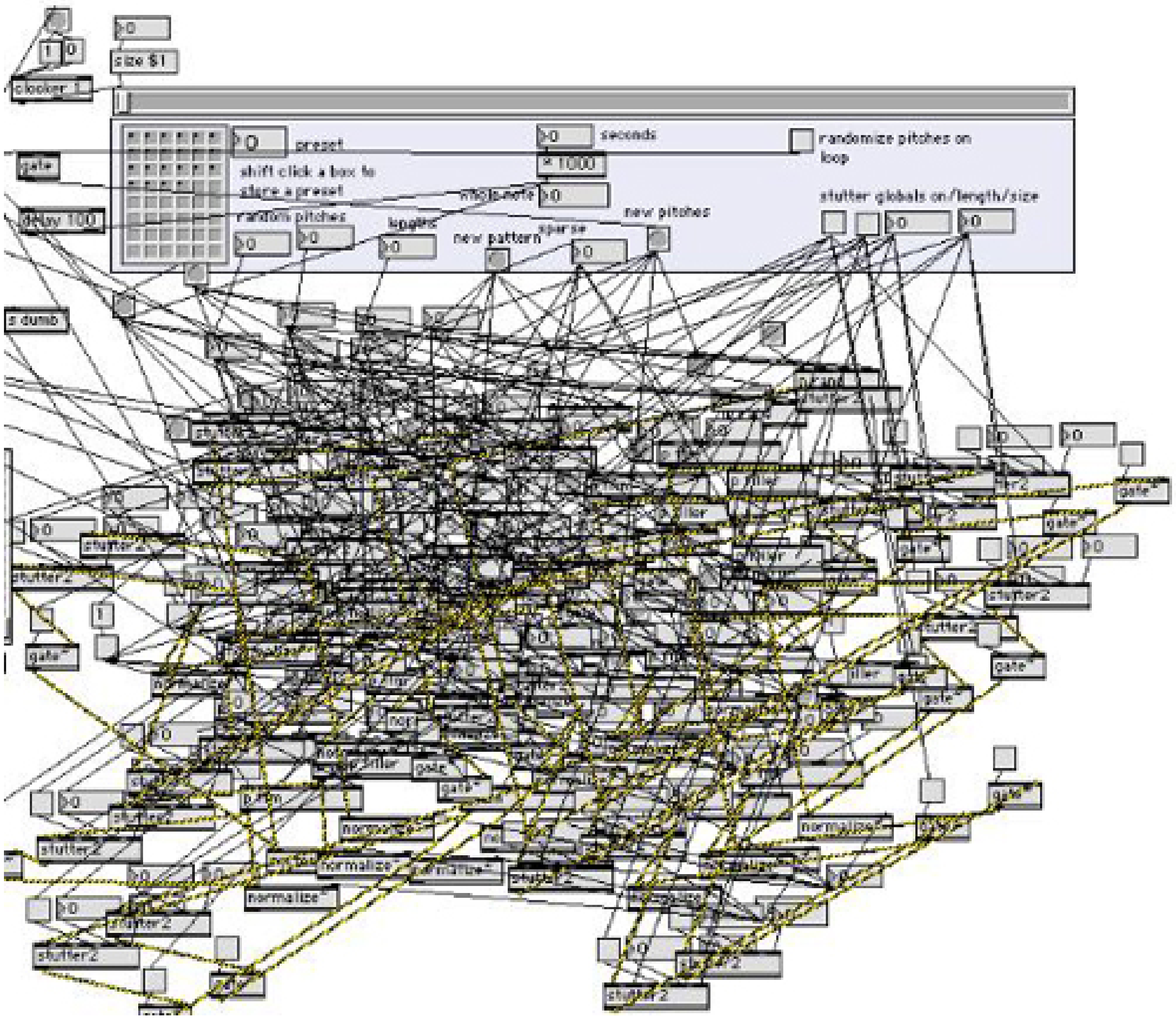
Programming by patching, the ENIAC computer :



