# Contents

**Introduction** **2**

- What is Faust? 2
- What is Faust Good For? 2
- What is Faust Not (So) Good For? 3
- Design Principles 3
- Signal Processor Semantic 4

**Quick Start** **5**

**Overview of the Faust Universe** **5**

- The Faust Distribution 5
  - Command-Line Compiler 6
  - `libfaust` 7
  - `faust2...` Scripts 7
- Web Tools 7
  - The Online Editor 7
  - The FaustPlayground 7
  - The Faust Online Compiler 7
  - Web Services 7
- Development Tools 7
  - FaustLive 7
  - FaustWorks 7

**Compiling and Installing Faust** **7**

- Getting the Source Code 7

**Faust Syntax** **8**

- Faust Program 8
- Statements 8
  - Metadata Declarations 9
  - Imports 9
  - Documentation Tags 9
- Definitions 10
  - Simple Definitions 10
- Function Definitions 11
- Definitions with pattern matching 11
- Expressions 12
  - Diagram Expressions 12
  - Infix Notation and Other Syntax Extensions 21
  - `'` Time Expression 22
  - `@` Time Expression 23
  - Environment Expressions 23
  - Foreign Expressions 26
  - Applications and Abstractions 27

Primitives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

**Using the Faust Compiler** **27**

**A Quick Tour of the Faust Targets** **27**

**Mathematical Documentation** **27**

# Introduction

## What is Faust?

Faust (Functional Audio Stream) is a functional programming language for sound synthesis and audio processing with a strong focus on the design of synthesizers, musical instruments, audio effects, etc. Faust targets high-performance signal processing applications and audio plug-ins for a variety of platforms and standards. It is used on stage for concerts and artistic productions, in education and research, in open source projects as well as in commercial applications.

The core component of Faust is its compiler. It allows to "translate" any Faust digital signal processing (DSP) specification to a wide range of non-domain specific languages such as C++, C, JAVA, JavaScript, LLVM bit code, WebAssembly, etc. In this regard, Faust can be seen as an alternative to C++ but is much simpler and intuitive to learn.

Thanks to a wrapping system called "architectures," codes generated by Faust can be easily compiled into a wide variety of objects ranging from audio plug-ins to standalone applications or smartphone and web apps, etc. (check the Quick Tour of the Faust Targets section for an exhaustive list.

This manual gives an overview of the Faust programming language and of its features through various interactive examples.

## What is Faust Good For?

Faust's syntax allows to express any DSP algorithm as a block diagram. For example, `+` is considered as a valid function (and block) taking two arguments (signals) and returning one:

```
process = +;
```

Blocks can be easily connected together using the `:` "connection" composition:

```
process = + : *(0.5);
```

In that case, we add two signals together and then scale the result of this operation.

Thus, **Faust is perfect to implement time-domain algorithms that can be easily represented as block diagrams** such as filters, waveguide physical models, virtual analog elements, etc.

**Faust is very concise**, for example, here's the implementation of a one pole filter/integrator equivalent to $y(n) = x(x) + a_1 y(n-1)$ (where $a_1$ is the pole):

```
a1 = 0.9;
process = +~*(a1);
```

**Codes generated by Faust are extremely optimized** and usually more efficient that handwritten codes (at least for C and C++). The Faust compiler tries to optimize each element of an algorithm. For example, you shouldn't have to worry about using divides instead of multiplies as they get automatically replaced by multiplies by the compiler when possible, etc.

**Faust is very generic** and allows to write code that will run on dozens of platforms.

## What is Faust Not (So) Good For?

Despite all this, Faust does have some limitations. For instance, it doesn't allow for the efficient implementation of algorithms requiring multi-rates such as the FFT, convolution, etc. While there are tricks to go around this issue, we're fully aware that it is a big one and we're working as hard as possible on it.

Faust's conciseness can sometimes become a problem too, especially for complex algorithms with lots of recursive signals. It is usually crucial in Faust to have the "mental global picture" of the algorithm to be implemented which in some cases can be hard.

While the Faust compiler is relatively bug-free, it does have some limitations and might get stuck in some extreme cases that you will probably never encounter. If you do, shoot us an e-mail!

From here, you can jump to … if you wanna get your hands dirty, etc. TODO.

## Design Principles

Since the beginning of its development in 2002, Faust has been guided by various design principles:

- Faust is a *specification language*. It aims at providing an adequate notation to describe *signal processors* from a mathematical point of view. Faust is, as much as possible, free from implementation details.
- Faust programs are fully compiled (i.e., not interpreted). The compiler translates Faust programs into equivalent programs in other languages (e.g., JAVA, JavaScript, LLVM bit code, WebAssembly, etc.) taking care

of generating the most efficient code. The result can generally compete with, and sometimes even outperform, C++ code written by seasoned programmers.

- The generated code works at the sample level. It is therefore suited to implement low-level DSP functions like recursive filters. Moreover the code can be easily embedded. It is self-contained and doesn't depend of any DSP library or runtime system. It has a very deterministic behavior and a constant memory footprint.
- The semantic of Faust is simple and well defined. This is not just of academic interest. It allows the Faust compiler to be *semantically driven*. Instead of compiling a program literally, it compiles the mathematical function it denotes. This feature is useful for example to promote components reuse while preserving optimal performance.

