Switched-on Yampa: Programming Modular Synthesizers in Haskell

MGS Christmas Seminar 2007

Henrik Nilsson and George Giorgidze

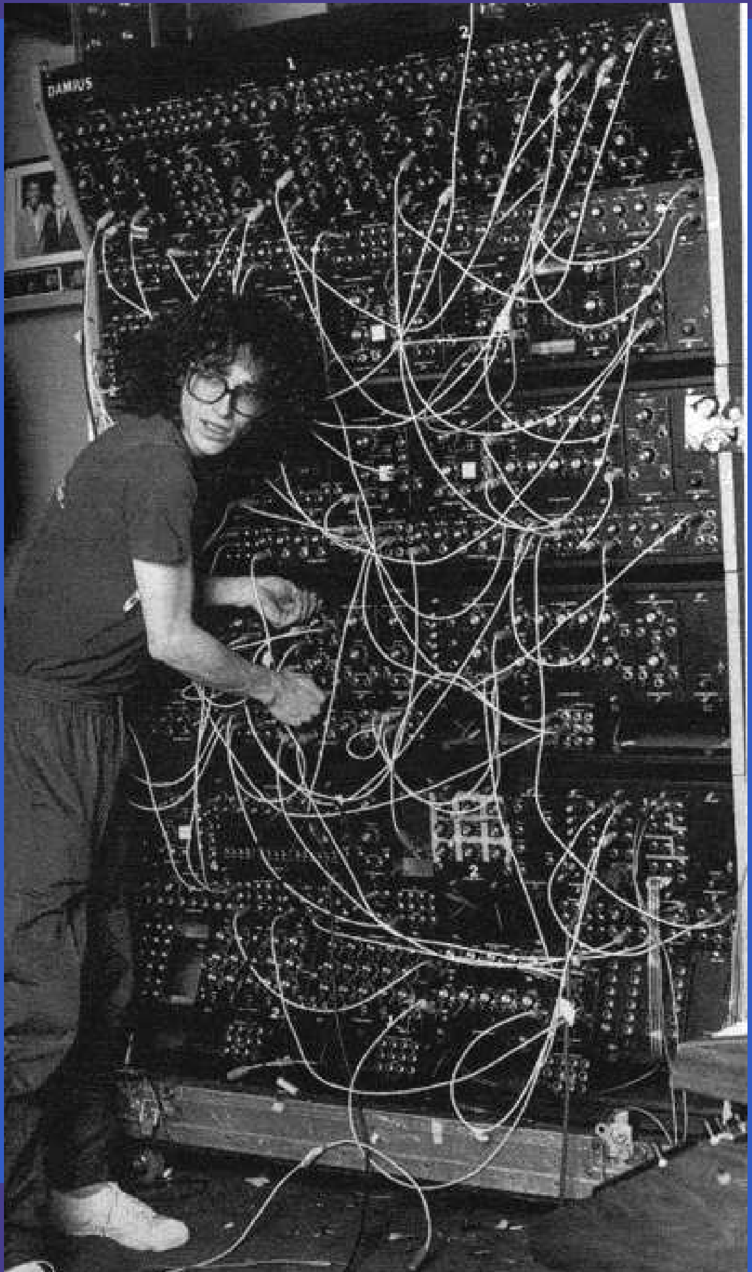
School of Computer Science

The University of Nottingham, UK

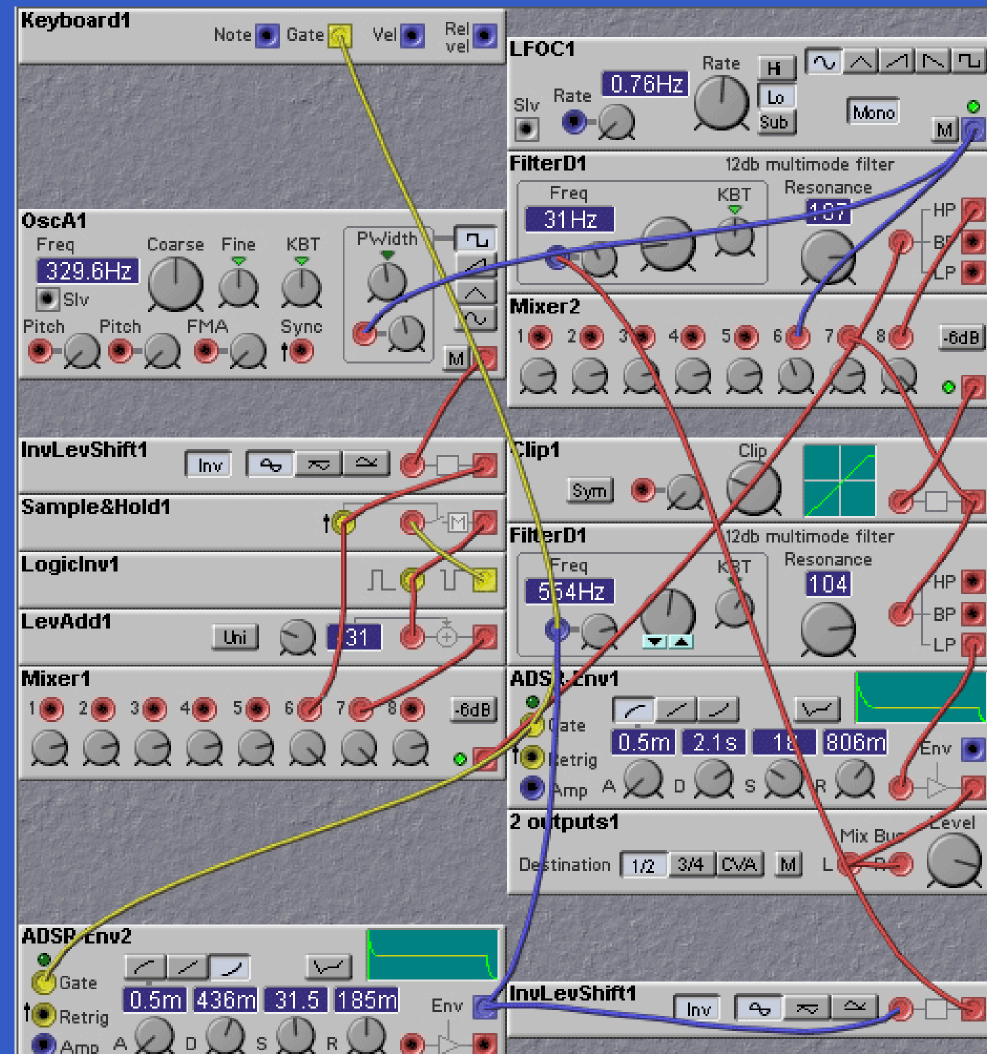
Modular synthesizers?