# Programming by Composition

## Block-Diagram Algebra

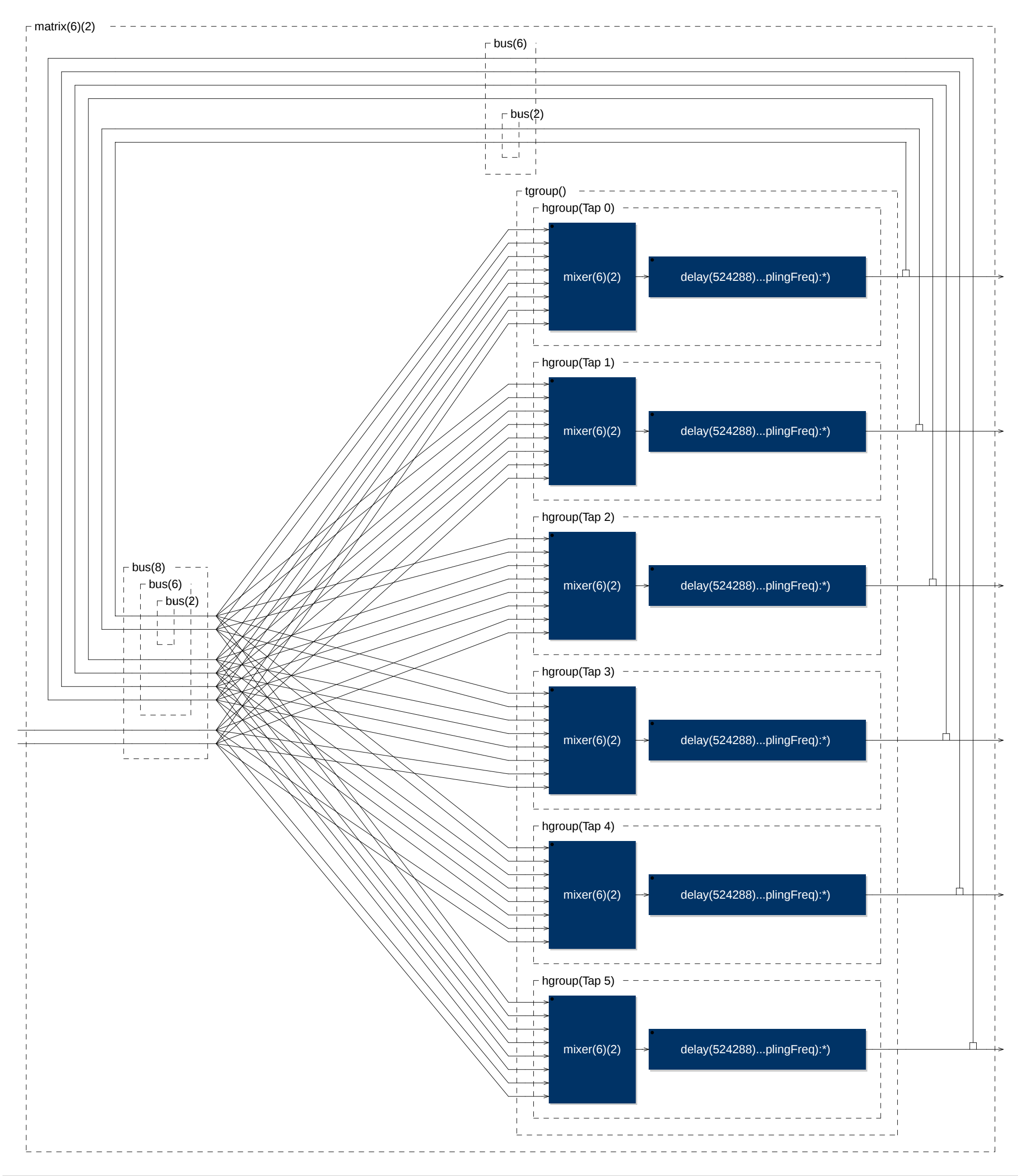
Block-diagrams are widely used in Visual Programming Languages like Max/MSP :



# Programming by Composition

## Block-Diagram Algebra

Faust allows structured block-diagrams



# Programming by Composition

## Some Primitive Signal Processors

- `!` : $\left| \begin{array}{l} \mathbb{S}^1 \rightarrow \mathbb{S}^0 \\ \lambda\langle x\rangle.\langle\rangle \end{array} \right.$ (cut)
- `_` : $\left| \begin{array}{l} \mathbb{S}^1 \rightarrow \mathbb{S}^1 \\ \lambda\langle x\rangle.\langle x\rangle \end{array} \right.$ (wire)
- `3` : $\left| \begin{array}{l} \mathbb{S}^0 \rightarrow \mathbb{S}^1 \\ \lambda\langle\rangle.\langle\lambda t. \begin{cases} 0 & t < 0 \\ 3 & t \geq 0 \end{cases}\rangle \end{array} \right.$ (number)
- `+` : $\left| \begin{array}{l} \mathbb{S}^2 \rightarrow \mathbb{S}^1 \\ \lambda\langle x, y\rangle.\langle\lambda t.x(t) + y(t)\rangle \end{array} \right.$ (addition)
- `@` : $\left| \begin{array}{l} \mathbb{S}^2 \rightarrow \mathbb{S}^1 \\ \lambda\langle x, y\rangle.\langle\lambda t.x(t - y(t))\rangle \end{array} \right.$ (delay)

# Programming by Composition

## Composition Operations

- `(A,B)` parallel composition
- `(A:B)` sequential composition
- `(A<:B)` split composition
- `(A:>B)` merge composition
- `(A~B)` recursive composition