- Faust is a textual language but nevertheless block-diagram oriented. It actually combines two approaches: *functional programming* and *algebraic block-diagrams*. The key idea is to view block-diagram construction as function composition. For that purpose, Faust relies on a *block-diagram algebra* of five composition operations: `:` `,` `~` `<:` `:>` (see the section on Diagram Composition Operations for more details).
- Thanks to the concept of *architecture*, Faust programs can be easily deployed on a large variety of audio platforms and plugin formats without any change to the Faust code.

## Signal Processor Semantic

A Faust program describes a *signal processor*. The role of a *signal processor* is to transforms a (possibly empty) group of *input signals* in order to produce a (possibly empty) group of *output signals*. Most audio equipments can be modeled as *signal processors*. They have audio inputs, audio outputs as well as control signals interfaced with sliders, knobs, vu-meters, etc.

More precisely :

- A *signal* $s$ is a discrete function of time $s : \mathbb{Z} \rightarrow \mathbb{R}$. The value of a signal $s$ at time $t$ is written $s(t)$. The values of signals are usually needed starting from time 0. But to take into account *delay operations*, negative times are possible and are always mapped to zeros. Therefore for any Faust signal $s$ we have $\forall t < 0, s(t) = 0$. In operational terms this corresponds to assuming that all delay lines are signals initialized with 0s.
- Faust considers two type of signals: *integer signals* ($s : \mathbb{Z} \rightarrow \mathbb{Z}$) and *floating point signals* ($s : \mathbb{Z} \rightarrow \mathbb{Q}$). Exchanges with the outside world are, by convention, made using floating point signals. The full range is represented by sample values between $-1.0$ and $+1.0$.
- The set of all possible signals is $\mathbb{S} = \mathbb{Z} \rightarrow \mathbb{R}$.

- A group of $n$ signals (a $n$-tuple of signals) is written $(s_1, \dots, s_n) \in \mathbb{S}^n$. The *empty tuple*, single element of $\mathbb{S}^0$ is notated $()$.
- A *signal processors* $p$, is a function from $n$-tuples of signals to $m$-tuples of signals $p : \mathbb{S}^n \to \mathbb{S}^m$. The set $\mathbb{P} = \bigcup_{n,m} \mathbb{S}^n \to \mathbb{S}^m$ is the set of all possible signal processors.

As an example, let's express the semantic of the Faust primitive `+`. Like any Faust expression, it is a signal processor. Its signature is $\mathbb{S}^2 \to \mathbb{S}$. It takes two input signals $X_0$ and $X_1$ and produces an output signal $Y$ such that $Y(t) = X_0(t) + X_1(t)$.

Numbers are signal processors too. For example the number 3 has signature $\mathbb{S}^0 \to \mathbb{S}$. It takes no input signals and produce an output signal $Y$ such that $Y(t) = 3$.

# Quick Start

TODO Will be all based on the online editor… May be could be a simple copy and paste of the session 1 of the Kadenze course…

# Overview of the Faust Universe

While in its most *primitive* form, Faust is distributed as a command-line compiler, a wide range of tools have been developed around it in the course of the past few years. Their variety and their function might be hard to grab at first. This sort chapter provides an overview of their role and will hopefully help you decide which one is better suited for your personal use.

TODO: here say a few words about the philosophy behind the disto: the online editor is the way to go for most users, then various pre-compiled packages of the compiler can be found, then source, then git. Finally other external tools for development.

## The Faust Distribution

The Faust distribution hosts the source of the Faust compiler (both in its command line and library version), the source of the Faust *architectures* (targets), the various Faust compilation scripts, a wide range of Faust-related-tools, the Faust DSP Libraries (which in practice are hosted a separate Git submodule), etc.

The latest stable release of the Faust distribution can be found here: https://github.com/grame-cncm/faust/releases. It is recommended for most Faust users willing to compile the Faust compiler and `libfaust` from scratch.

To have the latest stable development version, you can use the `master` branch of the Faust git repository which is hosted on GitHub: https://github.com/grame-cncm/faust/tree/master.

For something even more bleeding edge (to be used at your own risks), you might use the `master-dev` branch of the Faust git repository: https://github.com/grame-cncm/faust/tree/master-dev. `master-dev` is the development sub-branch of `master`. It is used by Faust developers to commit their changes and can be considered as "the main development branch." The goal is to make sure that `master` is always functional. Merges between `master-dev` and `master` are carried out multiple times a week by the GRAME team.

> Also, note that pre-compiled packages of the Faust compiler and of `libfaust` for various platforms can be found on the Download Page of the Faust website.

The Faust distribution is organized as follows:

```
architecture/          : the source of the architecture files
benchmark/             : tools to measure the efficiency of the generated code
build/                 : the various makefiles and build folders
compiler/              : sources of the Faust compiler
COPYING                : license information
debian/                : files for Debian installation
Dockerfile             : docker file
documentation/         : Faust's documentations
examples/              : Faust programs examples organized by categories
installer/             : various installers for Linux distribution
libraries/             : Faust DSP libraries
Makefile               : makefile used to build and install Faust
README.md              : instructions on how to build and install Faust
syntax-highlighting/   : support for syntax highlighting for several editors
tests/                 : various tests
tools/                 : tools to produce audio applications and plugins
windows/               : Windows related ressources
```

The following subsections present some of the main components of the Faust distribution.