Modular synthesizers?

Modular synthesizers?



Modern Modular Synthesizers



Yampa?

Yampa?

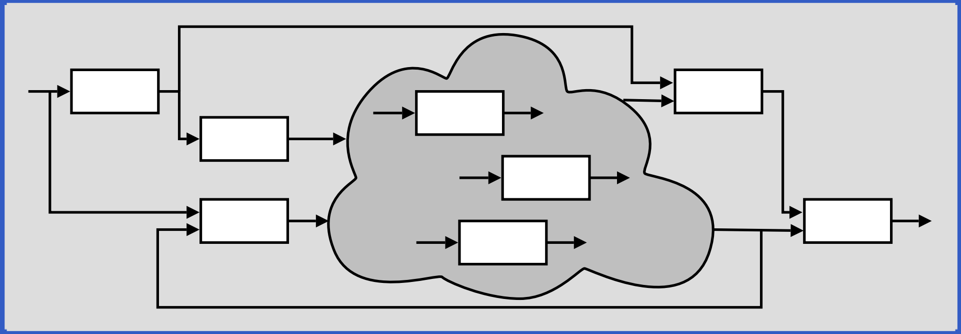
- Domain-specific language embedded in Haskell for programming ***hybrid*** (mixed discrete- and continuous-time) systems.

Yampa?

- Domain-specific language embedded in Haskell for programming ***hybrid*** (mixed discrete- and continuous-time) systems.
- Key concepts:
  - ***Signals***: time-varying values
  - ***Signal Functions***: functions on signals
  - ***Switching*** between signal functions

Yampa?

- Domain-specific language embedded in Haskell for programming ***hybrid*** (mixed discrete- and continuous-time) systems.
- Key concepts:
  - ***Signals***: time-varying values
  - ***Signal Functions***: functions on signals
  - ***Switching*** between signal functions
- Programming model:



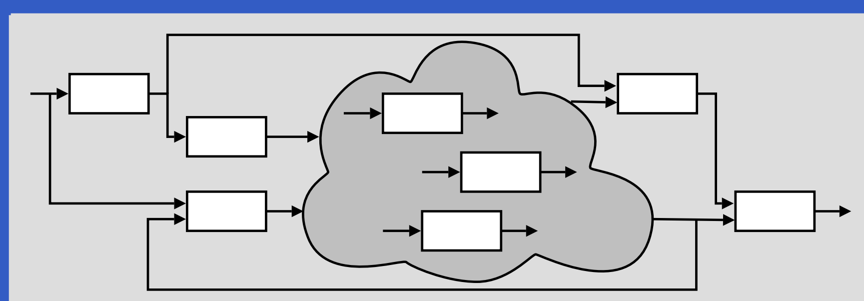
What is the point?

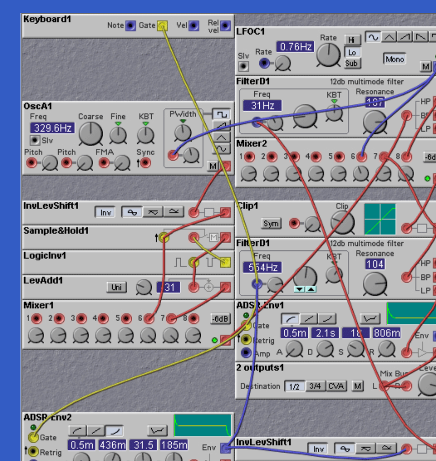
What is the point?

- Music can be seen as a hybrid phenomenon. Thus interesting to explore a hybrid approach to programming music and musical applications.

What is the point?

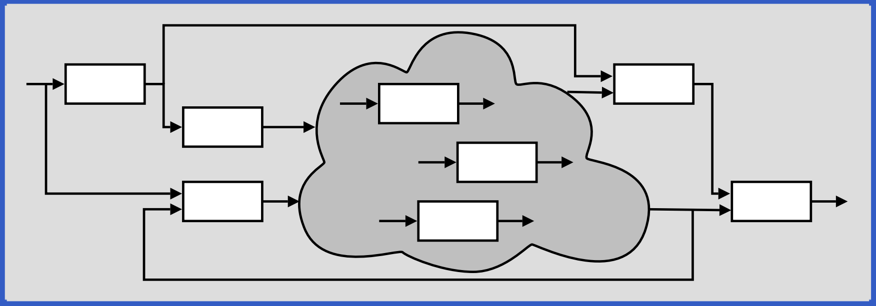
- Music can be seen as a hybrid phenomenon. Thus interesting to explore a hybrid approach to programming music and musical applications.
- Yampa’s programming model is very reminiscent of programming modular synthesizers:

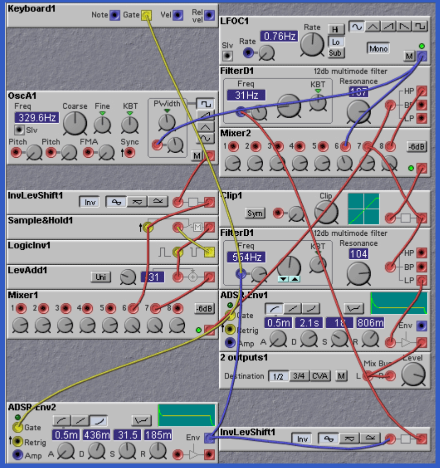




What is the point?

- Music can be seen as a hybrid phenomenon. Thus interesting to explore a hybrid approach to programming music and musical applications.
- Yampa's programming model is very reminiscent of programming modular synthesizers:





- Fun application! Useful for teaching?

What have we done?

What have we done?

Framework for programming modular synthesizers in Yampa:

What have we done?

Framework for programming modular synthesizers in Yampa:

- Sound-generating and sound-shaping modules

What have we done?

