Faust Quick Reference

Yann Orlarey
Grame, Centre National de Creation Musicale

October 2007

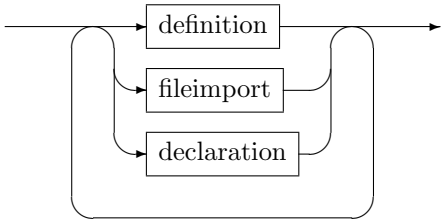
1 Introduction

This document is a quick-reference to the Faust language (version 0.9.9.2), a programming language for real-time signal processing and synthesis that targets high-performance signal processing applications and audio plug-ins.

2 Faust program

A Faust program describes a *signal processor* that transforms input signals into output signals. A Faust program is made of one or more source files. A source file is essentially a list of *definitions* with the possibility to recursively import definitions from other source files. Each definition associates an identifier (with an optional list of parameters or an optional list of patterns) with a *block-diagram* that it represents.

program

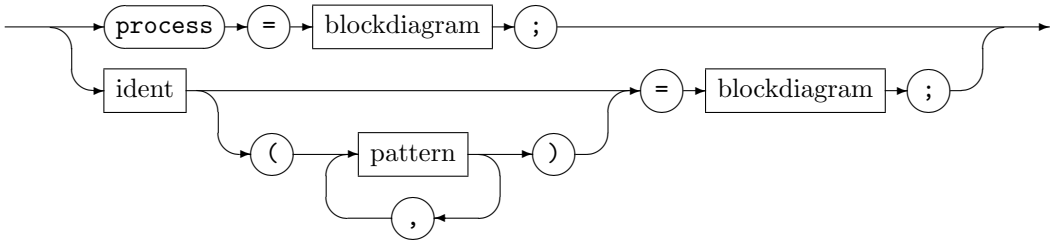


2.1 Definitions

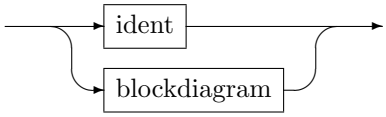
Faust is based on a declarative style of programming. A valid Faust program must contain at least one definition for the keyword *process* (the equivalent of *main* in C) . Definitions can appear in any order. In particular an identifier

can be used before being defined. Multiple definitions of the same identifier are treated as pattern-matching rules and analyzed as such.

definition



pattern



For example:

```
process  = +;
```

is a valid Faust program that add its two input signals to produce its output. Here is a more involved example of *noise* generetor:

```
random   = +(12345) ~ *(1103515245);
noise    = random/2147483647.0;
process  = noise * vslider("vol", 0, 0, 1, 0.1);
```

2.2 File Imports

File imports allow to add the definitions of another source file to the definitions of the current file. File imports can appear every where in a source file and in any order. Mutual recursive imports are allowed and handled correctly.

fileimport

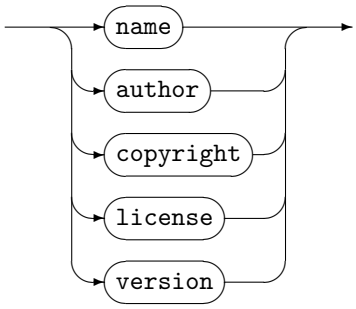
It is common for a Faust program to import the definitions of math.lib and music.lib files by including the lines : `import`("`math.lib`"); and `import`("`music.lib`");.

2.3 Declarations

Declarations can be used to define some meta-data documenting and describing the project. The currently implemented meta-data are the name, the author, the version, the copyright and the license of the project. This information is typically used when generating an XML description of the project (option -xml of the compiler).

declaration

property



Here is an example of declarations :

```
declare name "superFX";
declare author "Alonzo Church";
declare version "0.9.5c";
```

3 Block-Diagrams

Faust is a *block-diagram* language. Specific *composition operations* are used to "connect" two block-diagrams together in order to form a new one. For example the sequential composition operation (':') connect the outputs of the first block-diagram to the corresponding inputs of the second block-diagram. Five high-level composition operations are provided : *recursive composition*, *parallel composition*, *sequential composition*, *split composition* and *merge composition*. Moreover a block-diagram can have an associated set of local definitions.