### Command-Line Compiler

- Link to precompiled version versions (download page)
- What is the Faust compiler? (Quickly)
- Link to Using the Faust Compiler

#### `libfaust`

- Link to precompiled version versions (download page)
- What is it? (Quickly)
- Link to tutorial Embedding the Faust Compiler Using `libfaust`

#### `faust2...` Scripts

## Web Tools

### The Online Editor

### The FaustPlayground

### The Faust Online Compiler

### Web Services

## Development Tools

### FaustLive

### FaustWorks

# Compiling and Installing Faust

This chapter describes how to get and compile the Faust compiler as well as other tools related to Faust (e.g., `libfaust`, `libosc`, `libhttpd`, etc.).

## Getting the Source Code

An overview of the various places where the Faust source can be downloaded is given here.

If you downloaded the latest Faust release, just un-compressed the archive file and open it in a terminal. For instance, something like (this might vary depending on the version of Faust you downloaded):

```
tar xzf faust-2.5.31.tar.gz
cd faust-2.5.31
```

If you wish to get the Faust source directly from the git repository, just run:

```
git clone --recursive https://github.com/grame-cncm/faust.git
cd faust
```

in a terminal. Note that the `--recursive` option is necessary here since some elements (e.g., the Faust DSP libraries) are placed in other repositories.

Finally, if you wish to use the development (and potentially unstable) branch, just run:

```
git checkout master-dev
```

after the previous 2 commands.

TODO: see with Dominique for whatever comes next here…

Since release 2.5.18, Faust compilation and installation is based on `cmake`.

# Faust Syntax

## Faust Program

A Faust program is essentially a list of *statements*. These statements can be *metadata declarations*, *imports*, *definitions*, and *documentation tags*, with optional C++ style (`//...` and `/*...*/`) comments.

Here is a short Faust program that implements of a simple noise generator (called from the `noises.lib` Faust library). It exhibits various kind of statements : two *metadata declarations*, an *imports*, a *comment*, and a *definition*. We will study later how *documentation statements* work:

```
declare name  "Noise";
declare copyright "(c)GRAME 2018";

import("stdfaust.lib");

// noise level controlled by a slider
process = no.noise * hslider("gain",0,0,1, 0.1);
```

The keyword `process` is the equivalent of `main` in C/C++. Any Faust program, to be valid, must at least define `process`.

## Statements

The *statements* of a Faust program are of four kinds:

- *metadata declarations*,
- *file imports*,
- *definitions*,
- *documentation*.

All statements but *documentation* end with a semicolon `;`.

### Metadata Declarations

All metadata declaration in Faust start with `declare`.

When used in the context of Faust program (e.g., `.dsp` file), they are followed by a key and a string. For example:

```
declare name "Noise";
```

allows us to specify the name of a Faust program in its whole.

When used in the context of a library (e.g., `.lib` file), metadata declarations can either be “global” (as in the previous example), or associated to a specific function. In that case, `declare` will be followed by the name of the function, a key, and a string. For example:

```
declare add author "John Doe"
add = +;
```

This is very useful when a library has several contributors and that functions potentially have different license terms.

Unlike regular comments, metadata declarations will appear in the C++ code generated by the Faust compiler. A good practice is to start a Faust program with some standard declarations:

```
declare name "MyProgram";
declare author "MySelf";
declare copyright "MyCompany";
declare version "1.00";
declare license "BSD";
```

### Imports

File imports allow us to import definitions from other source files.

For example `import("maths.lib");` imports the definitions of the `maths.lib` library.

The most common file to be imported is the `stdfaust.lib` library which gives access to all the standard Faust libraries from a single point:

```
import("stdfaust.lib");
process = os.osc(440); // the "hello world" of computer music
```

### Documentation Tags

Documentation statements are optional and typically used to control the generation of the mathematical documentation of a Faust program. This documentation system is detailed in the Mathematical Documentation chapter. In this section we essentially describe the documentation statements syntax.

A documentation statement starts with an opening `<mdoc>` tag and ends with a closing `</mdoc>` tag. Free text content, typically in Latex format, can be placed in between these two tags.

Moreover, optional sub-tags can be inserted in the text content itself to require the generation, at the insertion point, of mathematical *equations*, graphical *block-diagrams*, Faust source code *listing* and explanation *notice*.

The generation of the mathematical equations of a Faust expression can be requested by placing this expression between an opening `<equation>` and a closing `</equation>` tag. The expression is evaluated within the lexical context of the Faust program.

Similarly, the generation of the graphical block-diagram of a Faust expression can be requested by placing this expression between an opening `<diagram>` and a closing `</diagram>` tag. The expression is evaluated within the lexical context of the Faust program.

The `<metadata>` tags allow to reference Faust metadata declarations, calling the corresponding keyword.