Framework for programming modular synthesizers in Yampa:

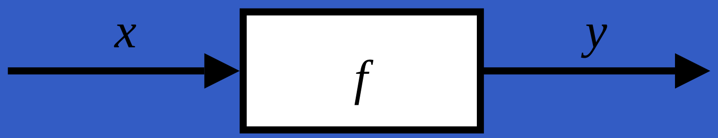
- Sound-generating and sound-shaping modules
- Additional supporting infrastructure:
  - Input: MIDI files (musical scores), keyboard
  - Output: audio files (.wav), sound card
  - Reading SoundFont files (instrument definitions)

What have we done?

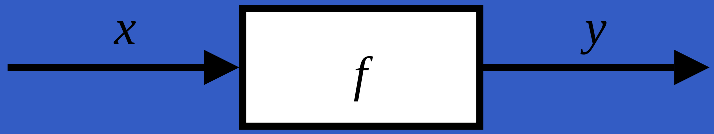
Framework for programming modular synthesizers in Yampa:

- Sound-generating and sound-shaping modules
- Additional supporting infrastructure:
  - Input: MIDI files (musical scores), keyboard
  - Output: audio files (.wav), sound card
  - Reading SoundFont files (instrument definitions)
- Status: proof-of-concept, but decent performance.

Yampa: Signal functions

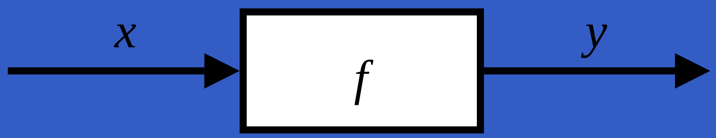


Yampa: Signal functions



Intuition:

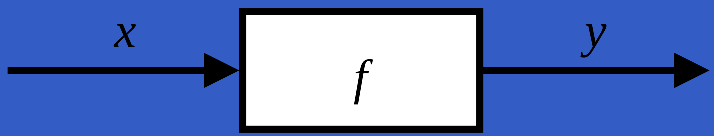
Yampa: Signal functions



Intuition:

$$Time \approx \mathbb{R}$$

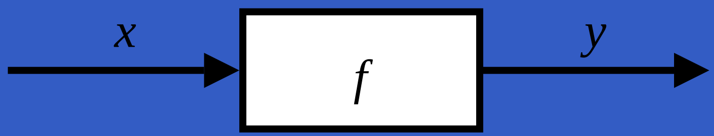
Yampa: Signal functions



Intuition:

$$Time \approx \mathbb{R}$$
$$Signal\ a \approx Time \rightarrow a$$
$$x :: Signal\ T1$$
$$y :: Signal\ T2$$

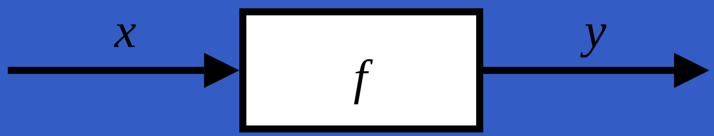
Yampa: Signal functions



Intuition:

$$Time \approx \mathbb{R}$$
$$Signal\ a \approx Time \to a$$
$$x :: Signal\ T1$$
$$y :: Signal\ T2$$
$$SF\ a\ b \approx Signal\ a \to Signal\ b$$
$$f :: SF\ T1\ T2$$

Yampa: Signal functions



Intuition:

$$Time \approx \mathbb{R}$$
$$Signal\ a \approx Time \rightarrow a$$
$$x :: Signal\ T1$$
$$y :: Signal\ T2$$
$$SF\ a\ b \approx Signal\ a \rightarrow Signal\ b$$
$$f :: SF\ T1\ T2$$

Additionally, ***causality*** required: output at time t must be determined by input on interval $[0, t]$.

Yampa: Related languages

FRP/Yampa related to:

- Synchronous dataflow languages, like Esterel, Lucid Synchrone.
- Modeling languages, like Simulink, Modelica.

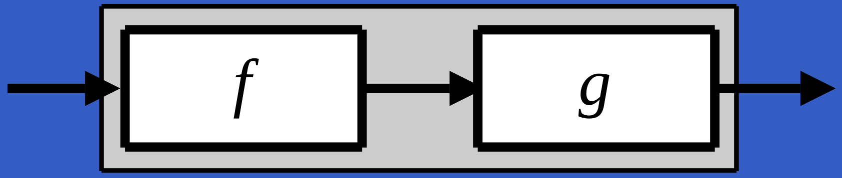
Yampa: Programming (1)

In Yampa, systems are described by combining signal functions (forming new signal functions).

Yampa: Programming (1)

In Yampa, systems are described by combining signal functions (forming new signal functions).

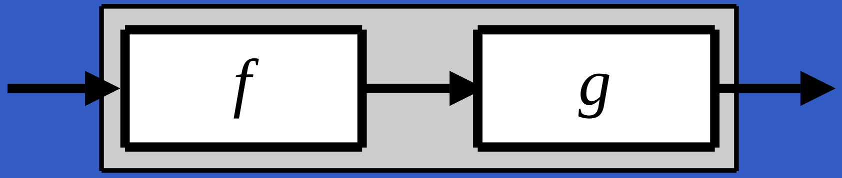
For example, serial composition:



Yampa: Programming (1)

In Yampa, systems are described by combining signal functions (forming new signal functions).

For example, serial composition:

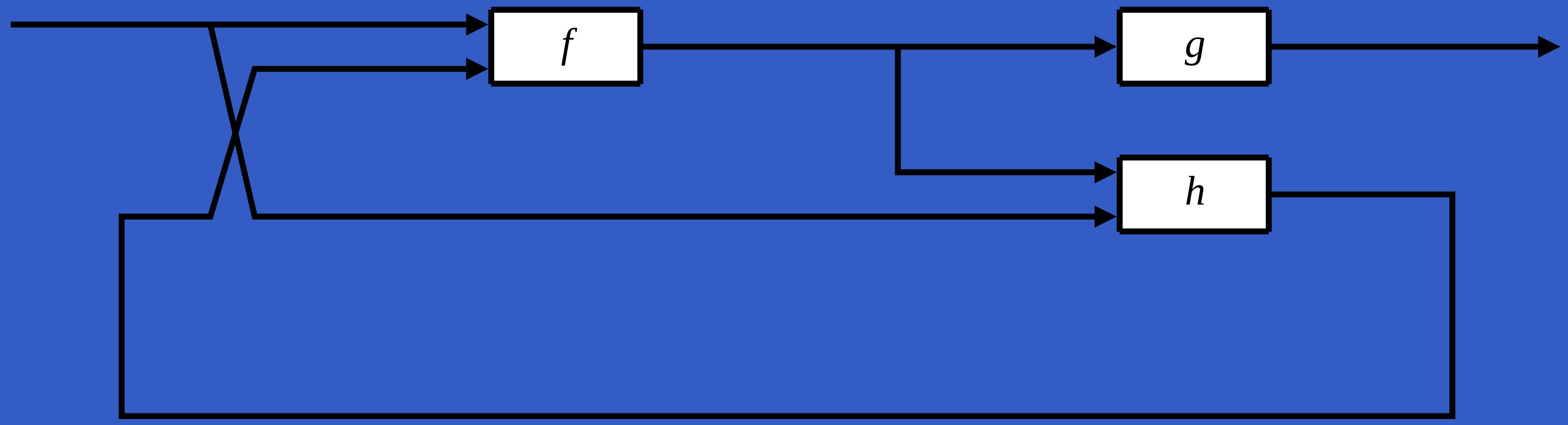


A *combinator* can be defined that captures this idea:

$$(\ggg) :: SF\ a\ b \rightarrow SF\ b\ c \rightarrow SF\ a\ c$$

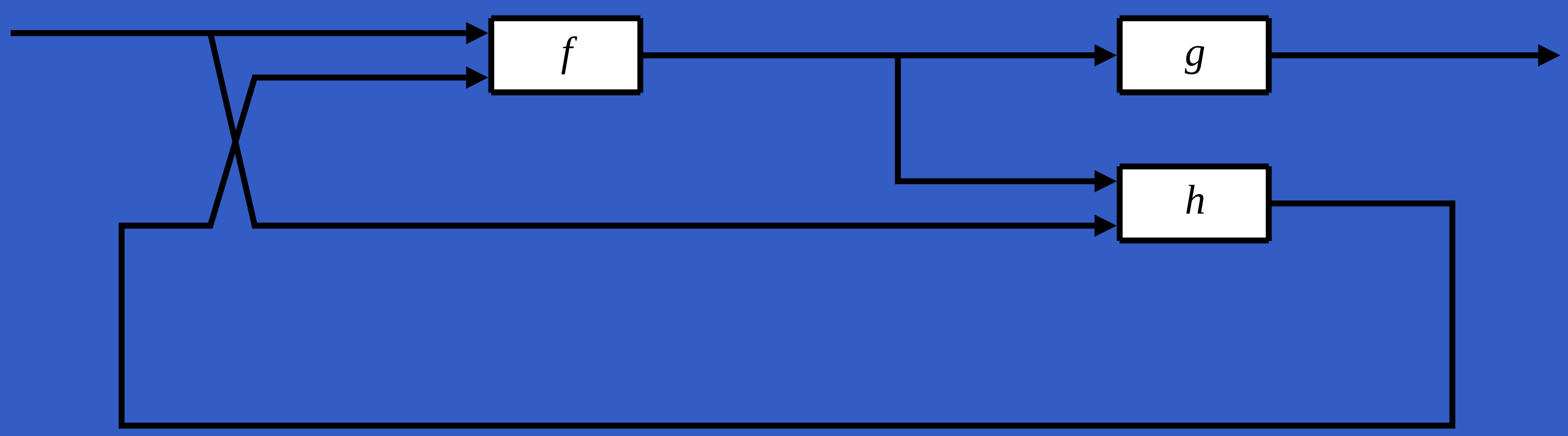
Yampa: Programming (2)

What about larger networks?
How many combinators are needed?



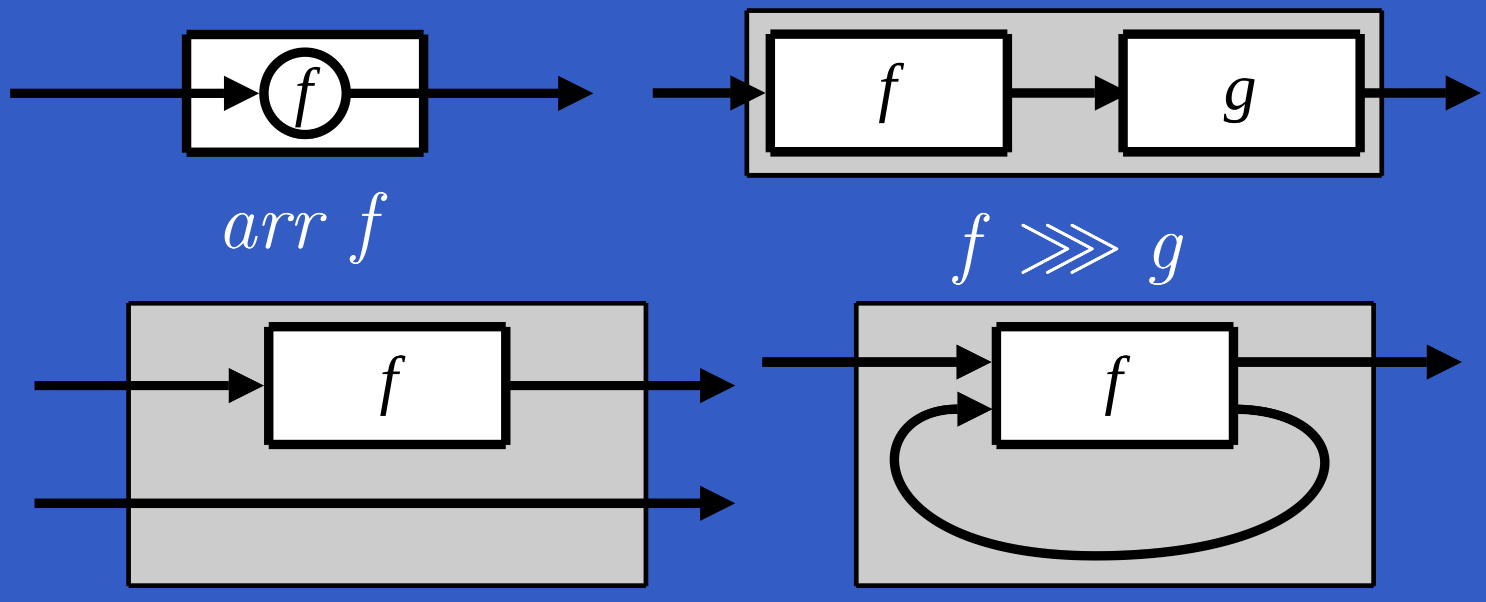
Yampa: Programming (2)

What about larger networks?
How many combinators are needed?