Syntax	Pri.	Description
blockdiagram ~ *blockdiagram*	4	recursive composition
blockdiagram , *blockdiagram*	3	parallel composition
blockdiagram : *blockdiagram*	2	sequential composition
blockdiagram <: *blockdiagram*	1	split composition
blockdiagram :> *blockdiagram*	1	merge composition
blockdiagram with { *definition* ...}	0	local definitions
expression		block-diagrams are made of expressions

All these composition operations are left associative. Based on these associativity and priority rules the block-diagram : $\texttt{A} : \texttt{B}, \texttt{C} \sim \texttt{D}, \texttt{E} :> \texttt{F}$ should be interpreted as: $(\texttt{A} : ((\texttt{B}, (\texttt{C} \sim \texttt{D})), \texttt{E})) :> \texttt{F}$.

4 Expressions

Faust *Expressions* provide *syntactic sugar* allowing traditional infix notation and function calls. For example instead of : $2, \texttt{A} : *, \texttt{B} : +$ one can write the infix expression : $2 * \texttt{A} + \texttt{B}$. Or instead of : $\texttt{A} : \sin$ one can use the function call notation : $\sin(\texttt{A})$.

Syntax	Pri.	Description
expression(*arg*,...)	10	function call
expression.*ident*	10	access to lexical environment
expression'	9	one sample delay
expression @ *expression*	8	fixed delay
expression * *expression*	7	multiplication
expression / *expression*	7	division
expression % *expression*	7	modulo
expression & *expression*	7	logical and
expression ∧ *expression*	7	logical xor
expression << *expression*	7	arithmetic left shift
expression >> *expression*	7	arithmetic right shift
expression + *expression*	6	addition
expression - *expression*	6	substraction
expression | *expression*	6	logical or
expression < *expression*	5	less than
expression <= *expression*	5	less or equal
expression > *expression*	5	greater than
expression >= *expression*	5	greater or equal
expression == *expression*	5	equal
expression != *expression*	5	not equal
primitive		expressions are made of primitives

Binary operators can also be used in function call notation. For example $+(2, \mathtt{A})$ is equivalent to $2 + \mathtt{A}$. Moreover partial applications are allowed like in $*(3)$.

5 Primitive Signal Processing Operations

The primitive signal processing operations represent the built-in functionalities of Faust, that is the atomic operations provided by the language. All these primitives (and the block-diagrams build on top of them) denote *signal processors*, functions transforming *input signals* into *output signals*. Let's define more precisely what a *signal processor* is.

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 $s(t)$. We denote by $\mathbb{S}$ the set of all possible signals : $\mathbb{S} = \mathbb{N} \to \mathbb{R}$. 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 $p : \mathbb{S}^n \to \mathbb{S}^m$. We notate $\mathbb{P}$ the set of all signal processors : $\mathbb{P} = \bigcup_{n,m} \mathbb{S}^n \to \mathbb{S}^m$.

All primitives and block-diagram expressed in Faust are members of $\mathbb{P}$ (i.e. signal processors) including numbers. For example number 3.14 doesn't represent neither a sample, nor a signal, but a *signal processor* : $\mathbb{S}^0 \to \mathbb{S}^1$ that transforms the empty tuple $()$ into a 1-tuple of signals (s) such that $\forall t \in \mathbb{N}, s(t) = 3.14$.

5.1 C-equivalent primitives

Most Faust primitives are analogue to their C counterpart but lifted to signal processing. For example $+$ is a function of type $\mathbb{S}^2 \to \mathbb{S}^1$ 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	$\mathbb{S}^0 \rightarrow \mathbb{S}^1$	integer number: $y(t) = n$
$n.m$	$\mathbb{S}^0 \rightarrow \mathbb{S}^1$	floating point number: $y(t) = n.m$
`_`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	identity function: $y(t) = x(t)$
`!`	$\mathbb{S}^1 \rightarrow \mathbb{S}^0$	cut function: $\forall x \in \mathbb{S}, (x) \rightarrow ()$
`int`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	cast into an int signal: $y(t) = (int)x(t)$
`float`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	cast into an float signal: $y(t) = (float)x(t)$
`+`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	addition: $y(t) = x_1(t) + x_2(t)$
`-`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	subtraction: $y(t) = x_1(t) - x_2(t)$
`*`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	multiplication: $y(t) = x_1(t) * x_2(t)$
`/`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	division: $y(t) = x_1(t)/x_2(t)$
`%`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	modulo: $y(t) = x_1(t)\%x_2(t)$
`&`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	logical AND: $y(t) = x_1(t)\&x_2(t)$
`|`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	logical OR: $y(t) = x_1(t)\|x_2(t)$
$\wedge$	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	logical XOR: $y(t) = x_1(t) \wedge x_2(t)$
`<<`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	arith. shift left: $y(t) = x_1(t) << x_2(t)$
`>>`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	arith. shift right: $y(t) = x_1(t) >> x_2(t)$
`<`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	less than: $y(t) = x_1(t) < x_2(t)$
`<=`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	less or equal: $y(t) = x_1(t) <= x_2(t)$
`>`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	greater than: $y(t) = x_1(t) > x_2(t)$
`>=`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	greater or equal: $y(t) = x_1(t) >= x_2(t)$
`==`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	equal: $y(t) = x_1(t) == x_2(t)$
`!=`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	different: $y(t) = x_1(t)! = x_2(t)$

5.2 `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: $y(t) = \text{rintf}(x(t))$

5.3 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 1).