# Programming by Composition

## 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.

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

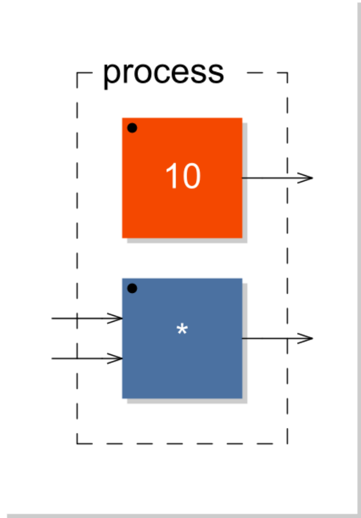



Figure: Example of parallel composition `(10,*)`

# Programming by Composition

## Sequential Composition

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

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

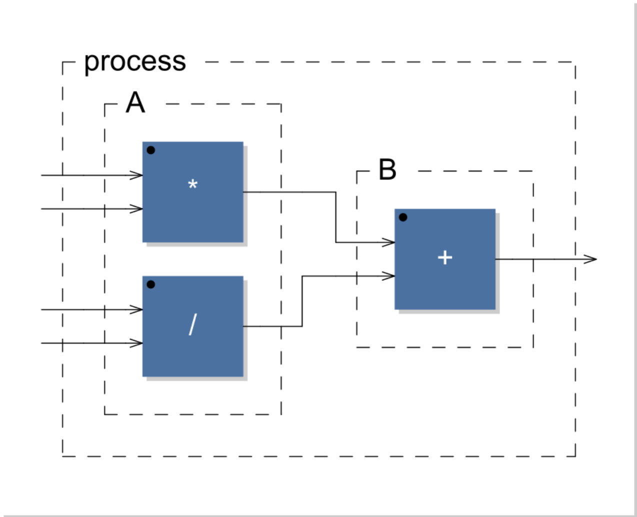



Figure: Example of sequential composition `((*,/):+)`

# Programming by Composition

## Split Composition

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

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

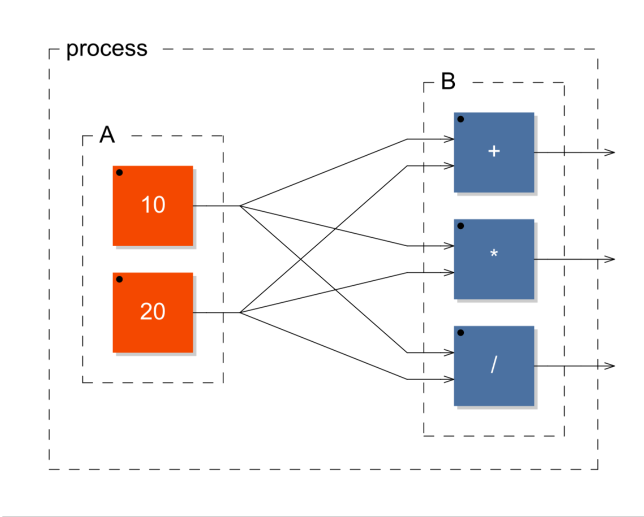



Figure: example of split composition `((10,20) <: (+,*,/))`

# Programming by Composition

## Merge Composition

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.