The `<notice/>` empty-element tag is used to generate the conventions used in the mathematical equations.

The `<listing/>` empty-element tag is used to generate the listing of the Faust program. Its three attributes `mdoctags`, `dependencies`, and `distributed` enable or disable respectively `<mdoc>` tags, other files dependencies and distribution of interleaved Faust code between `<mdoc>` sections.

## Definitions

A *definition* associates an identifier with an expression. Definitions are essentially a convenient shortcut avoiding to type long expressions. During compilation, more precisely during the evaluation stage, identifiers are replaced by their definitions. It is therefore always equivalent to use an identifier or directly its definition. Please note that multiple definitions of a same identifier are not allowed, unless it is a pattern matching based definition.

### Simple Definitions

The syntax of a simple definition is:

```
identifier = expression ;
```

For example here is the definition of `random`, a simple pseudo-random number generator:

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

## Function Definitions

Definitions with formal parameters correspond to functions definitions.

For example the definition of `linear2db`, a function that converts linear values to decibels, is:

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

Please note that this notation is only a convenient alternative to the direct use of *lambda-abstractions* (also called anonymous functions). The following is an equivalent definition of `linear2db` using a lambda-abstraction:

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

## Definitions with pattern matching

Moreover, formal parameters can also be full expressions representing patterns.

This powerful mechanism allows to algorithmically create and manipulate block diagrams expressions. Let's say that you want to describe a function to duplicate an expression several times in parallel:

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

Note that this last definition is a convenient alternative to the more verbose:

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

A use case for `duplicate` could be to put 5 white noise generators in parallel:

```
import("stdfaust.lib");
duplicate(1,x) = x;
duplicate(n,x) = x, duplicate(n-1,x);
process = duplicate(5,no.noise);
```

Here is another example to count the number of elements of a list. Please note that we simulate lists using parallel composition: `(1,2,3,5,7,11)`. The main limitation of this approach is that there is no empty list. Moreover lists of only one element are represented by this element:

```
count((x,xs)) = 1+count(xs);
count(x) = 1;
```

If we now write `count(duplicate(10,666))`, the expression will be evaluated as `10`.

Note that the order of pattern matching rules matters. The more specific rules must precede the more general rules. When this order is not respected, as in:

```
count(x) = 1;
count((x,xs)) = 1+count(xs);
```

the first rule will always match and the second rule will never be called.

## Expressions

Despite its textual syntax, Faust is conceptually a block-diagram language. Faust expressions represent DSP block-diagrams and are assembled from primitive ones using various *composition* operations. More traditional *numerical* expressions in infix notation are also possible. Additionally Faust provides time based expressions, like delays, expressions related to lexical environments, expressions to interface with foreign function and lambda expressions.

### Diagram Expressions

Diagram expressions are assembled from primitive ones using either binary composition operations or high level iterative constructions.

#### Diagram Composition Operations

Five binary *composition operations* are available to combine block-diagrams:

- *recursion* (`~`),
- *parallel* (`,`),
- *sequential* (`:`),
- *split* (`<:`),
- *merge* (`:>`).

One can think of each of these composition operations as a particular way to connect two block diagrams.

To describe precisely how these connections are done, we have to introduce some notation. The number of inputs and outputs of a block-diagram $A$ are expressed as $\mathrm{inputs}(A)$ and $\mathrm{outputs}(A)$. The inputs and outputs themselves are respectively expressed as: $[0]A$, $[1]A$, $[2]A$, $\ldots$ and $A[0]$, $A[1]$, $A[2]$, etc.

For each composition operation between two block-diagrams $A$ and $B$ we will describe the connections $A[i] \to [j]B$ that are created and the constraints on their relative numbers of inputs and outputs.

The priority and associativity of this five operations are:

| Syntax | Priority | Association | Description |
|---|---|---|---|
| `expression ~ expression` | 4 | left | Recursive Composition |
| `expression , expression` | 3 | right | Parallel Composition |
| `expression : expression` | 2 | right | Sequential Composition |
| `expression <: expression` | 1 | right | Split Composition |
| `expression :> expression` | 1 | right | Merge Composition |

### Parallel Composition

The *parallel composition* (e.g., `(A,B)`) is probably the simplest one. It places the two block-diagrams one on top of the other, without connections. The inputs of the resulting block-diagram are the inputs of `A` and `B`. The outputs of the resulting block-diagram are the outputs of `A` and `B`.

*Parallel composition* is an associative operation: `(A,(B,C))` and `((A,B),C)` are equivalents. When no parenthesis are used (e.g., `A,B,C,D`), Faust uses right associativity and therefore builds internally the expression `(A,(B,(C,D)))`. This organization is important to know when using pattern matching techniques on parallel compositions.