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: $T[] = \{x_0(t), x_1(t), x_2(t)\}; y(t) = T[s(t)]$

5.4 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 toolk-

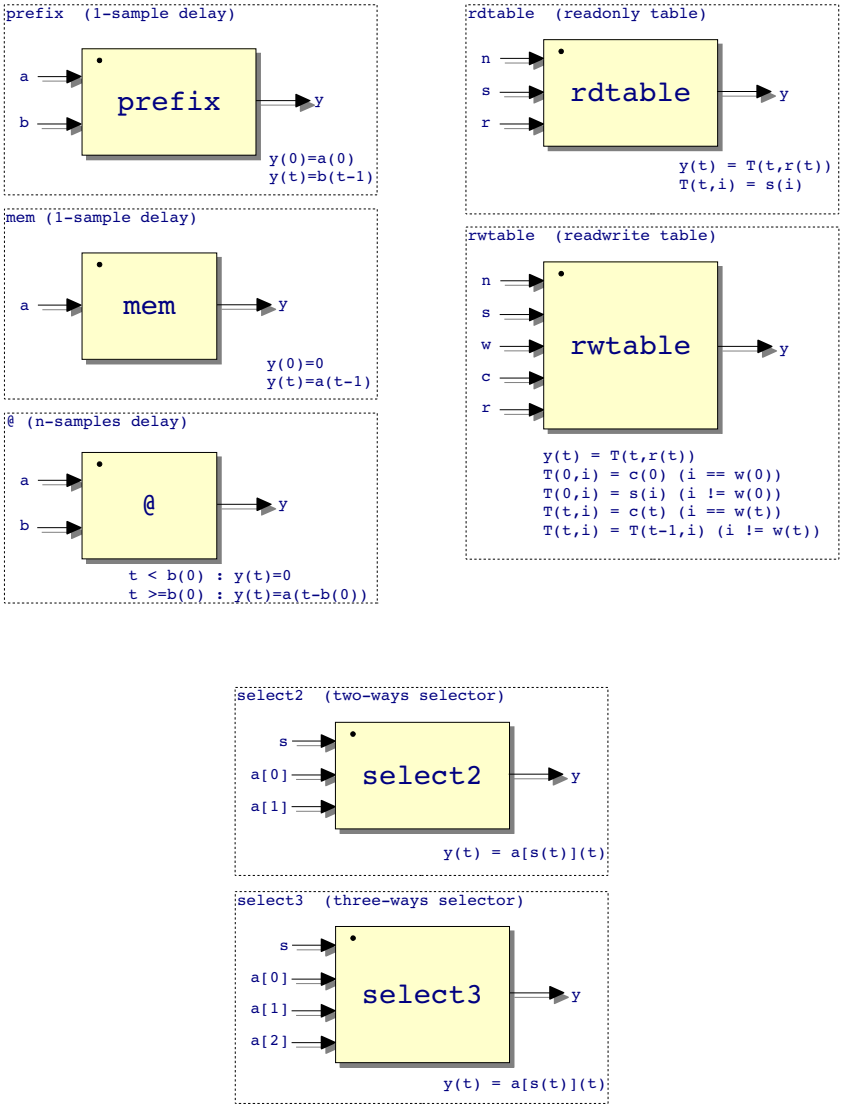


Figure 1: Delays, tables and selectors primitives

its. 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 2) produces a signal which is 1 when the button is pressed and 0 otherwise. These signals can be freely combined with other audio signals.

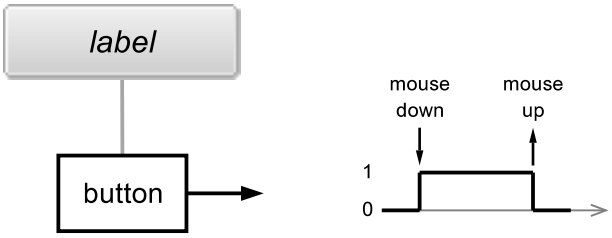


Figure 2: User Interface Button

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)`

note : The *str* string used in widgets can contain variable parts. These variable parts are indicated by the sign '`%`' followed by the name of a 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)`.