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

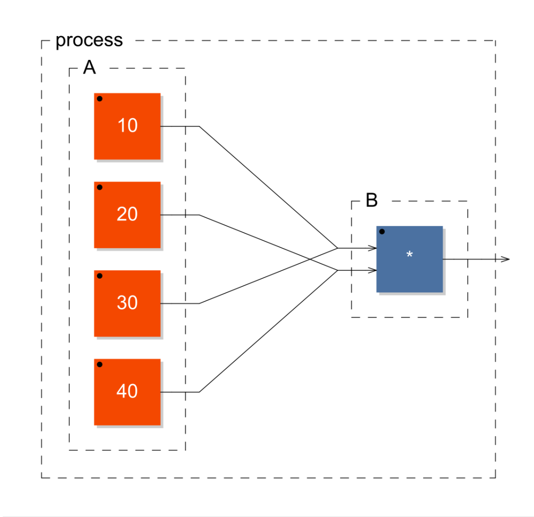



Figure: example of merge composition `((10,20,30,40) :> *)`

# Programming by Composition

## Recursive Composition

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

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

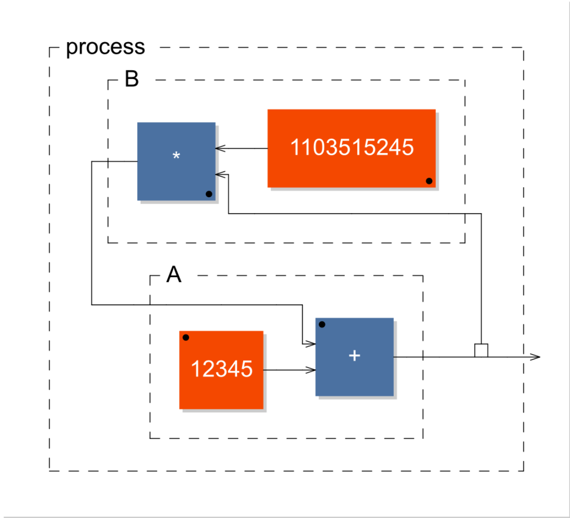



Figure: example of recursive composition `+(12345) ~ *(1103515245)`

# 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 <: +,*
```

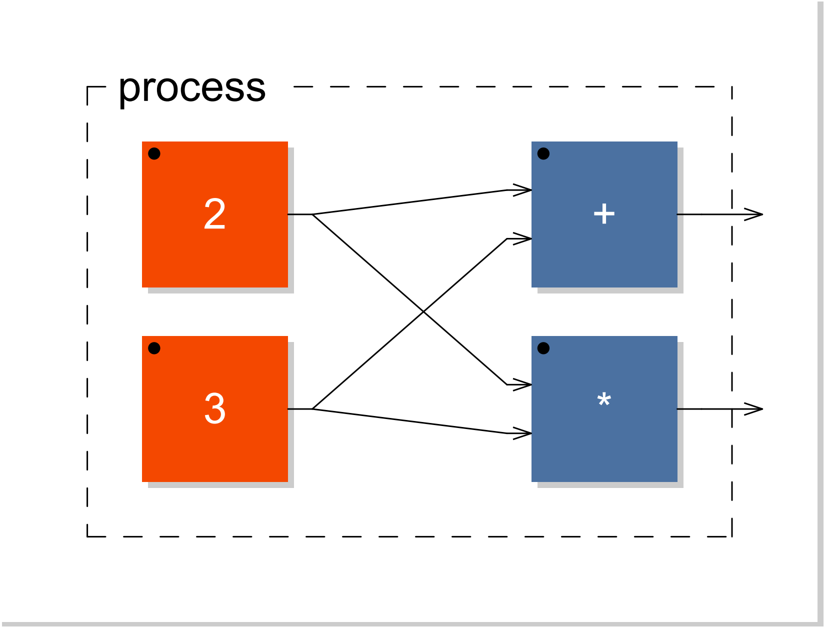


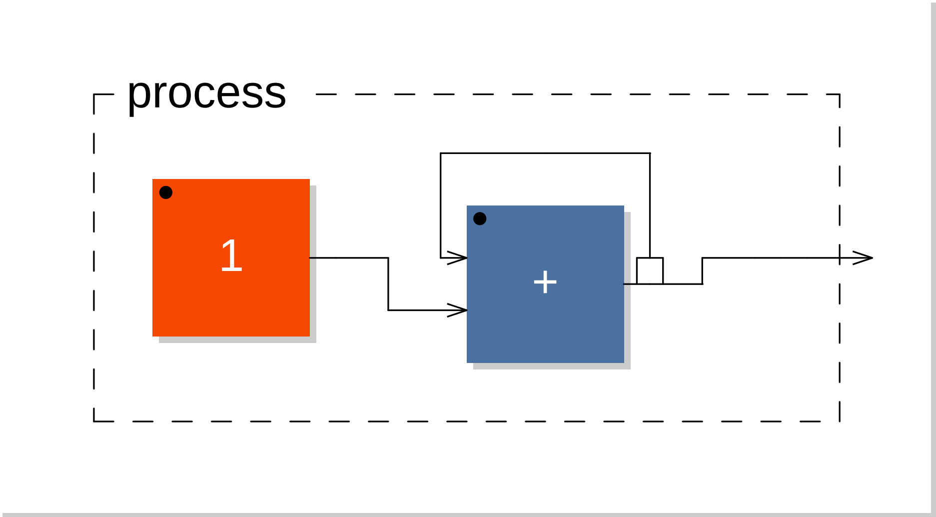

Figure: block-diagram of `2,3 <: +,*`

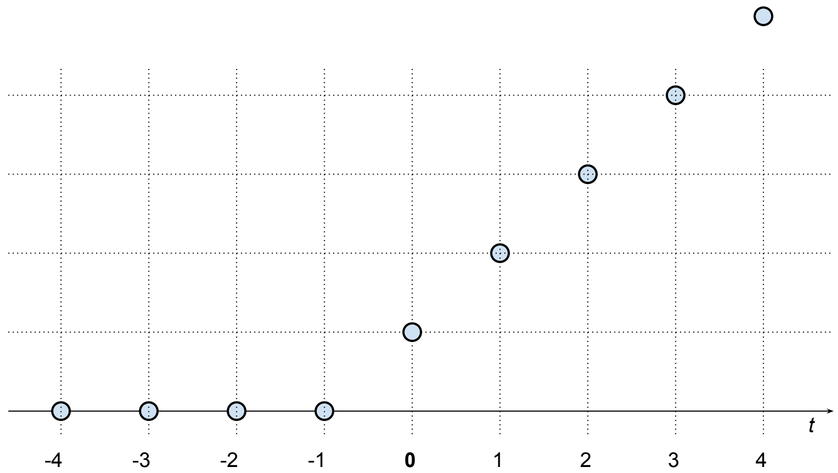
# Programming by Composition

## A Very Simple Example

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

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

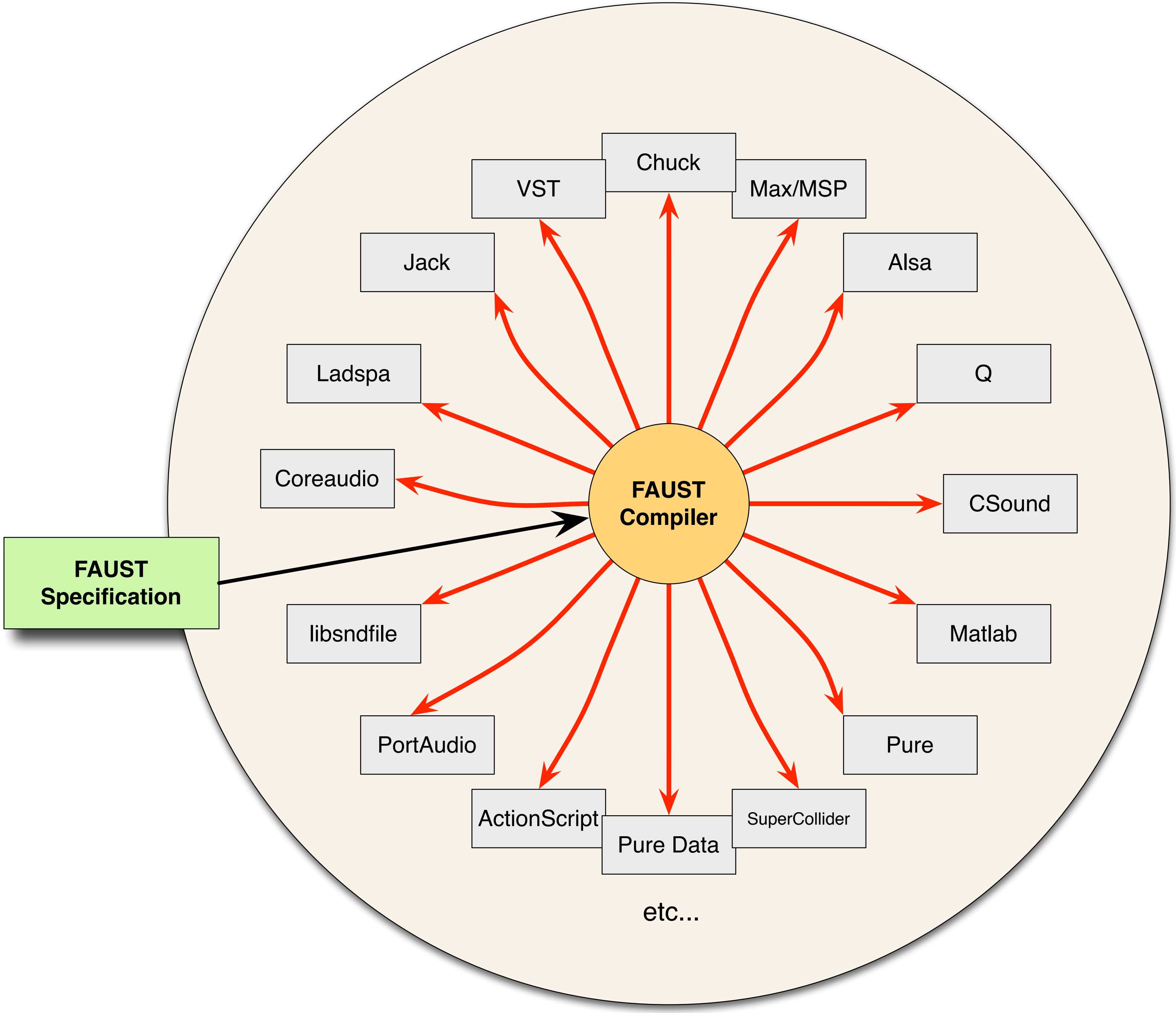


# 3-Easy Deployment

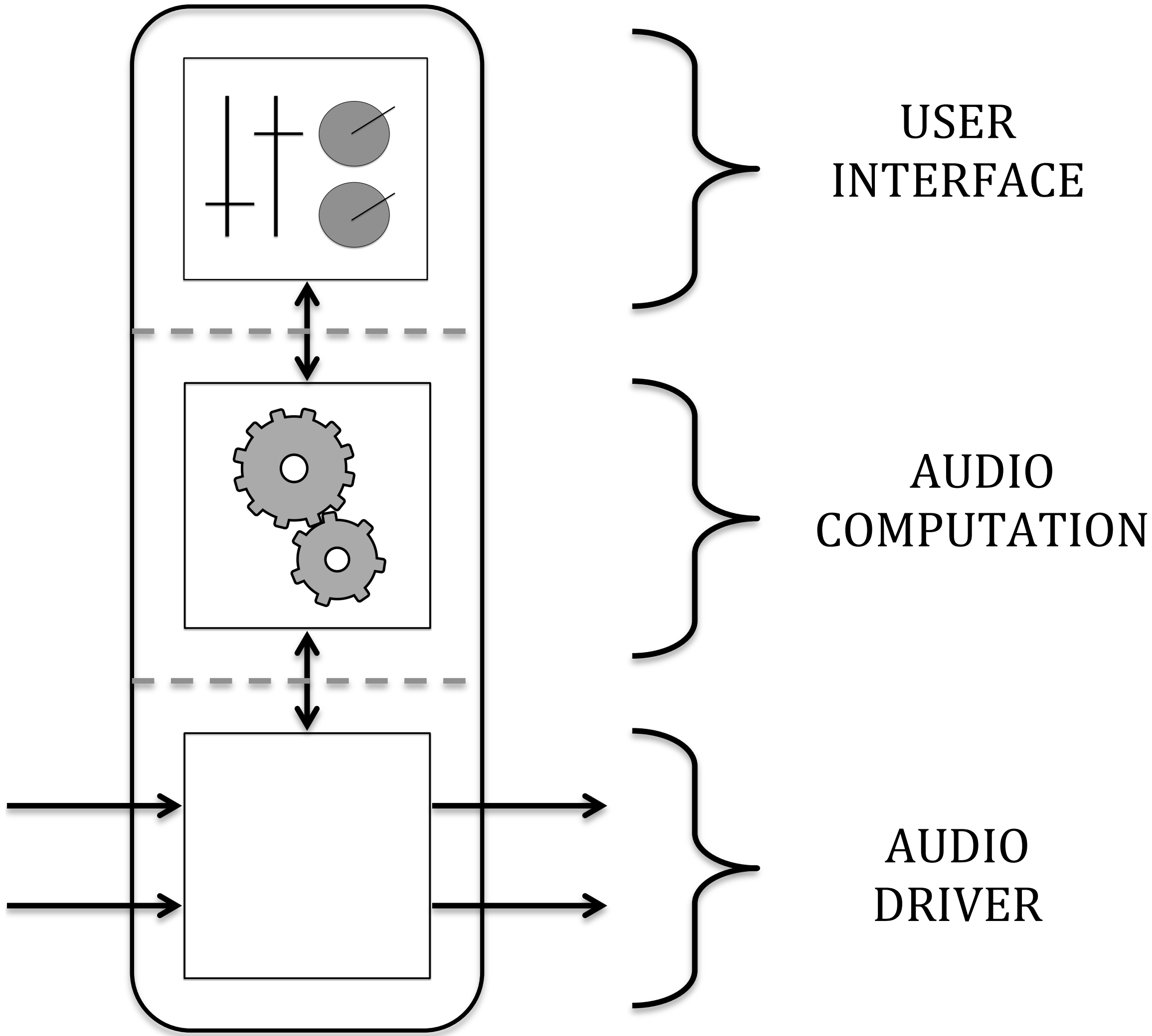
