FAUST Quick Reference

GRAME
Centre National de Création Musicale

December 2009

Contents

Chapter 1

Introduction

FAUST (*Functional Audio Stream*) is a functional programming language specifically designed for real-time signal processing and synthesis. FAUST targets high-performance signal processing applications and audio plug-ins for a variety of platforms and standards.

1.1 Design Principles

Various principles have guided the design of FAUST:

- FAUST is a *specification language*. It aims at providing an adequate notation to describe *signal processors* from a mathematical point of view. It is, as much as possible, free from implementation details.
- FAUST programs are fully compiled, not interpreted. The compiler translates FAUST programs into equivalent C++ programs 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 (`: , ~ <: :>`).

- Thanks to the notion 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.

1.2 Signal Processor Semantic

A FAUST program describes a *signal processor*. The role of a *signal processor* is to transforms a group of (possibly empty) *input signals* in order to produce a group of (possibly empty) *output signals*.

More precisely :

- A *signal* s is a discrete function of time $s : \mathbb{N} \to \mathbb{R}$. The value of signal s at time t is written $s(t)$. The set $\mathbb{S} = \mathbb{N} \to \mathbb{R}$ is the set of all possible signals.
- A group of n signals (a n-tuple of signals) is written $(s_1, \ldots, s_n) \in \mathbb{S}^n$. 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 produce 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 $Y(t) = 3$.

Chapter 2

Compiling and installing Faust

The FAUST source distribution `faust-0.9.10.tar.gz` can be downloaded from sourceforge (http://sourceforge.net/projects/faudiostream/).

2.1 Organization of the distribution

The first thing is to decompress the downloaded archive.

```
tar xzf faust-0.9.10.tar.gz
```

The resulting `faust-0.9.10/` folder should contain the following elements:

`architecture/`	FAUST libraries and architecture files
`benchmark`	tools to measure the efficiency of the generated code
`compiler/`	sources of the FAUST compiler
`examples/`	examples of FAUST programs
`syntax-highlighting/`	support for syntax highlighting for several editors
`documentation/`	FAUST's documentation, including this manual
`tools/`	tools to produce audio applications and plugins
`COPYING`	license information
`Makefile`	Makefile used to build and install FAUST
`README`	instructions on how to build and install FAUST

2.2 Compilation

FAUST has no dependencies outside standard libraries. Therefore the compilation should be straightforward. There is no configuration phase, to compile the FAUST compiler simply do :

```
cd faust-0.9.10/
make
```

If the compilation was successful you can test the compiler before installing it:

```
[cd faust-0.9.10/]
./compiler/faust -v
```

It should output:

```
FAUST, DSP to C++ compiler, Version 0.9.10
Copyright (C) 2002-2010, GRAME - Centre...
```

Then you can also try to compile one of the examples :

```
[cd faust-0.9.10/]
./compiler/faust examples/noise.dsp
```

It should produce some C++ code on the standard output

2.3 Installation

You can install FAUST with:

```
[cd faust-0.9.10/]
sudo make install
```

or

```
[cd faust-0.9.10/]
su
make install
```

depending on your system.

2.4 Compilation of the examples

Once FAUST correctly installed, you can have a look at the provided examples in the `examples/` folder. This folder contains a `Makefile` with all the required instructions to build these examples for various *architectures*[1], either standalone audio applications or plugins.

The command `make help` will list the available targets. Before using a specific target, make sure you have the appropriate development tools, libraries and headers installed. For example to compile the examples as ALSA applications with a GTK user interface do a `make alsagtk`. This will create a `alsagtkdir/` subfolder with all the binaries.

[1] an architecture file provides the code to connect a signal processor to the outside world (audio communications and user interface)

Chapter 3

Faust syntax

This section describes the syntax of FAUST. Figure 3.1 gives an overview of the various concepts and where they are defined in this section.

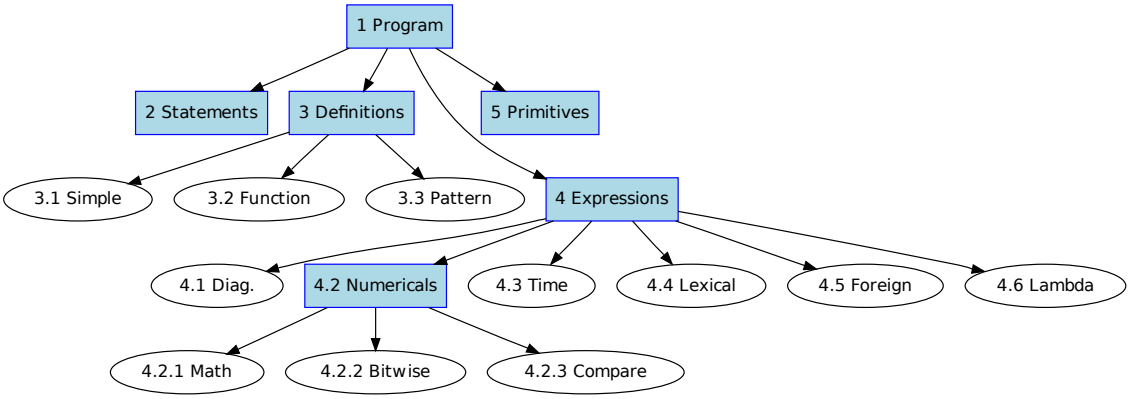


Figure 3.1: Overview of FAUST syntax

As we will see, *definitions* and *expressions* have a central role.

3.1 Faust program

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

⟨*program*⟩ ::= ⟨*statement*⟩

Here is a short FAUST program that implements of a simple noise generator. It exhibits the various kind of statements : two *declarations*, an *import*, a *comment* and a *definition*.

```
declare name        "noise";
declare copyright   "(c)GRAME 2006";

import("music.lib");

// noise level controlled by a slider
process = noise * vslider("volume", 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`.

3.2 Statements

The *statements* of a FAUST program are of three kinds : *metadata declarations*, *file imports* and *definitions*. All statements end with a semicolon (`;`).

⟨*statement*⟩ ::= ⟨*declaration*⟩ | ⟨*fileimport*⟩ | ⟨*definition*⟩ | ⟨*documentation*⟩

3.2.1 Declarations

Meta-data declarations (for example `declare name "noise";`) are optional and typically used to document a FAUST project.

⟨*declaration*⟩ ::= `declare` – ⟨*key*⟩ – ⟨*string*⟩ – `;`

⟨*key*⟩ ::= ⟨*identifier*⟩

Contrary to regular comments, these declarations will appear in the C++ code generated by the compiler.

3.2.2 Imports

File imports allow to import definitions from other source files.

⟨*fileimport*⟩ ::= `import` – `(` – ⟨*filename*⟩ – `)` – `;`

For example `import("math.lib");` imports the definitions of the `math.lib` library, a set of additional mathematical functions provided as foreign functions.

3.2.3 Documentation

Documentation statements are optional and typically used to control the generation of the mathematical documentation of a FAUST program. This documentation system is detailed chapter 6. In this section we will 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.

⟨*documentation*⟩ ::= `<mdoc>` ⟨*free text*⟩ | ⟨*equation*⟩ | ⟨*diagram*⟩ | ⟨*notice*⟩ | ⟨*listing*⟩ `</mdoc>`

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*.

⟨*equation*⟩ ::= `<equation>` – ⟨*expression*⟩ – `</equation>`

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.

⟨*diagram*⟩ ::= `<diagram>` – ⟨*expression*⟩ – `</diagram>`

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.

⟨*notice*⟩ ::= `<notice>`

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

⟨*listing*⟩ ::= `<listing>`

The `<listing>` tag is used to generate the full listing of the FAUST program, with all its dependencies.

3.3 Definitions

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

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.

3.3.1 Simple Definitions

The syntax of a simple definition is:

⟨*definition*⟩ ::= ⟨*identifier*⟩ – `=` – ⟨*expression*⟩ – `;`

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

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

3.3.2 Function Definitions

Definitions with formal parameters correspond to functions definitions.

⟨*definition*⟩ ::= ⟨*identifier*⟩ – `(` ⟨*parameter*⟩ (`,` ⟨*parameter*⟩)* `)` – `=` – ⟨*expression*⟩ – `;`

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));
```

3.3.3 Definitions with pattern matching

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

⟨*definition*⟩ ::= ⟨*identifier*⟩ – `(` ⟨*pattern*⟩ (`,` ⟨*pattern*⟩)* `)` – `=` – ⟨*expression*⟩ – `;`

⟨*pattern*⟩ ::= ⟨*identifier*⟩ | ⟨*expression*⟩

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:

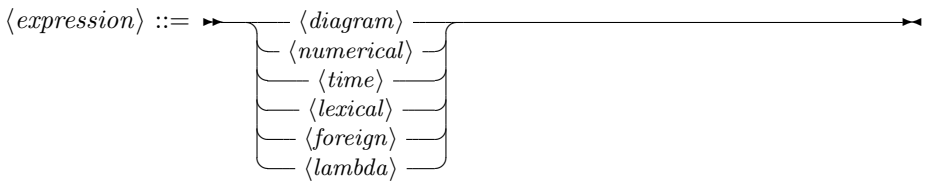
```
duplicate(1,exp) = exp;
duplicate(n,exp) = exp, duplicate(n-1,exp);
```

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

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

3.4 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.