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 (%)"`.

5.5 The Foreign Function Mechanism

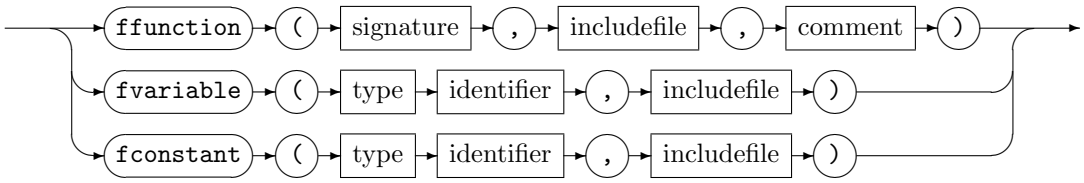
External C functions, variables and constants can be introduced using the foreign function mechanism. An external C function is declared by indicating its

name and signature as well as the required include file. External variables and constants can also be declared with a similar syntax.

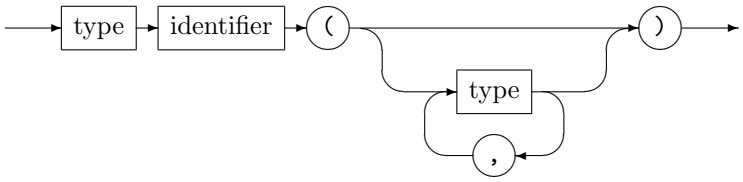
Variable are considered to vary at block speed. This means that expressions depending of external variables are computed every block, while expressions depending external constants are computed only once at initialization. There is currently no external signals that would be computed every sample.

The syntax of these foreign declarations is the following :

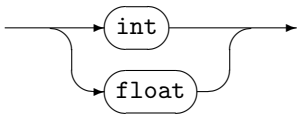
foreign



signature



type



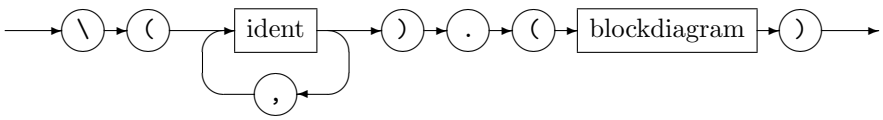
The file `math.lib` included in the Faust package defines most of the standard mathematical function of `<math.h>` (that are not already builtins) using the foreign function mechanism. Here is the list of these functions :

Name	Definition
SR	fconstant(int fSamplingFreq, <math.h>)
PI	3.1415926535897932385
cbrt	ffunction(float cbrt (float), <math.h>,"")
hypot	ffunction(float hypot (float, float), <math.h>,"")
ldexp	ffunction(float ldexp (float, int), <math.h>,"")
scalb	ffunction(float scalb (float, float), <math.h>,"")
log1p	ffunction(float log1p (float), <math.h>,"")
logb	ffunction(float logb (float), <math.h>,"")
ilogb	ffunction(int ilogb (float), <math.h>,"")
expm1	ffunction(float expm1 (float), <math.h>,"")
acosh	ffunction(float acosh (float), <math.h>, "")
asinh	ffunction(float asinh (float), <math.h>, "")
atanh	ffunction(float atanh (float), <math.h>, "")
sinh	ffunction(float sinh (float), <math.h>, "")
cosh	ffunction(float cosh (float), <math.h>, "")
tanh	ffunction(float tanh (float), <math.h>,"")
erf	ffunction(float erf(float), <math.h>,"")
erfc	ffunction(float erfc(float), <math.h>,"")
gamma	ffunction(float gamma(float), <math.h>,"")
J0	ffunction(float j0(float), <math.h>,"")
J1	ffunction(float j1(float), <math.h>,"")
Jn	ffunction(float jn(int, float), <math.h>,"")
lgamma	ffunction(float lgamma(float), <math.h>,"")
Y0	ffunction(float y0(float), <math.h>,"")
Y1	ffunction(float y1(float), <math.h>,"")
Yn	ffunction(float yn(int, float), <math.h>,"")
isnan	ffunction(int isnan (float),<math.h>,"")
nextafter	ffunction(float nextafter(float, float),<math.h>,"")

5.6 Programming constructions

Faust provides several programming constructions that give powerful ways to describe and transform block-diagrams algorithmically.