# Easy Deployment

## One Faust code, Multiple Targets



# Easy Deployment

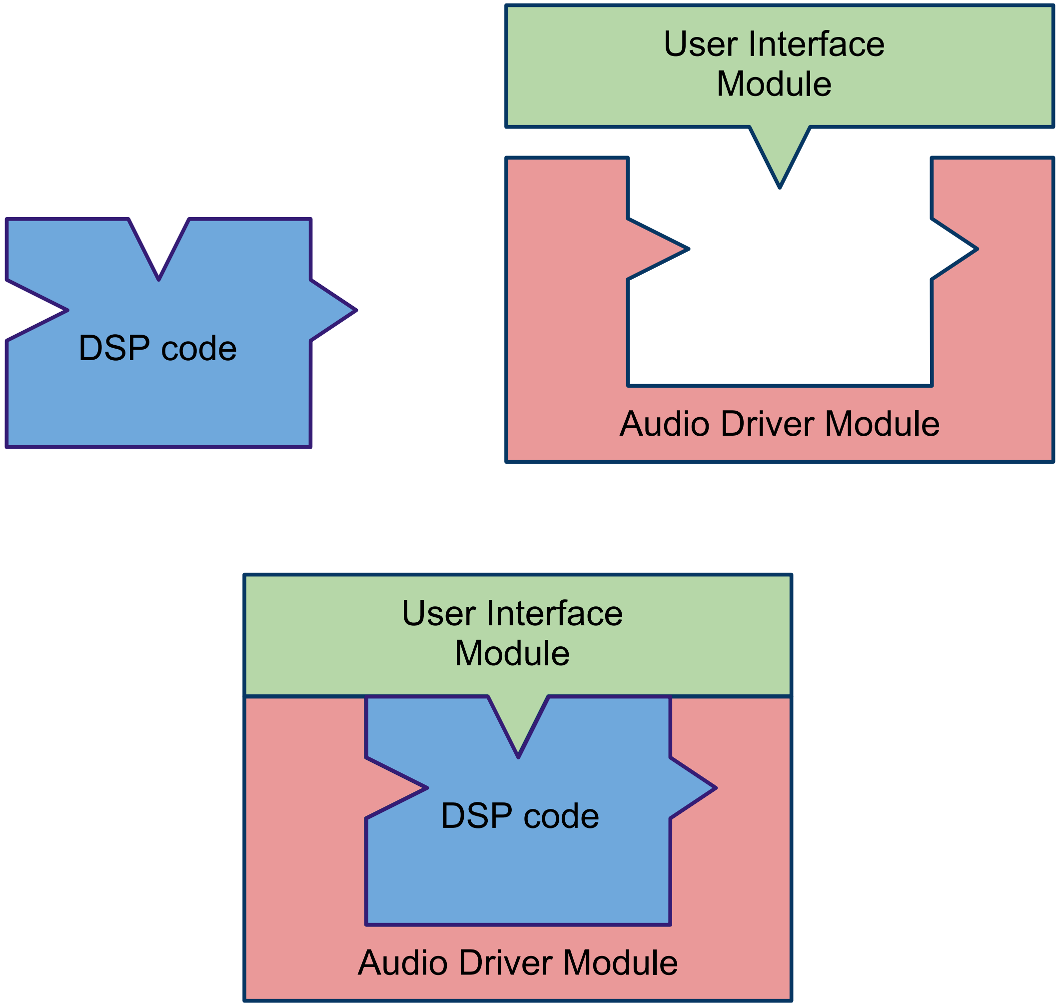
## Control/Compute/Communicate Model



# Easy Deployment

## Separation of concern

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



# Easy Deployment

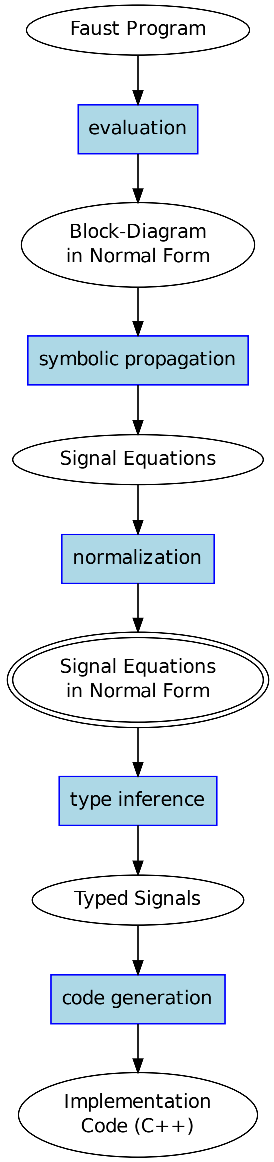
## Examples of supported architectures

- Audio plugins :
  - AudioUnit
  - LADSPA
  - DSSI
  - LV2
  - Max/MSP
  - VST
  - PD
  - CSound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash
- Audio drivers :
  - Jack
  - Alsa
  - CoreAudio
  - Web Audio API
- Graphic User Interfaces :
  - QT
  - GTK
  - Android
  - iOS
  - HTML5/SVG
- Other User Interfaces :
  - OSC
  - HTTPD

# 4-High Performances

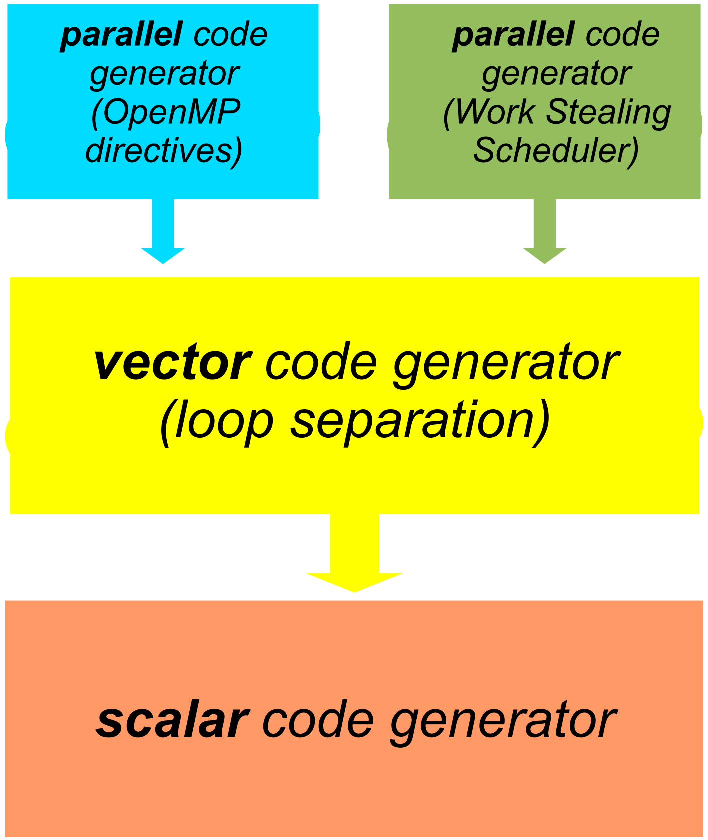
# High Performances

## Main Phases of the Faust Compiler



# High Performances

## Four Code Generation Modes



## STK vs FAUST (CPU load)

| File name | STK | FAUST | Difference |