**Example: Oscillators in Parallel**

*Parallel composition* can be used to put 3 oscillators of different kinds and frequencies in parallel, which will result in a Faust program with 3 outputs:

```
import("stdfaust.lib");
process = os.osc(440),os.sawtooth(550),os.triangle(660);
```

**Example: Stereo Effect**

*Parallel composition* can be used to easily turn a mono effect into a stereo one which will result in a Faust program with 2 inputs and 2 outputs:

```
import("stdfaust.lib");
level = 1;
process = ve.autowah(level),ve.autowah(level);
```

Note that there's a better to write this last example using the `par` iteration:

```
import("stdfaust.lib");
level = 1;
process = par(i,2,ve.autowah(level));
```

### Sequential Composition

The *sequential composition* (e.g., `A:B`) expects:

$$\text{outputs}(A) = \text{inputs}(B)$$

It connects each output of $A$ to the corresponding input of $B$:

$$A[i] \to [i]B$$

*Sequential composition* is an associative operation: `(A:(B:C))` and `((A:B):C)` are equivalents. When no parenthesis are used, like in `A:B:C:D`, Faust uses right associativity and therefore builds internally the expression `(A:(B:(C:D)))`.

**Example: Sine Oscillator**

Since everything is considered as a signal generator in Faust, *sequential composition* can be simply used to pass an argument to a function:

```
import("stdfaust.lib");
process = 440 : os.osc;
```

**Example: Effect Chain**

*Sequential composition* can be used to create an audio effect chain. Here we're plugging a guitar distortion to an autowah:

```
import("stdfaust.lib");
drive = 0.6;
offset = 0;
autoWahLevel = 1;
process = ef.cubicnl(drive,offset) : ve.autowah(autoWahLevel);
```

### Split Composition

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

For the operation to be valid, the number of inputs of $B$ must be a multiple of the number of outputs of $A$:

$$\text{outputs}(A).k = \text{inputs}(B)$$

Each input $i$ of $B$ is connected to the output $i \bmod k$ of $A$:

$$A[i \bmod k] \to [i]B$$

**Example: Duplicating the Output of an Oscillator**

*Split composition* can be used to duplicate signals. For example, the output of the following sawtooth oscillator is duplicated 3 times in parallel.

```
import("stdfaust.lib");
process = os.sawtooth(440) <: _,_,_;
```

Note that this can be written in a more effective way by replacing `_,_,_` with `par(i,3,_)` using the `par` iteration.

**Example: Connecting a Mono Effect to a Stereo One**

More generally, the *split composition* can be used to connect a block with a certain number of output to a block with a greater number of inputs:

```
import("stdfaust.lib");
drive = 0.6;
offset = 0;
process = ef.cubicnl(drive,offset) <: dm.zita_light;
```

Note that an arbitrary number of signals can be split, for example:

```
import("stdfaust.lib");
drive = 0.6;
offset = 0;
process = par(i,2,ef.cubicnl(drive,offset)) <: par(i,2,dm.zita_light);
```

Once again, the only rule with this is that in the expression `A<:B` the number of inputs of `B` has to be a multiple of the number of outputs of `A`.

### Merge Composition

The *merge composition* (e.g., `A:>B`) is the dual of the *split composition*. The number of outputs of $A$ must be a multiple of the number of inputs of $B$:

$$\text{outputs}(A) = k.\text{inputs}(B)$$

Each output $i$ of $A$ is connected to the input $i \bmod k$ of $B$ :

$$A[i] \rightarrow\ [i \bmod k]B$$

The $k$ incoming signals of an input of $B$ are summed together.

**Example: Summing Signals Together - Additive Synthesis**

*Merge composition* can be used to sum an arbitrary number of signals together. Here's an example of a simple additive synthesizer (note that the result of the sum of the signals is divided by 3 to prevent clicking):

```
import("stdfaust.lib");
freq = hslider("freq",440,50,3000,0.01);
gain = hslider("gain",1,0,1,0.01);
gate = button("gate");
envelope = gain*gate : si.smoo;
process = os.osc(freq),os.osc(freq*2),os.osc(freq*3) :> /(3)*envelope;
```

While the resulting block diagram will look slightly different, this is mathematically equivalent to:

```
import("stdfaust.lib");
freq = hslider("freq",440,50,3000,0.01);
gain = hslider("gain",1,0,1,0.01);
gate = button("gate");
envelope = gain*gate : si.smoo;
process = (os.osc(freq) + os.osc(freq*2) + os.osc(freq*3))/(3)*envelope;
```

**Example: Connecting a Stereo Effect to a Mono One**

More generally, the *merge composition* can be used to connect a block with a certain number of output to a block with a smaller number of inputs:

```
import("stdfaust.lib");
drive = 0.6;
offset = 0;
process = dm.zita_light :> ef.cubicnl(drive,offset);
```

Note that an arbitrary number of signals can be split, for example:

```
import("stdfaust.lib");
drive = 0.6;
offset = 0;
process = par(i,2,dm.zita_light) :> par(i,2,ef.cubicnl(drive,offset));
```

Once again, the only rule with this is that in the expression `A:>B` the number of outputs of `A` has to be a multiple of the number of inputs of `B`.

### Recursive Composition

The *recursive composition* (e.g., `A~B`) is used to create cycles in the block-diagram in order to express recursive computations. It is the most complex operation in terms of connections.