John Hughes's ***Arrow*** framework provides a good answer!

Yampa: The Arrow framework (1)



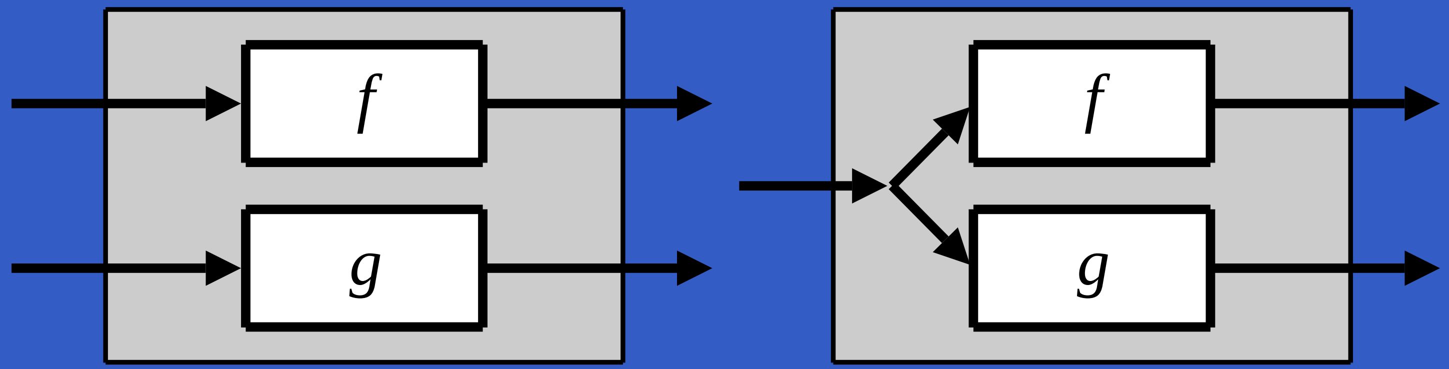
$arr\ f$ $f \ggg g$

$first\ f$ $loop\ f$

$$
\begin{aligned}
arr &:: (a \rightarrow b) \rightarrow SF\ a\ b \\
(\ggg) &:: SF\ a\ b \rightarrow SF\ b\ c \rightarrow SF\ a\ c \\
first &:: SF\ a\ b \rightarrow SF\ (a,c)\ (b,c) \\
loop &:: SF\ (a,c)\ (b,c) \rightarrow SF\ a\ b
\end{aligned}
$$

Yampa: The Arrow framework (2)

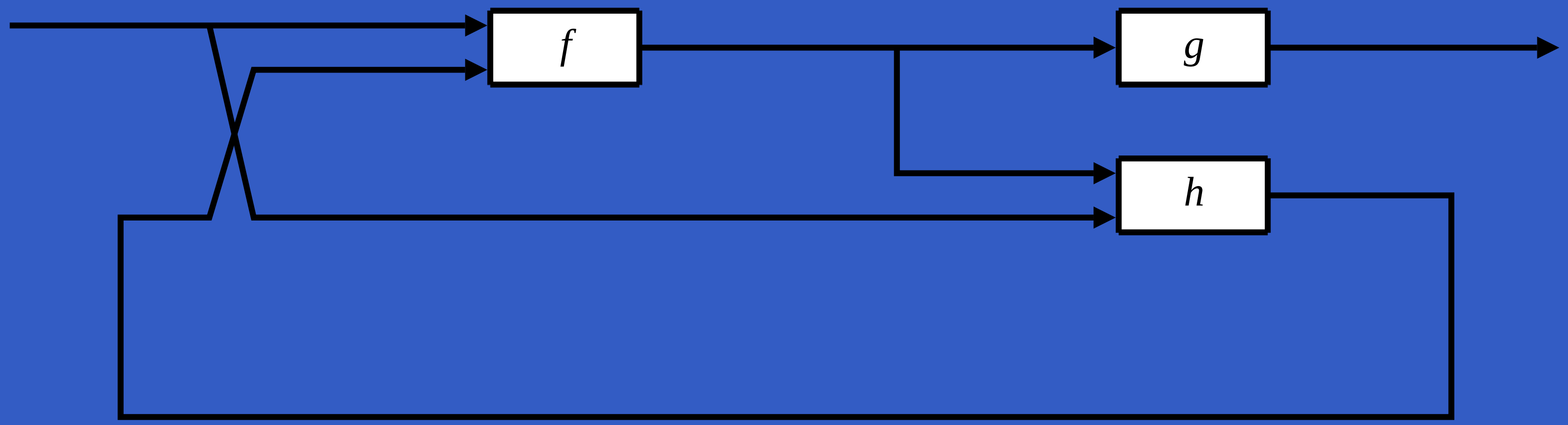
Some derived combinators:



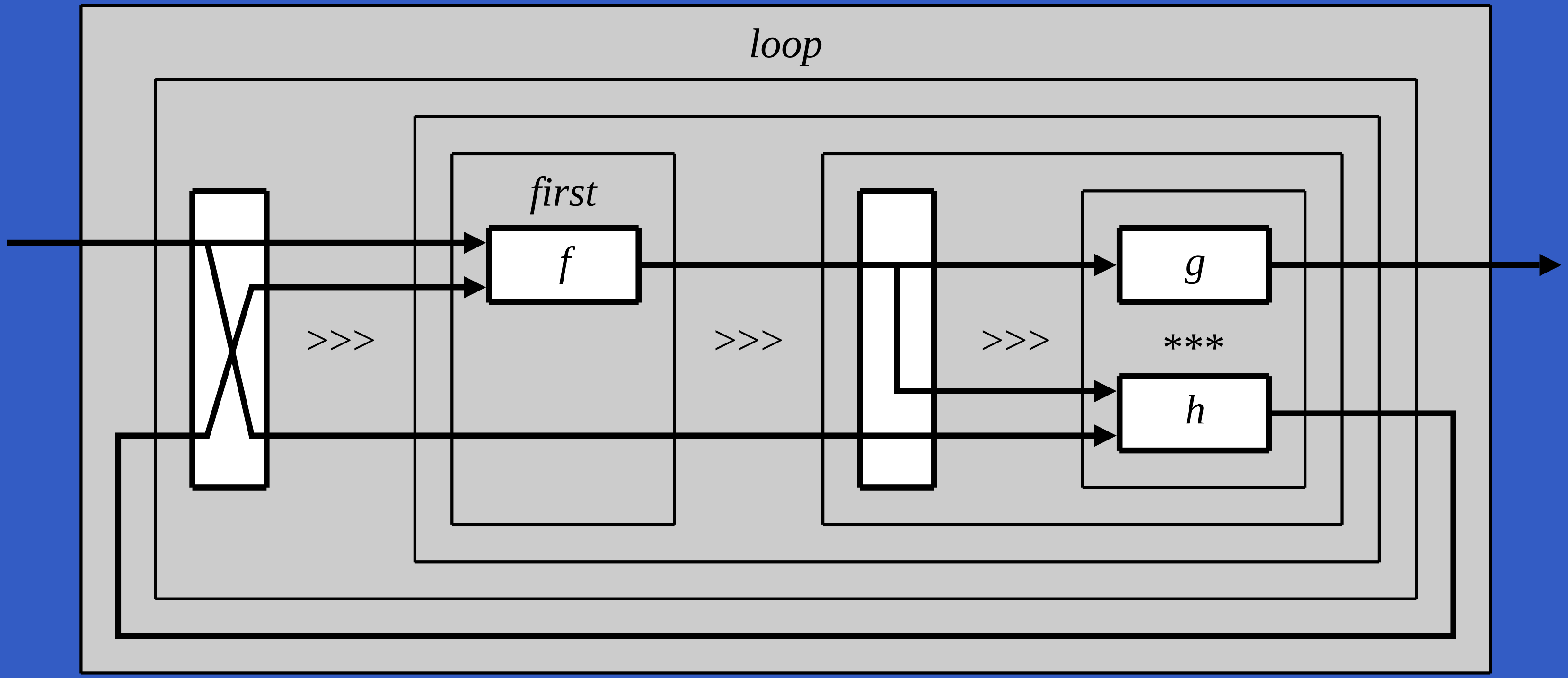
$f \ggg g$ $\qquad$ $f \mathbin{\&\&\&} g$

$$(\ast\ast\ast) :: SF\ a\ b \rightarrow SF\ c\ d \rightarrow SF\ (a, c)\ (b, d)$$
$$(\&\&\&) :: SF\ a\ b \rightarrow SF\ a\ c \rightarrow SF\ a\ (b, c)$$