|---|---|---|---|
| blowBottle.dsp | 3,23 | 2,49 | -22% |
| blowHole.dsp | 2,70 | 1,75 | -35% |
| bowed.dsp | 2,78 | 2,28 | -17% |
| brass.dsp | 10,15 | 2,01 | -80% |
| clarinet.dsp | 2,26 | 1,19 | -47% |
| flutestk.dsp | 2,16 | 1,13 | -47% |
| saxophony.dsp | 2,38 | 1,47 | -38% |
| sitar.dsp | 1,59 | 1,11 | -30% |
| tibetanBowl.dsp | 5,74 | 2,87 | -50% |

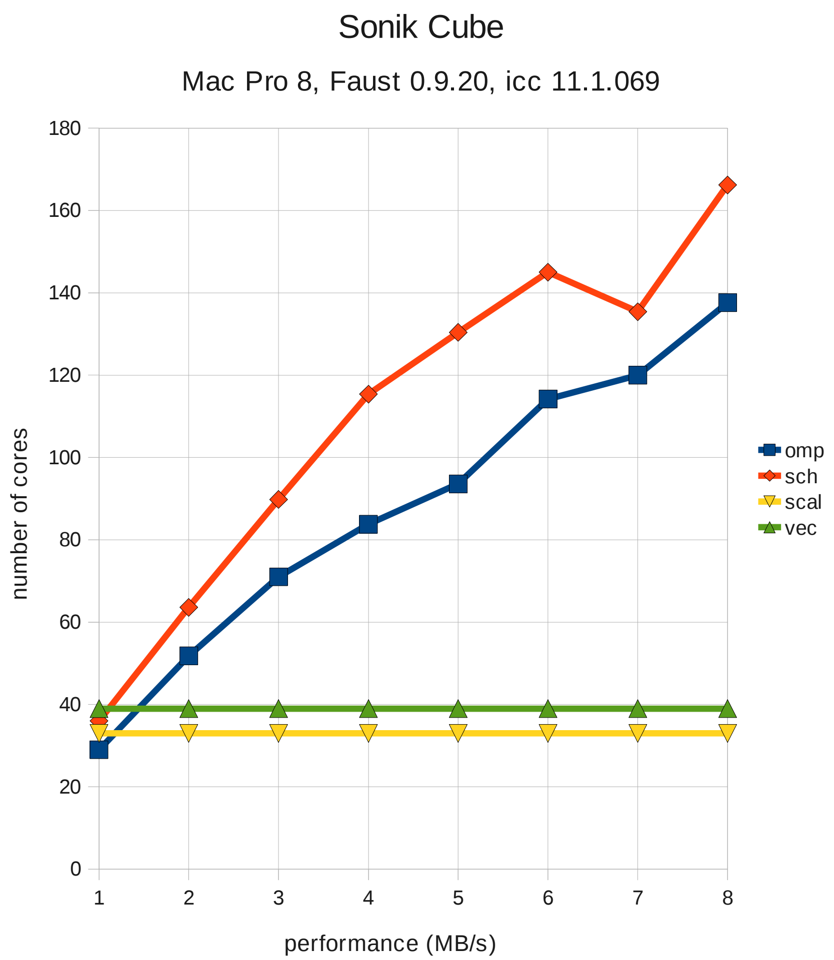
Overall improvement of about 41 % in favor of FAUST.

# Performance of the Generated Code

## Improvements to expect from Automatic Parallelized

### Sonik Cube

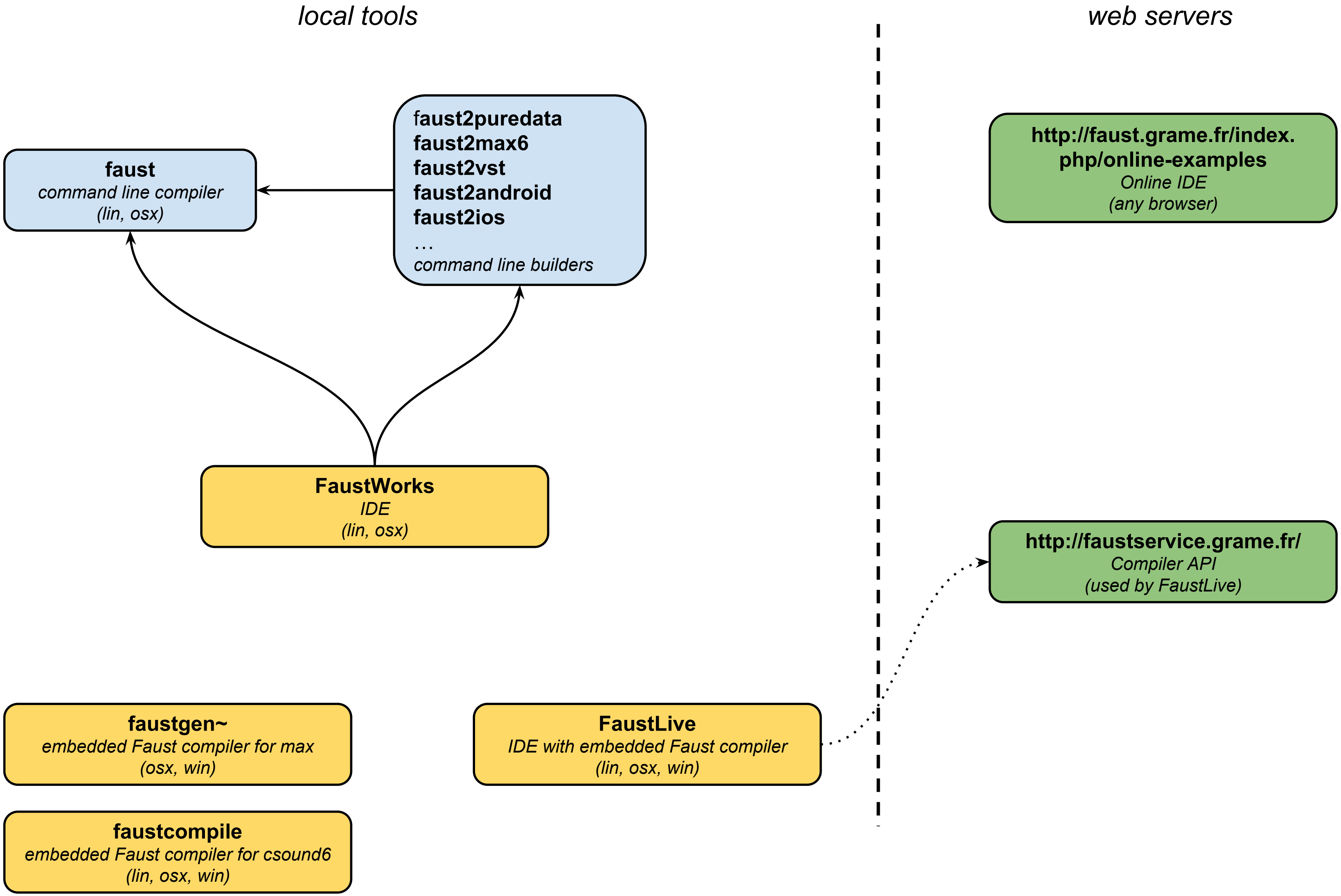
Compared performances of the various C++ code generation strategies according to the number of cores :



# 5-Rich Ecosystem

# Rich Ecosystem

## Overview



# Rich Ecosystem

## Compilers

### Several compilers are available

- Command Line Compilers
  - `faust` command line
  - `faust2xxx` command line