3.4.1 Diagram Expressions

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

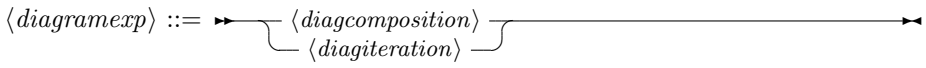
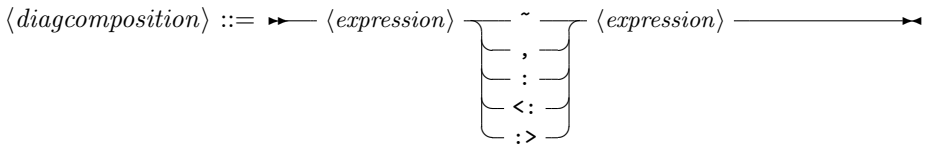


Diagram composition operations

Five binary composition operations are available to combine block-diagrams : *recursion*, *parallel*, *sequential*, *split* and *merge* composition.



Among these operations, recursion (`~`) has the highest priority and split (`<:`) and merge (`:>`) the lowest (see table 3.1).

Syntax	Pri.	Description
expression ~ *expression*	4	recursive composition
expression , *expression*	3	parallel composition
expression : *expression*	2	sequential composition
expression <: *expression*	1	split composition
expression :> *expression*	1	merge composition

Table 3.1: Block-Diagram composition operation priorities

Parallel Composition The *parallel composition* `A,B` (figure 3.2) 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`.

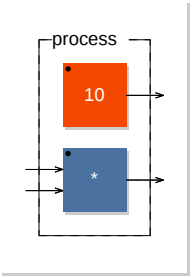


Figure 3.2: Example of parallel composition `(10,*)`

Sequential Composition The *sequential composition* `A:B` (figure 3.3) 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.

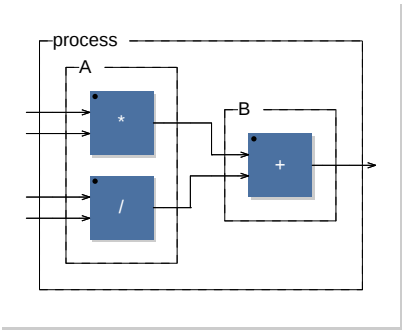


Figure 3.3: Example of sequential composition `((*,/):+)`

Split Composition The *split composition* `A<:B` (figure 3.4) operator is used to distribute the outputs of A to the inputs of B

Merge Composition The *merge composition* `A:>B` (figure 3.5)is used to connect several outputs of A to the same inputs of B.

Recursive Composition The *recursive composition* `A~B` (figure 3.6)is used to create cycles in the block-diagram in order to express recursive computations.

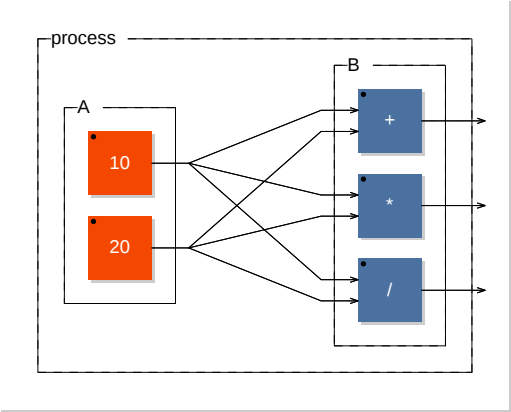


Figure 3.4: example of split composition `((10,20)<: (+,*,/))`

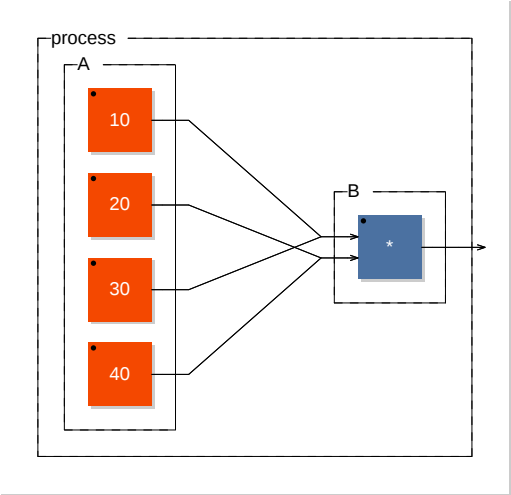
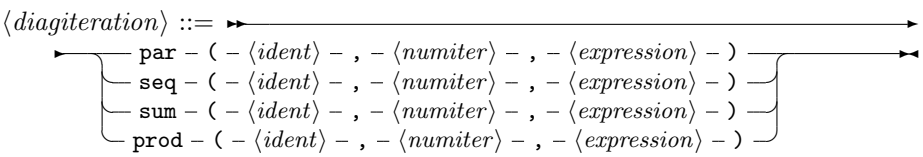


Figure 3.5: example of merge composition `((10,20,30,40):> *)`

Iterations

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



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

```
process  =  seq(i, 10,
                vgroup("band %i",
```

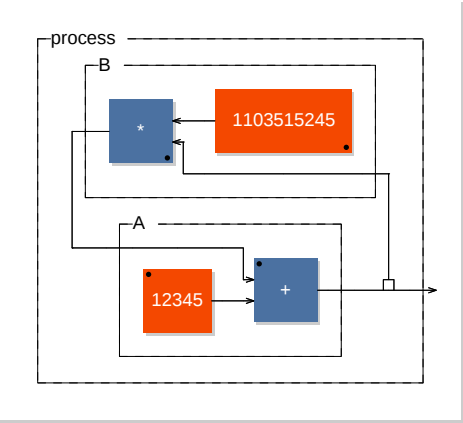


Figure 3.6: example of recursive composition `+(12345)~ *(1103515245)`

```
                    bandfilter( 1000*(1+i) )
                )
            );
```

⟨*numiter*⟩ ::= ⟨*expression*⟩

The number of iterations must be a constant expression.

3.4.2 Numerical Expressions

Numerical expressions are essentially syntactic sugar allowing to use a familiar infix notation to express mathematical expressions, bitwise operations and to compare signals. Please note that is this section only built-in primitives with an infix syntax are presented. A complete description of all the build-ins is available in the primitive section (see 3.5).

⟨*numerical*⟩ ::= ⟨*math*⟩ | ⟨*bitwise*⟩ | ⟨*comparison*⟩

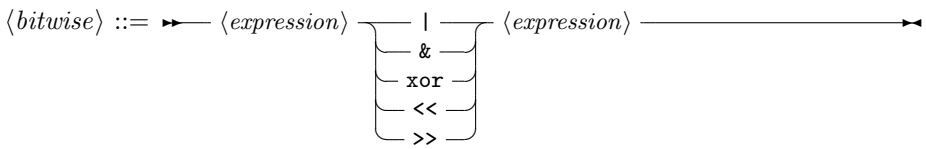
Mathematical expressions

are the familiar 4 operations as well as the modulo and power operations

⟨*math*⟩ ::= ⟨*expression*⟩ (+ | - | * | / | % | ^) ⟨*expression*⟩

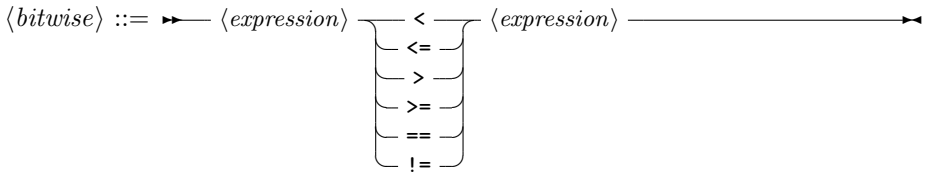
Bitwise expressions

are the boolean operations and the left and right arithmetic shifts.