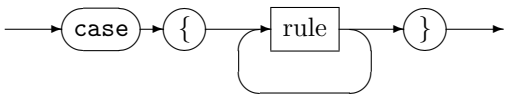
Abstraction



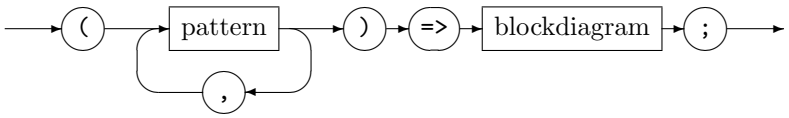
Abstractions allow to define anonymous functions like for example a square function : $\backslash(x).(x * x)$. Unapplied abstractions can be used for symbolic routing

of signals. For example : $\backslash(x, y).(y, x)$ denotes a signal processor that exchanges its two inputs.

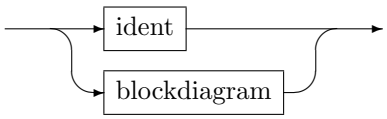
Case



Rule

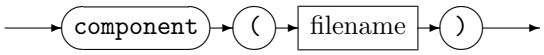


Pattern



case pattern matching rules provide an effective way to analyze and transform block-diagrams algorithmically. For example `case{ (x:y) => y:x; (x) => x; }` will invert the two parts of a sequential block-diagram and leave untouched any other construction. Please note that patterns are evaluated before the pattern matching operation. Thefore only variables that appear free in the pattern are binding variables during pattern matching.

Component

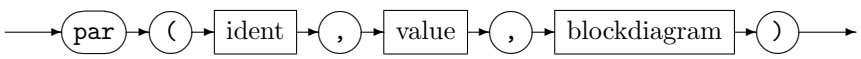


The *component* construction allows a very modular design. A whole Faust program can be included in another Faust program as a simple expression. For example `component("freeverb.dsp")` denotes the process defined in the file *"freeverb.dsp"*, and

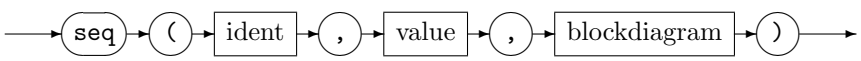
```
process = component("karplus32.dsp"):component("freeverb.dsp");
```

is a process that combines two components in sequence.

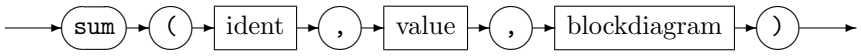
par



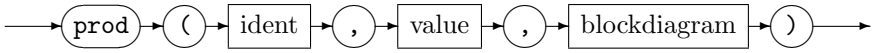
seq



sum



prod



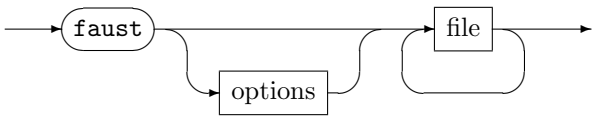
The `par`, `seq`, `sum` and `prod` constructions allow algorithmic descriptions of block-diagrams. For example :

`par(i,8,E(i))`	is equivalent to	`E(0),E(1),...,E(7)`
`seq(i,8,E(i))`	is equivalent to	`E(0):E(1):...:E(7)`
`sum(i,8,E(i))`	is equivalent to	`E(0)+E(1)+...+E(7)`
`prod(i,8,E(i))`	is equivalent to	`E(0)*E(1)*...*E(7)`

6 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



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
`-f` *n*	`--fold` *n*	max complexity of svg diagrams before splitting into several files (default 25 boxes)
`-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
`-lb`	`--left-balanced`	generate left-balanced expressions
`-mb`	`--mid-balanced`	generate mid-balanced expressions (default)
`-rb`	`--right-balanced`	generate right-balanced expressions
`-mcd` *n*	`--max-copy-delay` *n*	threshold between copy and ring buffer delays (default 16 samples)
`-a` *file*		C++ wrapper file
`-o` *file*		C++ output file

The main available architecture files are :

File name	Description
`max-msp.cpp`	Max/MSP plugin
`vst.cpp`	VST plugin
`jack-gtk.cpp`	Jack GTK full application
`jack-qt.cpp`	Jack QT full application
`alsa-gtk.cpp`	Alsa GTK full application
`alsa-qt.cpp`	Alsa QT full application
`oss-gtk.cpp`	OSS GTK full application
`ladspa.cpp`	LADSPA plugin
`q.cpp`	Q language plugin
`supercollider.cpp`	SuperCollider Unit Generator
`sndfile.cpp`	sound file transformation command
`bench.cpp`	speed benchmark

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`.