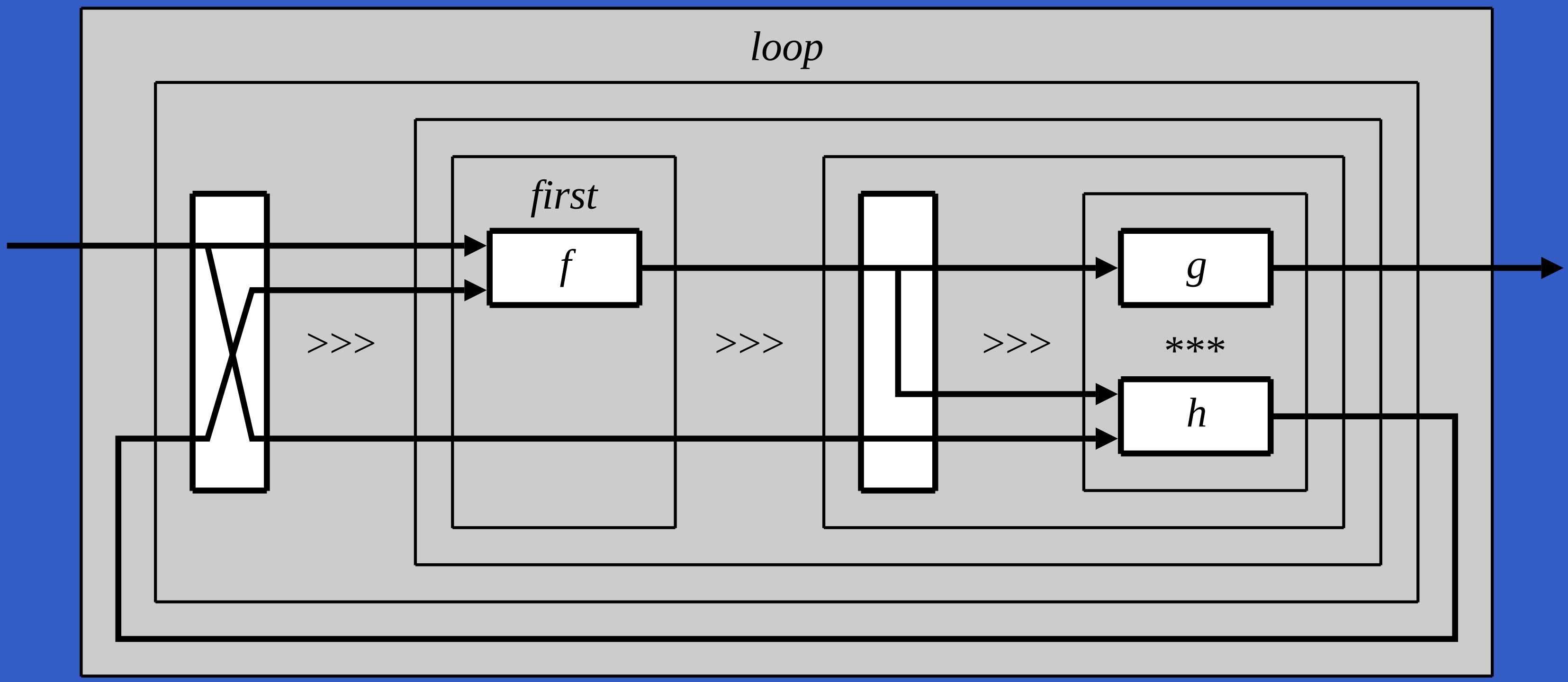
Yampa: Constructing a network



Yampa: Constructing a network

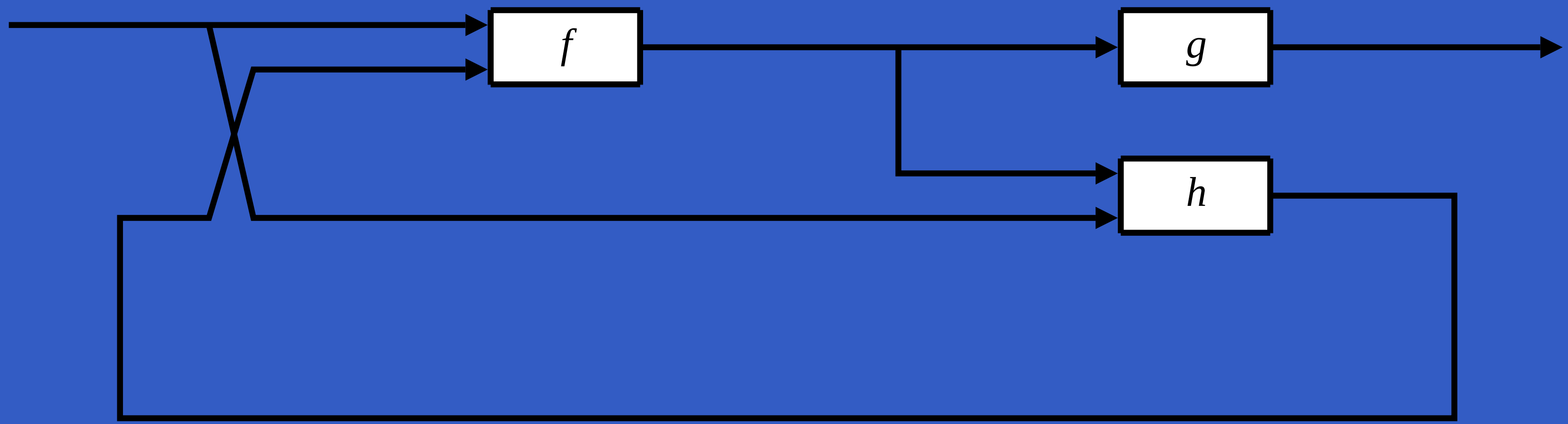


Yampa: Constructing a network

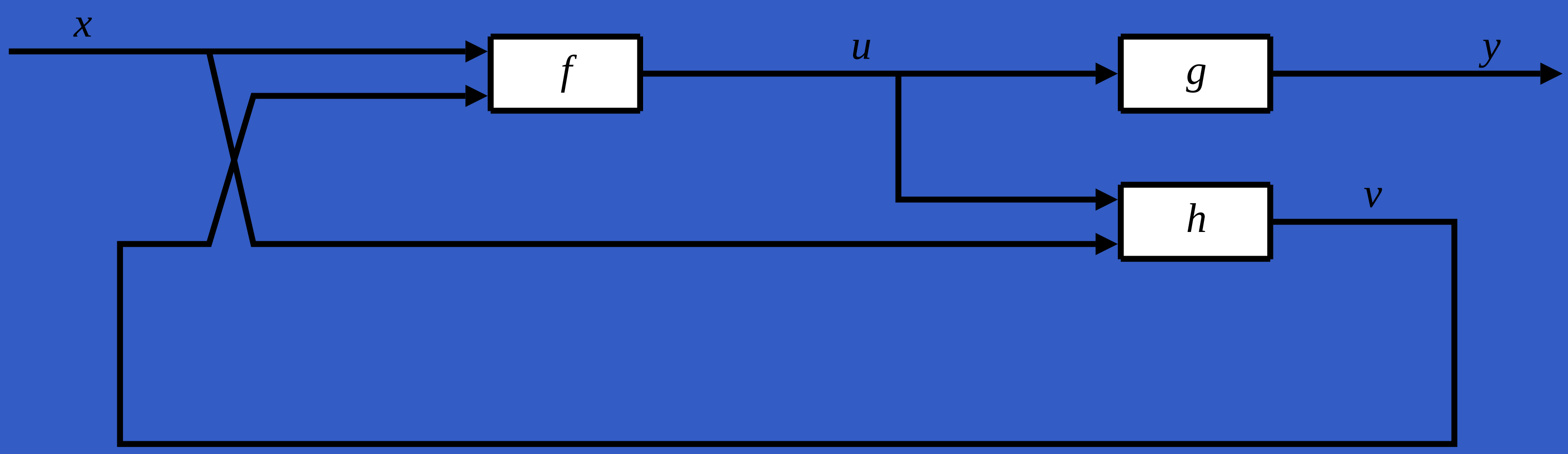


$$loop\ (arr\ (\lambda(x,y) \rightarrow ((x,y),x))$$
$$\ggg (first\ f$$
$$\ggg (arr\ (\lambda(x,y) \rightarrow (x,(x,y))) \ggg (g \ast\!\!\ast\!\!\ast h))))$$