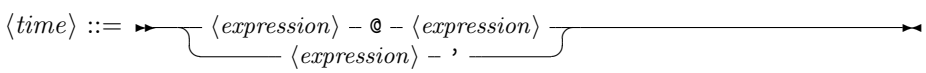
Comparison

operations allow to compare signals and result in a boolean signal that is 1 when the condition is true and 0 when the condition is false.



3.4.3 Time expressions

Time expressions are used to express delays. The notation `X@10` represent the signal `X` delayed by 10 samples. The notation `X'` represent the signal X delayed by one sample and is therefore equivalent to `X@1`.

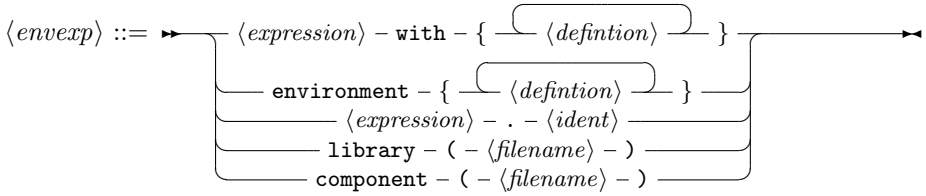


The delay don't have to be fixed, but it must be positive and bounded. The values of a slider are perfectcly acceptable as in the following example:

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

3.4.4 Environment expressions

Each FAUST expression has an associated lexical environment : a list of dictionaries where to look for definitions. The following expressions allow to create and access such environments.



With expression

allows to specify a *local environment*, a private list of definition that will be used to evaluate the left hand expression

⟨*withexpression*⟩ ::= ⟨*expression*⟩ – `with` – { ⟨*defintion*⟩ } ⟶

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';
};
```

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

Environment expression

allows to create an explicit environment: like a `with`, but without a 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.

⟨*environment*⟩ ::= `environment` – { ⟨*defintion*⟩ } ⟶

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

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

Access

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

⟨*access*⟩ ::= ⟨*expression*⟩ – . – ⟨*ident*⟩ ⟶

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

Please note that environment don't have to be named. We could have written directly `environment{pi = 3.14159; e = 2,718;....}.pi`

Library

allows to create an environment by reading the definitions from a file.

⟨*library*⟩ ::= `library` – `(` – ⟨*filename*⟩ – `)`

For example `library("filter.lib")` represents the lexical environment obtained by reading the file "filter.lib". It works like `import("filter.lib")` but all the read definitions are stored in a new separate lexical environment.

Component

is a powerful construction that allows to reuse a full FAUST program as a simple expression.

⟨*component*⟩ ::= `component` – `(` – ⟨*filename*⟩ – `)`

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`.

3.4.5 Foreign expressions

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

⟨*foreignexp*⟩ ::=
- `ffunction` – `(` – ⟨*signature*⟩ – `,` – ⟨*includefile*⟩ – `,` – ⟨*comment*⟩ – `)`
- `fvariable` – `(` – ⟨*type*⟩ – ⟨*identifier*⟩ – `,` – ⟨*includefile*⟩ – `)`
- `fconstant` – `(` – ⟨*type*⟩ – ⟨*identifier*⟩ – `,` – ⟨*includefile*⟩ – `)`

function

An external C function is declared by indicating its name and signature as well as the required include file. The file `"math.lib"` of the FAUST distribution contains several foreign function definitions, for example the inverse hyperbolic sine function `asinh`:

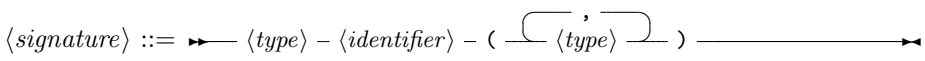
```
asinh = ffunction(float asinhf (float), <math.h>, "");
```

Foreign functions with input parameters are considered pure math functions. They are therefore considered free of side effects and called only when their parameters change (that is at the rate of the fastest parameter).

Exceptions are functions with no input parameters. A typical example is the C `rand()` function. In this case the compiler generate code to call the function at sample rate.

signature

The signature part (`float asinhf (float)` in our previous example) describes the prototype of the C function : return type, function name and list of parameter types.



types

Note that currently only numerical functions involving simple int and float parameters are allowed. No vectors, tables or data structures can be passed as parameters or returned.

⟨*type*⟩ ::= ⟨*int*⟩ | ⟨*float*⟩

variables and constants

External variables and constants can also be declared with a similar syntax. In the same `"math.lib"` file we can found the definition of the sampling rate constant `SR` and the definition of the block-size variable `BS` :

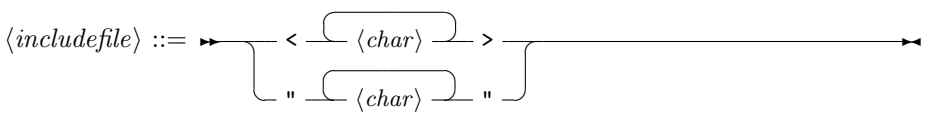
```
SR          = fconstant(int fSamplingFreq, <math.h>);
BS          = fvariable(int count, <math.h>);
```

Foreign constants are not supposed to vary. Therefore expressions involving only foreign constants are only computed once, during the initialization period.

Variable are considered to vary at block speed. This means that expressions depending of external variables are computed every block.

include file

In declaring foreign functions one as also to specify the include file. It allows the FAUST compiler to add the corresponding `#include...` in the generated code.



3.4.6 Applications and Abstractions

Abstractions and *applications* are fundamental programming constructions directly inspired by the Lambda-Calculus. These constructions provide powerful ways to describe and transform block-diagrams algorithmically.

Abstractions

Abstractions correspond to functions definitions and allow to generalize a block-diagram by *making variable* some of its parts.

⟨*abstraction*⟩ ::= ⟨*lambdaabstraction*⟩ | ⟨*patternabstraction*⟩

⟨*lambdaabstraction*⟩ ::= `\` – `(` ⟨*ident*⟩ (`,` ⟨*ident*⟩)* `)` – `.` – `(` – ⟨*expression*⟩ – `)`

Let's say you want to transform a stereo reverb, `freeverb` for instance, into a mono effect. You can write the following expression:

```
_ <: freeverb :> _
```

The incoming mono signal is splitted to feed the two input channels of the reverb, while the two output channels of the reverb are mixed together to produce the resulting mono output.

Imagine now that you are interested in transforming other stereo effects. It can be interesting to generalize this principle by making `freeverb` a variable:

```
\(freeverb).(_ <: freeverb :> _)
```

The resulting abstraction can then be applied to transform other effects. Note that if `freeverb` is a perfectly valid variable name, a more neutral name would probably be easier to read like:

```
\(fx).(_ <: fx :> _)
```

Moreover it could be convenient to give a name to this abstraction:

```
mono = \(fx).(_ <: fx :> _);
```

Or even use a more traditional, but equivalent, notation:

```
mono(fx) = _ <: fx :> _;
```

Applications

Applications correspond to function calls and allow to replace the variable parts of an abstraction with the specified arguments.

⟨*application*⟩ ::= ⟨*expression*⟩ – `(` (⟨*expression*⟩ `,`)* ⟨*expression*⟩ `)`

For example you can apply the previous abstraction to transform your stereo harmonizer:

```
mono(harmonizer)
```

The compiler will start by replacing `mono` by its definition:

```
\(fx).(_ <: fx :> _)(harmonizer)
```

Whenever the FAUST compiler find an application of an abstraction it replaces the variable part with the argument [1]. The resulting expression is as expected:

```
(_ <: harmonizer :> _)
```

Pattern Matching

Pattern matching rules provide an effective way to analyze and transform block-diagrams algorithmically.

⟨*patternabstraction*⟩ ::= ▸▸ `case` – { ⟨*rule*⟩ } ⋈

⟨*rule*⟩ ::= ▸▸ `(` ⟨*pattern*⟩ `,` ⟨*pattern*⟩ `)` – `=>` – ⟨*expression*⟩ – `;` ⋈

⟨*pattern*⟩ ::= ⟨*ident*⟩ | ⟨*expression*⟩

For example `case{ (x:y)=> y:x; (x)=> x; }` contains two rules. The first one will match a sequential expression and invert the two part. The second one will match all remaining expressions and leave it untouched. Therefore the application:

```
case{(x:y) => y:x; (x) => x;}(freeverb:harmonizer)
```

will produce:

```
(harmonizer:freeverb)
```

Please note that patterns are evaluated before the pattern matching operation. Therefore only variables that appear free in the pattern are binding variables during pattern matching.

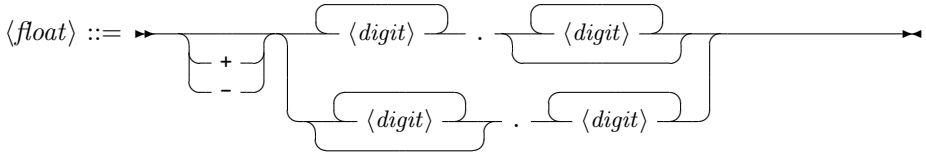
3.5 Primitives

The primitive signal processing operations represent the built-in functionalities of FAUST, that is the atomic operations on signals provided by the language. All these primitives denote *signal processors*, functions transforming *input signals* into *output signals*.

[1] This is called β-reduction in Lambda-Calculus

3.5.1 Numbers

FAUST considers two types of numbers : *integers* and *floats*. Integers are implemented as 32-bits integers, and floats are implemented either with a simple, double or extended precision depending of the compiler options.



⟨*digit*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Like any other FAUST expression, numbers are signal processors. For example the number 0.95 is a signal processor of type $\mathbb{S}^0 \rightarrow \mathbb{S}^1$ that transforms an empty tuple of signals () into a 1-tuple of signals (y) such that $\forall t \in \mathbb{N}, y(t) = 0.95$.

3.5.2 C-equivalent primitives

Most FAUST primitives are analogue to their C counterpart but lifted to signal processing. For example + is a function of type $\mathbb{S}^2 \rightarrow \mathbb{S}^1$ that transforms a pair of signals (x_1, x_2) into a 1-tuple of signals (y) such that $\forall t \in \mathbb{N}, y(t) = x_1(t)+x_2(t)$.

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

3.5.3 math.h-equivalent primitives

Most of the C math.h functions are also built-in as primitives (the others are defined as external functions in file math.lib).

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

3.5.4 Delay, Table, Selector primitives

The following primitives allow to define fixed delays, read-only and read-write tables and 2 or 3-ways selectors (see figure 3.7).

Syntax	Type	Description
`mem`	$\mathbb{S}^1 \to \mathbb{S}^1$	1-sample delay: $y(t+1) = x(t), y(0) = 0$
`prefix`	$\mathbb{S}^2 \to \mathbb{S}^1$	1-sample delay: $y(t+1) = x_2(t), y(0) = x_1(0)$
`@`	$\mathbb{S}^2 \to \mathbb{S}^1$	fixed 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: $T[] = \{x_0(t), x_1(t), x_2(t)\}; y(t) = T[s(t)]$

3.5.5 User Interface Elements

FAUST user interface widgets allow an abstract description of the user interface from within the FAUST code. This description is independent of any GUI toolkits. It is based on *buttons*, *checkboxes*, *sliders*, etc. that are grouped together vertically and horizontally using appropriate grouping schemes.

All these GUI elements produce signals. A button for example (see figure 3.8) produces a signal which is 1 when the button is pressed and 0 otherwise. These signals can be freely combined with other audio signals.

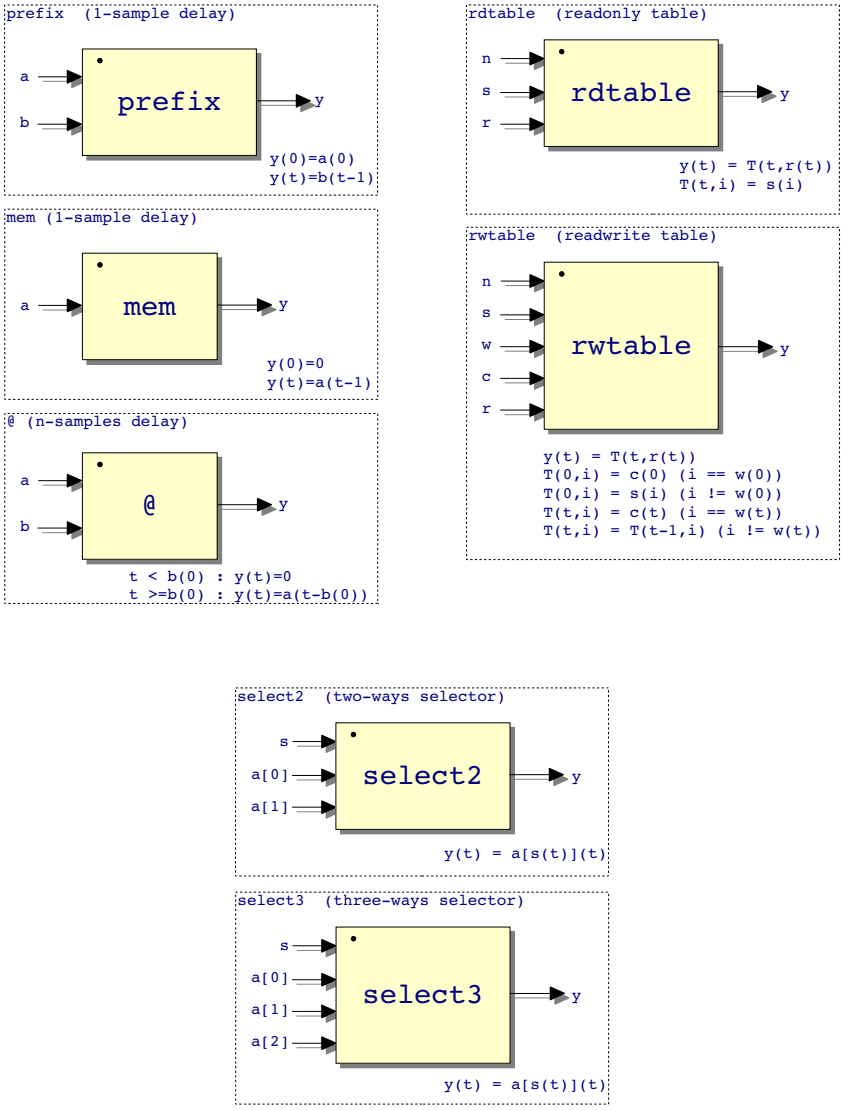


Figure 3.7: Delays, tables and selectors primitives

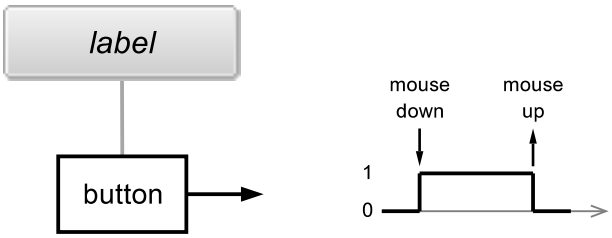


Figure 3.8: User Interface Button

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

Labels

Every user interface widget has a label (a string) that identifies it and informs the user of its purpose. There are three important mechanisms associated with labels : *variable parts*, *pathnames* and *metadata*.

Variable parts. Labels can contain variable parts. These variable parts are indicated by the sign '`%`' followed by the name of a variable. During compilation each label is processed in order to replace the variable parts by the value of the variable. For example `par(i,8,hslider("Voice %i", 0.9, 0, 1, 0.01) )` creates 8 different sliders in parallel :

```
hslider("Voice 0", 0.9, 0, 1, 0.01),
hslider("Voice 1", 0.9, 0, 1, 0.01),
...
hslider("Voice 7", 0.9, 0, 1, 0.01).
```

while `par(i,8,hslider("Voice", 0.9, 0, 1, 0.01))` would have created only one slider and duplicated its output 8 times.

An escape mechanism is provided. If the sign `%` is followed by itself, it will be included in the resulting string. For example `"feedback (%%)"` will result in `"feedback (%)"`.

Pathnames. Thanks to horizontal, vertical and tabs groups, user interfaces have a hierarchical structure analog to a hierarchical file system. Each widget has an associated *pathname* obtained by concatenating the labels of all its surrounding groups with its own label.

In the following example :