To be applicable, it requires that:

$$\text{outputs}(A) \geq \text{inputs}(B) and \text{inputs}(A) \geq \text{outputs}(B)$$

Each input of $B$ is connected to the corresponding output of $A$ via an implicit 1-sample delay :

$$A[i] \overset{Z^{-1}}{\rightarrow} [i]B$$

and each output of $B$ is connected to the corresponding input of $A$:

$$B[i] \rightarrow [i]A$$

The inputs of the resulting block diagram are the remaining unconnected inputs of $A$. The outputs are all the outputs of $A$.

**Example: Timer**

*Recursive composition* can be used to implement a "timer" that will count each sample starting at time $n = 0$:

```
process = _~+(1);
```

The difference equation corresponding to this program is:

$$y(n) = y(n-1) + 1$$

an its output signal will look like: $(1, 2, 3, 4, 5, 6, \dots)$.

**Example: One Pole Filter**

*Recursive composition* can be used to implement a one pole filter with one line of code and just a few characters:

```
a1 = 0.999; // the pole
process = +~*(a1);
```

The difference equation corresponding to this program is:

$$y(n) = x(n) + a_1 y(n-1)$$

Note that the one sample delay of the filter is implicit here so it doesn't have to be declared.

### Inputs and Outputs of an Expression

The number of inputs and outputs of a Faust expression can be known at compile time simply by using `inputs(expression)` and `outputs(expression)`.

For example, the number of outputs of a sine wave oscillator can be known simply by writing the following program:

```
import("stdfaust.lib");
process = outputs(os.osc(440));
```

Note that Faust automatically simplified the expression by generating a program that just outputs `1`.

This type of construction is useful to define high order functions and build algorithmically complex block-diagrams. Here is an example to automatically reverse the order of the outputs of an expression.

```
Xo(expr) = expr <: par(i,n,ba.selector(n-i-1,n))
with {
  n = outputs(expr);
};
```

And the inputs of an expression :

```
Xi(expr) = si.bus(n) <: par(i,n,ba.selector(n-i-1,n)) : expr
with {
  n = inputs(expr);
};
```

For example `Xi(-)` will reverse the order of the two inputs of the substraction:

```
import("stdfaust.lib");
Xi(expr) = si.bus(n) <: par(i,n,ba.selector(n-i-1,n)) : expr
with {
  n = inputs(expr);
};
toto = os.osc(440),os.sawtooth(440), os.triangle(440);
process = Xi(-);
```

### Iterations

Iterations are analogous to `for(...)` loops in other languages and provide a convenient way to automate some complex block-diagram constructions.

The use and role of `par`, `seq`, `sum`, and `prod` are detailed in the following sections.

### `par` Iteration

The `par` iteration can be used to duplicate an expression in parallel. Just like other types of iterations in Faust:

- its first argument is a variable name containing the number of the current iteration (a bit like the variable that is usually named `i` in a for loop) starting at 0,
- its second argument is the number of iterations,
- its third argument is the expression to be duplicated.

**Example: Simple Additive Synthesizer**

```
import("stdfaust.lib");
freq = hslider("freq",440,50,3000,0.01);
gain = hslider("gain",1,0,1,0.01);
gate = button("gate");
envelope = gain*gate : si.smoo;
nHarmonics = 4;
process = par(i,nHarmonics,os.osc(freq*(i+1))) :> /(nHarmonics)*envelope;
```

`i` is used here at each iteration to compute the value of the frequency of the current oscillator. Also, note that this example could be re-wrtitten using `sum` iteration (see example in the corresponding section).

### `seq` Iteration

The `seq` iteration can be used to duplicate an expression in series. Just like other types of iterations in Faust:

- its first argument is a variable name containing the number of the current iteration (a bit like the variable that is usually named `i` in a for loop) starting at 0,
- its second argument is the number of iterations,
- its third argument is the expression to be duplicated.

#### Example: Peak Equalizer

The `fi.peak_eq` function of the Faust libraries implements a second order "peak equalizer" section (gain boost or cut near some frequency). When placed in series, it can be used to implement a full peak equalizer:

```
import("stdfaust.lib");
nBands = 8;
filterBank(N) = hgroup("Filter Bank",seq(i,N,oneBand(i)))
with{
    oneBand(j) = vgroup("[%j]Band %a",fi.peak_eq(l,f,b))
    with{
        a = j+1; // just so that band numbers don't start at 0
        l = vslider("[2]Level[unit:db]",0,-70,12,0.01) : si.smoo;
        f = nentry("[1]Freq",(80+(1000*8/N*(j+1)-80)),20,20000,0.01) : si.smoo;
        b = f/hslider("[0]Q[style:knob]",1,1,50,0.01) : si.smoo;
    };
};
process = filterBank(nBands);
```

Note that `i` is used here at each iteration to compute various elements and to format some labels. Having user interface elements with different names is a way to force their differentiation in the generated interface.

### `sum` Iteration

The `sum` iteration can be used to duplicate an expression as a sum. Just like other types of iterations in Faust:

- its first argument is a variable name containing the number of the current iteration (a bit like the variable that is usually named `i` in a for loop) starting at 0,
- its second argument is the number of iterations,

- its third argument is the expression to be duplicated.