- Web Based Compilers
  - Online Compiler (http ://faust.grame.fr)
  - Faustweb API (http ://faustservice.grame.fr)
- Embedded Compilers (libfaust)
  - Faustgen for Max/MSP
  - Faustcompile, etc. for Csound (V. Lazzarini)
  - Faustnode for the Web Audio API
  - Antescofo (IRCAM's score follower)
  - iScore (LaBRI)
  - *your app...*
- IDE
  - FaustWorks (requires Faust)
  - FaustLive (self contained)

## Some useful libraries

- math.lib
- music.lib, imports math.lib
- hoa.lib, imports math.lib
- filter.lib, imports music.lib
- effect.lib, imports filter.lib
- oscillator.lib, imports filter.lib

# Rich Ecosystem

## Links

- Website and online compiler :
  - `http://faust.grame.fr`
- Faust distribution :
  - `http://sourceforge.net/projects/faudiostream/`
  - git clone git ://faudiostream.git.sourceforge.net/gitroot/faudiostream/faudiostream
  - cd faudiostream ; make ; sudo make install
- FaustWorks :
  - `http://sourceforge.net/projects/faudiostream/`
  - git clone git ://faudiostream.git.sourceforge.net/gitroot/faudiostream/FaustWorks
- FaustLive :
  - `http://sourceforge.net/projects/faudiostream/`
  - git clone git ://faudiostream.git.sourceforge.net/gitroot/faudiostream/faustlive

# 6-References

# References


- Orlarey, Fober, Letz 2004 : *Syntactical and Semantical Aspects of Faust* in Soft Computing 8(9), Springer-Verlag.
- Orlarey, Fober, Letz 2009 : *FAUST : an Efficient Functional Approach to DSP Programming* in New Computational Paradigms for Computer Music. Delatour.
- P Jouvelot, Y Orlarey 2011 : *Dependent Vector Types for Data Structuring in Multirate Faust* in Computer Languages, Systems & Structures, Elsevier.
- Denoux, Letz, Orlarey, Fober 2014 : *FaustLive : Just-In-Time Faust Compiler... and much more*. LAC 2014.
- FaustLive link : `https://dl.dropboxusercontent.com/u/1522100/FaustLive-OSX-2.36.dmg`