```
hgroup("Foo",
    ...
    vgroup("Faa",
        ...
        hslider("volume",...)
        ...
    )
    ...
)
```

the volume slider has pathname `/h:Foo/v:Faa/volume`.

In order to give more flexibility to the design of user interfaces, it is possible to explicitly specify the absolute or relative pathname of a widget directly in its label.

In our previous example the pathname of :

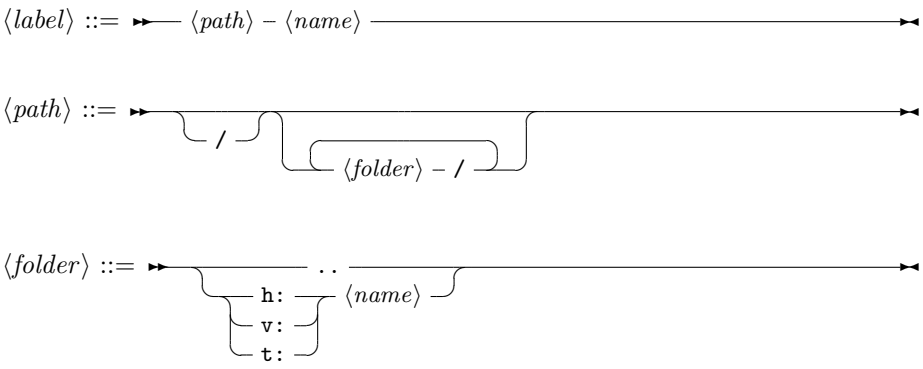
```
    hslider("../volume",...)
```

would have been `/h:Foo/volume`, while the pathname of :

```
    hslider("t:Fii/volume",...)
```

would have been : `/h:Foo/v:Faa/t:Fii/volume`.

The grammar for labels with pathnames is the following:



Metadata Widget labels can contain metadata enclosed in square brackets. These metadata associate a key with a value and are used to provide additional information to the architecture file. They are tipically used to improve the look and feel of the user interface. The FAUST code :

```
process = *(hslider("foo [key1: val 1][key2: val 2]",
                    0, 0, 1, 0.1));
```

will produce and the corresponding C++ code :

```
class mydsp : public dsp {
    ...
    virtual void buildUserInterface(UI* interface) {
        interface->openVerticalBox("m");
        interface->declare(&fslider0, "key1", "val 1")
            ;
        interface->declare(&fslider0, "key2", "val 2")
            ;
        interface->addHorizontalSlider("foo", &
            fslider0,
                                0.0f, 0.0f, 1.0f, 0.1f
                                    );
        interface->closeBox();
    }
...
};
```

All the metadata are removed from the label by the compiler and transformed in calls to the `UI::declare()` method. All these `UI::declare()` calls will always take place before the `UI::AddSomething()` call that creates the User Interface element. This allows the `UI::AddSomething()` method to make full use of the available metadata.

It is the role of the architecture file to decide what to do with these metadata. The `jack-qt.cpp` architecture file for example implements the following :

1. `"...[type:knob]..."` creates a rotative knob instead of a regular slider or nentry.
2. `"...[type:led]..."` in a bargraph's label creates a small LED instead of a full bargraph
3. `"...[unit:dB]..."` in a bargraph's label creates a more realistic bargraph with colors ranging from green to red depending of the level of the value
4. `"...[unit:xx]..."` in a widget postfix the value displayed with xx
5. `"...[tooltip:bla bla]..."` add a tooltip to the widget

Moreover starting a label with a number option like in `"[1]..."` provide a convenient mean to control the alphabetical order of the widgets

Chapter 4

Invoking the Faust compiler

The FAUST compiler is invoked using the `faust` command. It translate FAUST programs into C++ code. The generated code can be wrapped into an optional *architecture file* allowing to directly produce a fully operational program.

compiler:- `faust` -(options)(file+)

For example `faust noise.dsp` will compile `noise.dsp` and output the corresponding C++ code on the standard output. The option `-o` allows to choose the output file : `faust noise.dsp -o noise.cpp`. The option `-a` allows to choose the architecture file : `faust -a alsa-gtk.cpp noise.dsp`.

To compile a FAUST program into an ALSA application on Linux you can use the following commands:

```
faust -a alsa-gtk.cpp noise.dsp -o noise.cpp
g++ -lpthread -lasound
    `pkg-config --cflags --libs gtk+-2.0`
    noise.cpp -o noise
```

Compilation options are listed in the following table :

Short	Long	Description
`-h`	`--help`	print the help message
`-v`	`--version`	print version information
`-d`	`--details`	print compilation details
`-ps`	`--postscript`	generate block-diagram postscript file
`-svg`	`--svg`	generate block-diagram svg files
`-blur`	`--shadow-blur`	add a blur to boxes shadows
`-sd`	`--simplify-diagrams`	simplify block-diagram before drawing them
`-f` *n*	`--fold` *n*	max complexity of svg diagrams before splitting into several files (default 25 boxes)

continued on next page

Short	Long	Description
`-mns` *n*	`--max-name-size` *n*	max character size used in svg diagram labels
`-sn`	`--simple-names`	use simple names (without arguments) for block-diagram (default max size : 40 chars)
`-xml`	`--xml`	generate an additional description file in xml format
`-uim`	`--user-interface-macros`	add user interface macro definitions to the C++ code
`-flist`	`--file-list`	list all the source files and libraries implied in a compilation
`-lb`	`--left-balanced`	generate left-balanced expressions
`-mb`	`--mid-balanced`	generate mid-balanced expressions (default)
`-rb`	`--right-balanced`	generate right-balanced expressions
`-lt`	`--less-temporaries`	generate less temporaries in compiling delays
`-mcd` *n*	`--max-copy-delay` *n*	threshold between copy and ring buffer delays (default 16 samples)
`-vec`	`--vectorize`	generate easier to vectorize code
`-vs` *n*	`--vec-size` *n*	size of the vector (default 32 samples) when -vec
`-lv` *n*	`--loop-variant` *n*	loop variant [0:fastest (default), 1:simple] when -vec
`-dfs`	`--deepFirstScheduling`	schedule vector loops in deep first order when -vec
`-omp`	`--openMP`	generate parallel code using openMP (implies -vec)
`-sch`	`--scheduler`	generate parallel code using threads directly (implies -vec)
`-g`	`--groupTasks`	group sequential tasks together when -omp
`-single`	`--single-precision-floats`	use floats for internal computations (default)
`-double`	`--double-precision-floats`	use doubles for internal computations
`-quad`	`--quad-precision-floats`	use extended for internal computations
`-mdoc`	`--mathdoc`	generates the full mathematical description of a FAUST program
`-mdlang` *l*	`--mathdoc-lang` *l*	choose the language of the mathematical description (*l* = en, fr, ...)
`-stripmdoc`	`--strip-mdoc-tags`	remove documentation tags when printing FAUST listings
`-a` *file*		architecture file to use
`-o` *file*		C++ output file

The main available architecture files are :

File name	Description
jack-gtk.cpp	Jack GTK standalone application
jack-qt.cpp	Jack QT4 standalone application
jack-console.cpp	Jack command line application
jack-internal.cpp	Jack serve plugin
jack-wx.cpp	Jack wxWindows standalone application
alsa-gtk.cpp	ALSA GTK standalone application
alsa-qt.cpp	ALSA QT4 standalone application
oss-gtk.cpp	OSS GTK standalone application
oss-wx.cpp	OSS wxWindows standalone application
pa-gtk.cpp	PortAudio GTK standalone application
pa-qt.cpp	PortAudio QT4 standalone application
pa-wx.cpp	PortAudio wxWindows standalone application
max-msp.cpp	Max/MSP external
vst.cpp	VST plugin
vst2p4.cpp	VST 2.4 plugin
vsti-mono.cpp	VSTi mono instrument
ladspa.cpp	LADSPA plugin
q.cpp	Q language plugin
supercollider.cpp	SuperCollider Unit Generator
csound.cpp	CSOUND opcode
puredata.cpp	PD external
sndfile.cpp	sound file transformation command
bench.cpp	speed benchmark
octave.cpp	Octave plugin
plot.cpp	Command line application
sndfile.cpp	Command line application

Here is an example of compilation command that generates the C++ source code of a Jack application using the GTK graphic toolkit:

`faust -a jack-gtk.cpp -o freeverb.cpp freeverb.dsp`.

Chapter 5

Controlling the code generation

Several options of the FAUST compiler allow to control how the C++ code generated. By default the computations are done sample by sample in a single loop. But the compiler can also generate *vector* and *parallel* code.

5.1 Vector Code generation

Modern C++ compilers are able to do autovectorization, that is to use SIMD instructions to speedup the code. These instructions can typically operate in parallel on short vectors of 4 simple precision floating point numbers thus leading to a theoretical speedup of ×4. Autovectorization of C/C++ programs is a difficult task. Current compilers are very sensitive to the way the code is arranged. In particular too complex loops can prevent autovectorization. The goal of the vector code generation is to rearrange the C++ code in a way that facilitates the autovectorization job of the C++ compiler. Instead of generating a single sample computation loop, it splits the computation into several simpler loops that communicates by vectors.

The vector code generation is activated by passing the `--vectorize` (or `-vec`) option to the FAUST compiler. Two additional options are available: `--vec-size <n>` controls the size of the vector (by default 32 samples) and `--loop-variant 0/1` gives some additional control on the loops.

To illustrate the difference between scalar code and vector code, let's take the computation of the RMS (Root Mean Square) value of a signal. Here is the FAUST code that computes the Root Mean Square of a sliding window of 1000 samples:

```
// Root Mean Square of n consecutive samples
RMS(n) = square : mean(n) : sqrt ;

// Square of a signal
square(x) = x * x ;
```

```
// Mean of n consecutive samples of a signal
// (uses fixpoint to avoid the accumulation of
// rounding errors)
mean(n) = float2fix : integrate(n) :
          fix2float : /(n);

// Sliding sum of n consecutive samples
integrate(n,x) = x - x@n : +~_ ;

// Convertion between float and fix point
float2fix(x) = int(x*(1<<20));
fix2float(x) = float(x)/(1<<20);

// Root Mean Square of 1000 consecutive samples
process = RMS(1000) ;
```

The compute() method generated in scalar mode is the following:

```
virtual void compute (int count,
                      float** input,
                      float** output)
{
  float* input0 = input[0];
  float* output0 = output[0];
  for (int i=0; i<count; i++) {
    float fTemp0 = input0[i];
    int iTemp1 = int(1048576*fTemp0*fTemp0);
    iVec0[IOTA&1023] = iTemp1;
    iRec0[0] = ((iVec0[IOTA&1023] + iRec0[1])
                  - iVec0[(IOTA-1000)&1023]);
    output0[i] = sqrtf(9.536744e-10f *
                       float(iRec0[0]));
    // post processing
    iRec0[1] = iRec0[0];
    IOTA = IOTA+1;
  }
}
```

The `-vec` option leads to the following reorganization of the code:

```
virtual void compute (int fullcount,
                      float** input,
                      float** output)
{
  int     iRec0_tmp[32+4];
  int*    iRec0 = &iRec0_tmp[4];
  for (int index=0; index<fullcount; index+=32)
  {
    int count = min (32, fullcount-index);
    float* input0 = &input[0][index];
```

```
    float* output0 = &output[0][index];
    for (int i=0; i<4; i++)
      iRec0_tmp[i]=iRec0_perm[i];
    // SECTION : 1
    for (int i=0; i<count; i++) {
      iYec0[(iYec0_idx+i)&2047] =
               int(1048576*input0[i]*input0[i]);
    }
    // SECTION : 2
    for (int i=0; i<count; i++) {
      iRec0[i] = ((iYec0[i] + iRec0[i-1]) -
               iYec0[(iYec0_idx+i-1000)&2047]);
    }
    // SECTION : 3
    for (int i=0; i<count; i++) {
      output0[i] = sqrtf((9.536744e-10f *
                 float(iRec0[i])));
    }
    // SECTION : 4
    iYec0_idx = (iYec0_idx+count)&2047;
    for (int i=0; i<4; i++)
      iRec0_perm[i]=iRec0_tmp[count+i];
  }
}
```

While the second version of the code is more complex, it turns out to be much easier to vectorize efficiently by the C++ compiler. Using Intel icc 11.0, with the exact same compilation options: `-O3 -xHost -ftz -fno-alias -fp-model fast=2`, the scalar version leads to a throughput performance of 129.144 MB/s, while the vector version achieves 359.548 MB/s, a speedup of x2.8 !

The vector code generation is built on top of the scalar code generation (see figure 5.1). Every time an expression needs to be compiled, the compiler checks to see if it needs to be in a separate loop or not. It applies some simple rules for that. Expressions that are shared (and are complex enough) are good candidates to be compiled in a separate loop, as well as recursive expressions and expressions used in delay lines.

The result is a directed graph in which each node is a computation loop (see Figure 5.2). This graph is stored in the klass object and a topological sort is applied to it before printing the code.

5.2 Parallel Code generation

The parallel code generation is activated by passing the `--openMP` (or `-omp`) option to the FAUST compiler. It implies the `-vec` options as the parallel code generation is built on top of the vector code generation by inserting appropriate OpenMP directives in the C++ code.

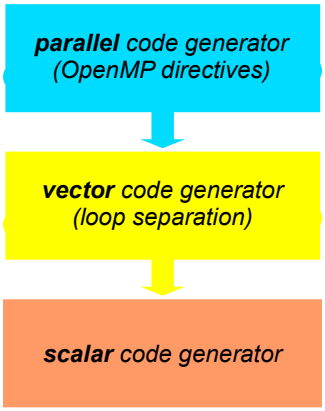


Figure 5.1: FAUST's stack of code generators

5.2.1 The OpenMP API

OpenMP (http://wwww.openmp.org) is a well established API that is used to explicitly define direct multi-threaded, shared memory parallelism. It is based on a fork-join model of parallelism (see figure 5.3). Parallel regions are delimited by using the `#pragma omp parallel` construct. At the entrance of a parallel region a team of parallel threads is activated. The code within a parallel region is executed by each thread of the parallel team until the end of the region.

```
#pragma omp parallel
{
  // the code here is executed simultaneously by
  // every thread of the parallel team
  ...
}
```

In order not to have every thread doing redundantly the exact same work, OpemMP provides specific *work-sharing* directives. For example `#pragma omp sections` allows to break the work into separate, discrete sections. Each section being executed by one thread:

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    {
      // job 1
    }
```

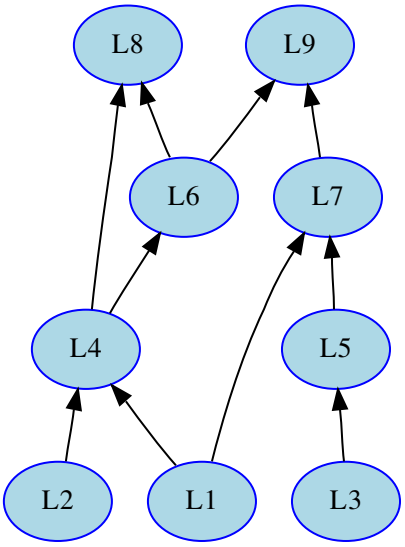


Figure 5.2: The result of the -vec option is a directed acyclic graph (DAG) of small computation loops

```
    #pragma omp section
    {
      // job 2
    }
    ...
  }

  ...
}
```

5.2.2 Adding OpenMP directives

As said before the parallel code generation is built on top of the vector code generation. The graph of loops produced by the vector code generator is topologically sorted in order to detect the loops that can be computed in parallel. The first set S_0 (loops $L1$, $L2$ and $L3$ in the DAG of Figure 5.2) contains the loops that don't depend on any other loops, the set S_1 contains the loops that only depend on loops of S_0, (that is loops $L4$ and $L5$), etc..

As all the loops of a given set S_n can be computed in parallel, the compiler will generate a `sections` construct with a `section` for each loop.

```
  #pragma omp sections
```

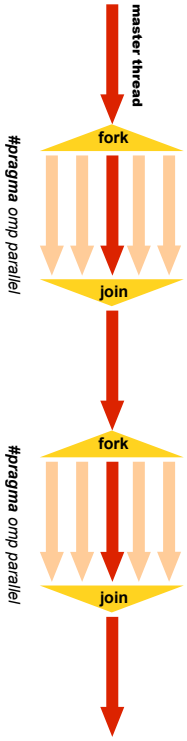


Figure 5.3: OpenMP is based on a fork-join model

```
{
  #pragma omp section
  for (...) {
    // Loop 1
  }
  #pragma omp section
  for (...) {
    // Loop 2
  }
  ...
}
```

If a given set contains only one loop, then the compiler checks to see if the loop can be parallelized (no recursive dependencies) or not. If it can be parallelized, it generates:

```
#pragma omp for
for (...) {
 // Loop code
}
```

otherwise it generates a `single` construct so that only one thread will execute the loop:

```
  #pragma omp single
  for (...) {
   // Loop code
  }
```

5.2.3 Example of parallel code

To illustrate how FAUST uses the OpenMP directives, here is a very simple example, two 1-pole filters in parallel connected to an adder (see figure 5.4 the corresponding block-diagram):

```
filter(c) = *(1-c) : + ~ *(c);
process = filter(0.9), filter(0.9) : +;
```