**Example: Simple Additive Synthesizer**

The following example is just a slightly different version from the one presented in the `par` iteration section. While their block diagrams look slightly different, the generated code is exactly the same.

```
import("stdfaust.lib");
freq = hslider("freq",440,50,3000,0.01);
gain = hslider("gain",1,0,1,0.01);
gate = button("gate");
envelope = gain*gate : si.smoo;
nHarmonics = 4;
process = sum(i,nHarmonics,os.osc(freq*(i+1)))/(nHarmonics)*envelope;
```

`i` is used here at each iteration to compute the value of the frequency of the current oscillator.

### `prod` Iteration

The `sum` iteration can be used to duplicate an expression as a product. Just like other types of iterations in Faust:

- its first argument is a variable name containing the number of the current iteration (a bit like the variable that is usually named `i` in a for loop) starting at 0,
- its second argument is the number of iterations,
- its third argument is the expression to be duplicated.

**Example: Amplitude Modulation Synthesizer**

The following example implements an amplitude modulation synthesizer using an arbitrary number of oscillators thanks to the `prod` iteration:

```
import("stdfaust.lib");
freq = hslider("[0]freq",440,50,3000,0.01);
gain = hslider("[1]gain",1,0,1,0.01);
shift = hslider("[2]shift",0,0,1,0.01);
gate = button("[3]gate");
envelope = gain*gate : si.smoo;
nOscs = 4;
process = prod(i,nOscs,os.osc(freq*(i+1+shift)))*envelope;
```

`i` is used here at each iteration to compute the value of the frequency of the current oscillator. Note that the `shift` parameter can be used to tune the frequency drift between each oscillator.

### Infix Notation and Other Syntax Extensions

> Infix notation is commonly used in mathematics. It consists in placing the operand between the arguments as in $2+3$

Besides its algebra-based core syntax, Faust provides some syntax extensions, in particular the familiar *infix notation*. For example if you want to multiply two numbers, say `2` and `3`, you can write directly `2*3` instead of the equivalent core-syntax expression `2,3 : *`.

The *infix notation* is not limited to numbers or numerical expressions. Arbitrary expressions `A` and `B` can be used, provided that `A,B` has exactly two outputs. For example `_/2` is equivalent to `_,2:/` which divides the incoming signal by `2`.

Here are a few examples of equivalences:

| Infix Syntax | | Core Syntax |
|---|---|---|
| `2-3` | ≡ | `2,3 : -` |
| `2*3` | ≡ | `2,3 : *` |
| `_@7` | ≡ | `_,7 : @` |
| `_/2` | ≡ | `_,2 : /` |
| `A<B` | ≡ | `A,B : <` |

In case of doubts on the meaning of an infix expression, for example `_*_`, it is useful to translate it to its core syntax equivalent, here `_,_:*`, which is equivalent to `*`.

### Infix Operators

Built-in primitives that can be used in infix notation are called *infix operators* and are listed below. Please note that a more detailed description of these operators is available section on primitives.

### Prefix Notation

Beside *infix notation*, it is also possible to use *prefix notation*. The *prefix notation* is the usual mathematical notation for functions $f(x, y, z, \ldots)$, but extended to *infix operators*.

It consists in first having the operator, for example `/`, followed by its arguments between parentheses: `/(2,3)`:

| Prefix Syntax | | Core Syntax |
|---|---|---|
| `*(2,3)` | ≡ | `2,3 : *` |
| `@(_,7)` | ≡ | `_,7 : @` |

| Prefix Syntax | | Core Syntax |
|---|---|---|
| `/(_,2)` | ≡ | `_,2 : /` |
| `<(A,B)` | ≡ | `A,B : <` |

### Partial Application

The *partial application* notation is a variant of the *prefix notation* in which not all arguments are given. For instance `/(2)` (divide by 2), `^(3)` (rise to the cube), and `@(512)` (delay by 512 samples) are examples of partial applications where only one argument is given. The result of a partial application is a function that "waits" for the remaining arguments.

When doing partial application with an *infix operator*, it is important to note that the supplied argument is not the first argument, but always the second one:

| Prefix Partial Application Syntax | | Core Syntax |
|---|---|---|
| `+(C)` | ≡ | `_,C : *` |
| `-(C)` | ≡ | `_,C : -` |
| `<(C)` | ≡ | `_,C : <` |
| `/(C)` | ≡ | `_,C : /` |

For commutative operations that doesn't matter. But for non-commutative ones, it is more "natural" to fix the second argument. We use divide by 2 (`/(2)`) or rise to the cube (`^(3)`) more often than the other way around.

Please note that this rule only applies to infix operators, not to other primitives or functions. If you partially apply a regular function to a single argument, it will correspond to the first parameter.

**Example: Gain Controller**

The following example demonstrates the use of partial application in the context of a gain controller:

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

### `'` Time Expression

`'` is used to express a one sample delay. For example:

```
process = _';
```

will delay the incoming signal by one sample.

`'` time expressions can be chained, so the output signal of this program:

```
process = 1'';
```

will look like: $(0, 0, 1, 1, 1, 1, \ldots)$.

The `'` time expression is useful when designing filters, etc. and is equivalent to `@(1)` (see the `@` Time Expression).