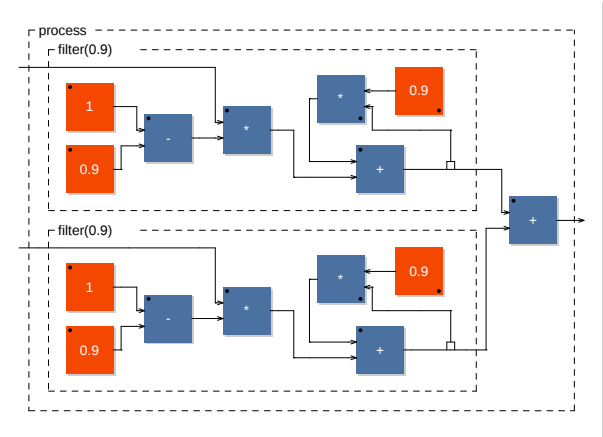


Figure 5.4: two filters in parallel connected to an adder

The corresponding compute() method obtained using the -omp option is the following:

```
virtual void compute (int fullcount,
                      float** input,
                      float** output)
{
  float   fRec0_tmp[32+4];
  float   fRec1_tmp[32+4];
  float*  fRec0 = &fRec0_tmp[4];
  float*  fRec1 = &fRec1_tmp[4];
  #pragma omp parallel firstprivate(fRec0,fRec1)
  {
    for (int index = 0; index < fullcount;
                                index += 32)
    {
      int count = min (32, fullcount-index);
      float* input0 = &input[0][index];
```

```
      float* input1 = &input[1][index];
      float* output0 = &output[0][index];
      #pragma omp single
      {
        for (int i=0; i<4; i++)
          fRec0_tmp[i]=fRec0_perm[i];
        for (int i=0; i<4; i++)
          fRec1_tmp[i]=fRec1_perm[i];
      }
      // SECTION : 1
      #pragma omp sections
      {
        #pragma omp section
        for (int i=0; i<count; i++) {
          fRec0[i] = ((0.1f * input1[i])
                   + (0.9f * fRec0[i-1]));
        }
        #pragma omp section
        for (int i=0; i<count; i++) {
          fRec1[i] = ((0.1f * input0[i])
                   + (0.9f * fRec1[i-1]));
        }
      }
      // SECTION : 2
      #pragma omp for
      for (int i=0; i<count; i++) {
        output0[i] = (fRec1[i] + fRec0[i]);
      }
      // SECTION : 3
      #pragma omp single
      {
        for (int i=0; i<4; i++)
          fRec0_perm[i]=fRec0_tmp[count+i];
        for (int i=0; i<4; i++)
          fRec1_perm[i]=fRec1_tmp[count+i];
      }
    }
  }
}
```

This code appeals for some comments:

1. The parallel construct `#pragma omp parallel` is the fundamental construct that starts parallel execution. The number of parallel threads is generally the number of CPU cores but it can be controlled in several ways.

2. Variables external to the parallel region are shared by default. The pragma `firstprivate(fRec0,fRec1)` indicates that each thread should have its private copy of fRec0 and fRec1. The reason is that accessing shared vari-

ables requires an indirection and is quite inefficient compared to private copies.

3. The top level loop `for (int index = 0;...)...` is executed by all threads simultaneously. The subsequent work-sharing directives inside the loop will indicate how the work must be shared between the threads.

4. Please note that an implied barrier exists at the end of each work-sharing region. All threads must have executed the barrier before any of them can continue.

5. The work-sharing directive `#pragma omp single` indicates that this first section will be executed by only one thread (any of them).

6. The work-sharing directive `#pragma omp sections` indicates that each corresponding `#pragma omp section`, here our two filters, will be executed in parallel.

7. The loop construct `#pragma omp for` specifies that the iterations of the associated loop will be executed in parallel. The iterations of the loop are distributed across the parallel threads. For example, if we have two threads, the first one can compute indices between 0 and count/2 and the other between count/2 and count.

8. Finally `#pragma omp single` in section 3 indicates that this last section will be executed by only one thread (any of them).

Chapter 6

Mathematical Documentation

The FAUST compiler provides a mechanism to produce a self-describing documentation of the mathematical semantic of a FAUST program, essentially as a pdf file. The corresponding options are `-mdoc` (short) or `--mathdoc` (long).

6.1 Goals of the mathdoc

There are three main goals, or uses, of this mathematical documentation:

1. to preserve signal processors, independently from any computer language but only under a mathematical form;

2. to bring some help for debugging tasks, by showing the formulas as they are really computed after the compilation stage;

3. to give a new teaching support, as a bridge between code and formulas for signal processing.

6.2 Installation requirements

- `faust`, of course!

- `svg2pdf` (from the Cairo 2D graphics library), to convert block-diagrams, as LaTeX doesn't eat SVG directly yet...

- `breqn`, a LaTeX package to handle automatic breaking of long equations,

- `pdflatex`, to compile the LaTeX output file.

6.3 Generating the mathdoc

The easiest way to generate the complete mathematical documentation is to call the `faust2mathdoc` script on a FAUST file, as the `-mdoc` option leave the documentation production unfinished. For example:

```
faust2mathdoc noise.dsp
```

6.3.1 Invoking the -mdoc option

Calling directly `faust -mdoc` does only the first part of the work, generating:

- a top-level directory, suffixed with "`-mdoc`",
- 5 subdirectories (`cpp/`, `pdf/`, `src/`, `svg/`, `tex/`),
- a LaTeX file containing the formulas,
- SVG files for block-diagrams.

At this stage:

- `cpp/` remains empty,
- `pdf/` remains empty,
- `src/` contains all FAUST sources used (even libraries),
- `svg/` contains SVG block-diagram files,
- `tex/` contains the generated LaTeX file.

6.3.2 Invoking faust2mathdoc

The `faust2mathdoc` script calls `faust --mathdoc` first, then it finishes the work:

- moving the output C++ file into `cpp/`,
- converting all SVG files into pdf files (you must have `svg2pdf` installed, from the Cairo 2D graphics library),
- launching `pdflatex` on the LaTeX file (you must have both `pdflatex` and the `breqn` package installed),
- moving the resulting pdf file into `pdf/`.

6.3.3 Online examples

To have an idea of the results of this mathematical documentation, which captures the mathematical semantic of FAUST programs, you can look at two pdf files online:

- http://faust.grame.fr/pdf/karplus.pdf (automatic documentation),
- http://faust.grame.fr/pdf/noise.pdf (manual documentation).

You can also generate all *mdoc* pdfs at once, simply invoking the `make mathdoc` command inside the `examples/` directory:

- for each `%.dsp` file, a complete `%-mdoc` directory will be generated,
- a single `allmathpdfs/` directory will gather all the generated pdf files.

6.4 Automatic documentation

By default, when no `<mdoc>` tag can be found in the input FAUST file, the `-mdoc` option automatically generates a LaTeX file with four sections:

1. "**Equations of process**", gathering all formulas needed for `process`,
2. "**Block-diagram schema of process**", showing the top-level block-diagram of `process`,
3. "**Notice of this documentation**", summing up generation and conventions information,
4. "**Complete listing of the input code**", listing all needed input files (including libraries).

6.5 Manual documentation

You can specify yourself the documentation instead of using the automatic mode, with five xml-like tags. That permits you to modify the presentation and to add your own comments, not only on `process`, but also about any expression you'd like to. Note that as soon as you declare an `<mdoc>` tag inside your FAUST file, the default structure of the automatic mode is ignored, and all the LaTeX stuff becomes up to you!

6.5.1 Five tags

Here are the five specific tags:

- `<mdoc></mdoc>`, to open a documentation field in the FAUST code,
  - `<equation></equation>`, to get equations of a FAUST expression,

- `<diagram></diagram>`, to get the top-level block-diagram of a FAUST expression,
- `<notice>`, to insert the "adaptive" notice all formulas actually printed,
- `<listing>`, to insert the complete listings of all FAUST files called.

6.5.2 The mdoc top-level tags

The `<mdoc></mdoc>` tags are the top-level delimiters for FAUST mathematical documentation sections. This means that the four other documentation tags can't be used outside these pairs (see section 3.2.3).

In addition of the four inner tags, `<mdoc></mdoc>` tags accept free LaTeX text, including its standard macros (like `\section`, `\emph`, etc.). This allows to manage the presentation of resulting tex file directly from within the input FAUST file.