### `@` Time Expression

`@` is used to express a delay with an arbitrary number of samples. For example:

```
process = @(10);
```

will delay the incoming signal by 10 samples.

A delay expressed with `@` doesn't have to be fixed but it must be positive and bounded. Therefore, the values of a slider are perfectly acceptable:

```
process = @(hslider("delay",0,0,100,1));
```

`@` only allows for the implementation of integer delay. Thus, various fractional delay algorithms are implemented in the Faust libraries.

### Environment Expressions

Faust is a lexically scoped language. The meaning of a Faust expression is determined by its context of definition (its lexical environment) and not by its context of use.

To keep their original meaning, Faust expressions are bounded to their lexical environment in structures called *closures*. The following constructions allow to explicitly create and access such environments. Moreover they provide powerful means to reuse existing code and promote modular design.

#### `with` Expression

The `with` construction allows to specify a *local environment*: a private list of definition that will be used to evaluate the left hand expression.

In the following example :

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

the definitions of `f(x)` and `g(x)` are local to `f : + ~ g`.

Please note that `with` is left associative and has the lowest priority:

- `f : + ~ g with {...}` is equivalent to `(f : + ~ g) with {...}`.
- `f : + ~ g with {...} with {...}` is equivalent to `((f : + ~ g) with {...}) with {...}`.

### `letrec` Expression

The `letrec` construction is somehow similar to `with`, but for difference equations instead of regular definitions. It allows us to easily express groups of mutually recursive signals, for example:

$$x(t) = y(t-1) + 10 y(t) = x(t-1) - 1$$

as `E letrec { 'x = y+10; 'y = x-1; }`

The syntax is defined by the following rules:

Note the special notation `'x = y + 10` instead of `x = y' + 10`. It makes syntactically impossible to write non-sensical equations like `x=x+1`.

Here is a more involved example. Let say we want to define an envelope generator with an attack and a release time (as a number of samples), and a gate signal. A possible definition could be:

```
import("stdfaust.lib");
ar(a,r,g) = v
letrec {
  'n = (n+1) * (g<=g');
  'v = max(0, v + (n<a)/a - (n>=a)/r) * (g<=g');
};
gate = button("gate");
process = os.osc(440)*ar(1000,1000,gate);
```

With the following semantics for $n(t)$ and $v(t)$:

$$n(t) = (n(t-1)+1) * (g(t) <= g(t-1)) v(t) = max(0, v(t-1) + (n(t-1) < a(t))/a(t) - (n(t-1) >= a(t))/r(t)) * (g(t$$

### `environment` Expression

The `environment` construction allows to create an explicit environment. It is like a ‘with’, but without the left hand expression. It is a convenient way to group together related definitions, to isolate groups of definitions and to create a name space hierarchy.

In the following example an `environment` construction is used to group together some constant definitions :

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

The `.` construction allows to access the definitions of an environment (see next section).

### Access Expression

Definitions inside an environment can be accessed using the `.` construction.

For example `constant.pi` refers to the definition of `pi` in the `constant` environment defined above.

Note that environments don't have to be named. We could have written directly:

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

### `library` Expression

The `library` construct allows to create an environment by reading the definitions from a file.

For example `library("filters.lib")` represents the environment obtained by reading the file `filters.lib`. It works like `import("miscfilter.lib")` but all the read definitions are stored in a new separate lexical environment. Individual definitions can be accessed as described in the previous paragraph. For example `library("filters.lib").lowpass` denotes the function `lowpass` as defined in the file `miscfilter.lib`.

To avoid name conflicts when importing libraries it is recommended to prefer `library` to `import`. So instead of :

```
import("filters.lib");
  ...
...lowpass....
    ...
};
```

the following will ensure an absence of conflicts :

```
fl = library("filters.lib");
  ...
...fl.lowpass....
    ...
```

```
};
```

In practice, that's how the `stdfaust.lib` library works.

### `component` Expression

The `component` construction allows us to reuse a full Faust program (e.g., a `.dsp` file) as a simple expression.

For example `component("freeverb.dsp")` denotes the signal processor defined in file `freeverb.dsp`.

Components can be used within expressions like in:

```
...component("karplus32.dsp") : component("freeverb.dsp")...
```

Please note that `component("freeverb.dsp")` is equivalent to `library("freeverb.dsp").process`.

`component` works well in tandem with explicit substitution (see next section).

### Explicit Substitution

Explicit substitution can be used to customize a component or any expression with a lexical environment by replacing some of its internal definitions, without having to modify it.

For example we can create a customized version of `component("freeverb.dsp")`, with a different definition of `foo(x)`, by writing:

```
...component("freeverb.dsp")[foo(x) = ...;]...
};
```

### Foreign Expressions

Reference to external C *functions*, *variables* and *constants* can be introduced using the *foreign function* mechanism.

#### `ffunction`

#### Signature

#### Types

#### Variables and Constants

#### File Include

#### Library File

#### Applications and Abstractions

#### Abstractions

#### Applications

#### Pattern Matching

### Primitives

## Using the Faust Compiler

## A Quick Tour of the Faust Targets

## Mathematical Documentation