The complete list of the LaTeX packages included by FAUST can be found in the file `architecture/latexheader.tex`.

6.5.3 An example of manual mathdoc

```
declare name        "Noise";
declare version     "1.1";
declare author      "Grame";
declare license     "BSD";
declare copyright   "(c)GRAME 2009";

//-----------------------------------------------------
// Demo noise generator for the Faust math documentation
//-----------------------------------------------------

<mdoc>
\section{Presentation of the "noise.dsp" Faust program}
This program describes a white noise generator with an interactive
    volume, using a random function.

\subsection{The random function}
The \texttt{random} function describes a generator of random
    numbers, which equation follows. You should notice hereby the
    use of an integer arithmetic on 32 bits, relying on integer
    wrapping for big numbers.
<equation>random</equation>

\subsection{The noise function}
The white noise then corresponds to:
<equation>noise</equation>
</mdoc>

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

<mdoc>
\subsection{Just add a user interface element to play volume!}
Endly, the sound level of this program is controlled by a user
    slider, which gives the following equation:
<equation>process</equation>
</mdoc>
```

```
<mdoc>
\section{Block-diagram schema of process}
This process is illustrated on figure 1.
<diagram>process</diagram>
</mdoc>

process = noise * vslider("Volume[style:knob]", 0, 0, 1, 0.1);

<mdoc>
\section{Notice of this documentation}
You might be careful of certain information and naming conventions
    used in this documentation:
<notice>

\section{Listing of the input code}
The following listing shows the input Faust code, parsed to compile
     this mathematical documentation.
<listing>
</mdoc>
```

The next page gather the four resulting pages of `noise.pdf`, in small size, to give an idea.

6.5.4 The -stripmdoc option

As you can see on the resulting file `noise.pdf` on its pages 3 and 4, the listing of the input code (section 4) contains all the mathdoc text (here coloured in grey). As it may be unneeded for certain uses (see Goals, section 6.1), we provide an option to strip mathdoc contents directly at compilation stage: `-stripmdoc` (short) or `--strip-mdoc-tags` (long).

6.6 Localization of mathdoc results

By default, texts used by the documentator are in English, but you can specify another language (French and Italian for the moment), using the `-mdlang` (or `--mathdoc-lang`) option with a two-letters argument (`en`, `fr`, `it`, etc.).

If you'd like to contribute to the localization effort, feel free to translate the mathdoc texts from any of the `mathdoctexts-*.txt` files, that are in the `architecture` directory (`mathdoctexts-fr.txt`, `mathdoctexts-it.txt`, etc.).

6.7 Summary of the mathdoc generation steps

1. first, simply call `faust2mathdoc myfaustfile.dsp` to get the full mathematical documentation stuff done on your faust file,
2. then simply open the pdf file `myfaustfile-mdoc/pdf/myfaustfile.pdf`.

Noise

Grame

December 14, 2009

name	Noise
version	1.1
author	Grame
license	BSD
copyright	(c)GRAME 2009

1 Presentation of the "noise.dsp" Faust program

This program describes a white noise generator with an interactive volume, using a random function.

1.1 The random function

The `random` function describes a generator of random numbers, which equation follows. You should notice hereby the use of an integer arithmetic on 32 bits, relying on integer wrapping for big numbers.

1. Input signal: none
2. Output signal:
$$y(t) = r_1(t)$$
3. Internal signal:
$$r_1(t) = 12345 \oplus 1103515245 \odot r_1(t-1)$$

1.2 The noise function

The white noise then corresponds to:

1. Input signal: none
2. Output signal:
$$y(t) = 4.65661 \cdot 10^{-10} \cdot r_1(t)$$

1.3 Just add a user interface element to play volume!

Endly, the sound level of this program is controlled by a user slider, which gives the following equation:

1. Input signal: none
2. Output signal:
$$y(t) = p_1(t) \cdot r_1(t)$$
3. User interface element:
$$\text{"Volume"}: \quad u_{s1}(t) \quad \in \quad [0,1] \quad \text{(default value} = 0)$$
4. Parameter signal:
$$p_1(t) = 4.65661 \cdot 10^{-10} \cdot u_{s1}(t)$$

2 Block-diagram schema of process

This process is illustrated on figure 1.

3 Notice of this documentation

You might be careful of certain information and naming conventions used in this documentation:

- This documentation was generated with Faust version 0.9.9.6b15mdoc, on December 14, 2009.
- Eventual sub-block-diagrams may be found in the "svg" sub-directory (only top-level block-diagrams are represented in this documentation).
- Warning: symbolic names eventually used inside block-diagrams have NO direct relation with signal names used in formulas ("$x(t)$", "$y(t)$", ...). Moreover, the computation may be simplified and reorganized.
- $\forall\, s(t) \in \mathbb{S},\ s(t<0) = 0$.
- The middle dot operator "$\cdot$" denotes multiplication in formulas.
- The circled plus operator "$\oplus$" denotes an integer addition.
- The circled dot operator "$\odot$" denotes an integer multiplication.
- $y(t)$ denotes an output signal.
- $p_i(t)$ denote parameter signals (running at "block rate").
- $u_{si}(t)$ denote user interface signals of sliders.
- $r_i(t)$ denote recursive signals (delayed as $r_i(t-d)$).

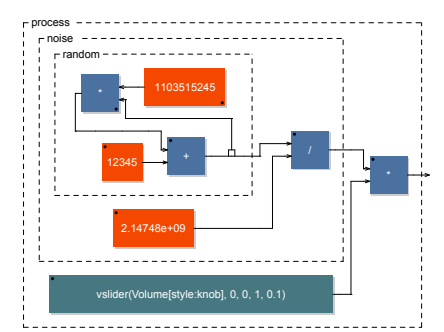


Figure 1: block-diagram of `process`

4 Listing of the input code

The following listing shows the input Faust code, parsed to compile this mathematical documentation.

Listing 1: `noise.dsp`

```
declare name      "Noise";
declare version   "1.1";
declare author    "Grame";
declare license   "BSD";
declare copyright  "(c)GRAME 2009";


//-----------------------------------------------------------------
// Noise generator and demo file for the Faust math documentation
//-----------------------------------------------------------------


<mdoc>
\section{Presentation of the "noise.dsp" Faust program}
This program describes a white noise generator with an interactive volume, using a random
     function.
```



```
\subsection{The random function}
The \texttt{random} function describes a generator of random numbers, which equation follows
     . You should notice hereby the use of an integer arithmetic on 32 bits, relying on
     integer wrapping for big numbers.
<equation>random</equation>

\subsection{The noise function}
The white noise then corresponds to:
<equation>noise</equation>
</mdoc>

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

<mdoc>
\subsection{Just add a user interface element to play volume!}
Endly, the sound level of this program is controlled by a user slider, which gives the
     following equation:
<equation>process</equation>
</mdoc>

<mdoc>
\section{Block-diagram schema of process}
This process is illustrated on figure 1.
<diagram>process</diagram>
</mdoc>

process = noise * vslider("Volume[style:knob]", 0, 0, 1, 0.1);

<mdoc>
\section{Notice of this documentation}
You might be careful of certain information and naming conventions used in this
     documentation:
<notice>

\section{Listing of the input code}
The following listing shows the input Faust code, parsed to compile this mathematical
     documentation.
<listing>
</mdoc>
```

Chapter 7

Acknowledgments

Many persons are contributing to the FAUST project, by providing code for the compiler, architecture files, libraries, examples, documentation, scripts, bug reports, ideas, etc. I would like in particular to thank:

- Fons Adriaensen
- Tiziano Bole
- Thomas Charbonnel
- Damien Cramet
- Étienne Gaudrin
- Albert Gräf
- Stefan Kersten
- Victor Lazzarini
- Matthieu Leberre
- Mathieu Leroi
- Rémy Muller
- Nicolas Scaringella
- Julius Smith

Many developments of the FAUST project are now taking place within the ASTREE project (ANR 2008 CORD 003 02). I would like to thank my ASTREE's partners:

- Jérôme Barthélemy (IRCAM)
- Alain Bonardi (IRCAM)
- Raffaele Ciavarella (IRCAM)

- Pierre Jouvelot (École des Mines/ParisTech)

- Laurent Pottier (U. Saint-Etienne)

as well as my colleagues at GRAME, in particular : Dominique Fober, Stéphane Letz and Karim Barkati.

I would like also to thank for their financial support:

- the French Ministry of Culture

- the Rhône-Alpes Region

- the City of Lyon

- the French National Research Agency (ANR)