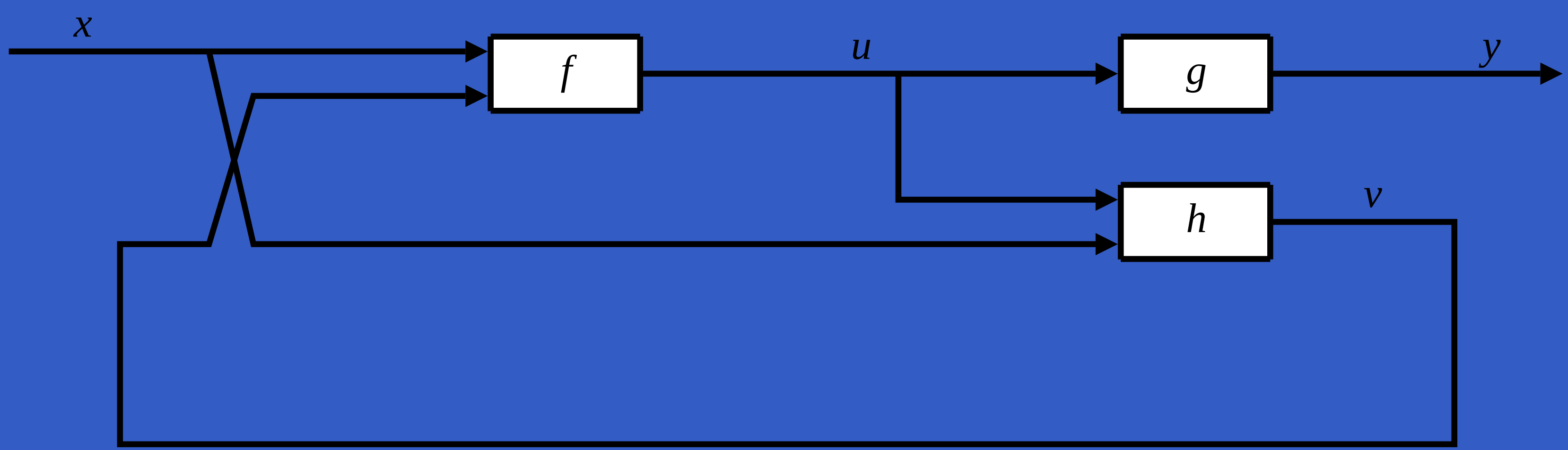
Yampa: Paterson's Arrow notation



Yampa: Paterson's Arrow notation



Yampa: Paterson's Arrow notation



$$
\begin{array}{l}
\textbf{proc}\ x \rightarrow \textbf{do} \\
\quad \textbf{rec} \\
\qquad u \leftarrow f \prec (x, v) \\
\qquad y \leftarrow g \prec u \\
\qquad v \leftarrow h \prec (u, x) \\
\quad returnA \prec y
\end{array}
$$

Yampa: Discrete-time signals

Yampa’s signals are conceptually ***continuous-time*** signals.

Yampa: Discrete-time signals

Yampa's signals are conceptually ***continuous-time*** signals.

Discrete-time signals: signals defined at discrete points in time.

Yampa: Discrete-time signals

Yampa's signals are conceptually ***continuous-time*** signals.

Discrete-time signals: signals defined at discrete points in time.

Yampa models discrete-time signals by lifting the ***co-domain*** of signals using an option-type:

$$\mathbf{data}\ Event\ a = NoEvent \mid Event\ a$$

Example:

$$repeatedly :: Time \rightarrow b \rightarrow SF\ a\ (Event\ b)$$

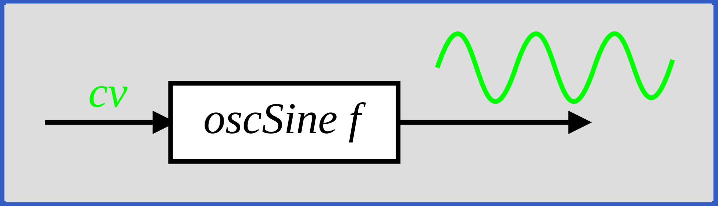
Yampa: Switching

The structure of a Yampa system may evolve over time. This is expressed through ***switching*** primitives.

Example:

$$
\begin{aligned}
\mathit{switch} :: \mathit{SF}\ a\ (b, \mathit{Event}\ c) \rightarrow (c \rightarrow \mathit{SF}\ a\ b) \\
\rightarrow \mathit{SF}\ a\ b
\end{aligned}
$$

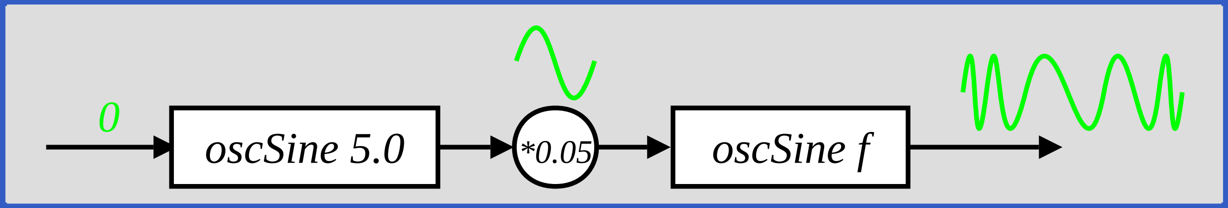
Example 1: Sine oscillator



$$
\begin{aligned}
&oscSine :: Frequency \rightarrow SF\ CV\ Sample\\
&oscSine\ f0 = \mathbf{proc}\ cv \rightarrow \mathbf{do}\\
&\quad \mathbf{let}\ f = f0 * (2 ** cv)\\
&\quad phi \leftarrow integral \prec 2 * pi * f\\
&\quad returnA \prec sin\ phi
\end{aligned}
$$

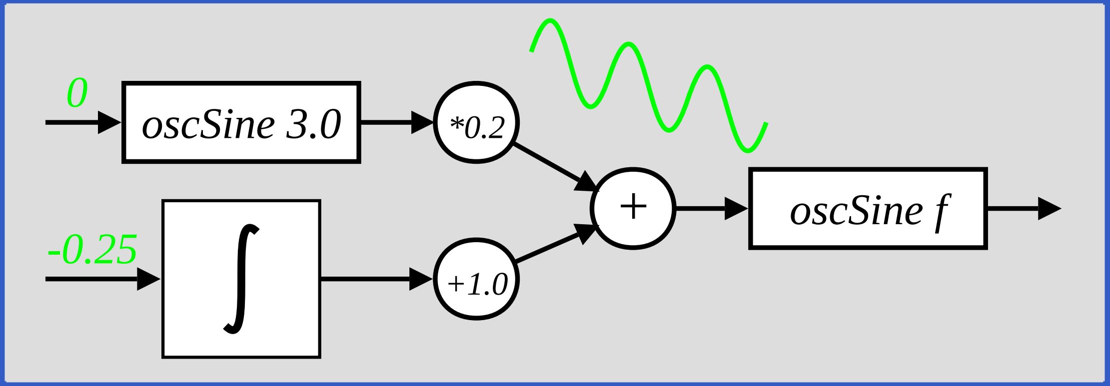
$$constant\ 0 \ggg oscSine\ 440$$

Example 2: Vibrato



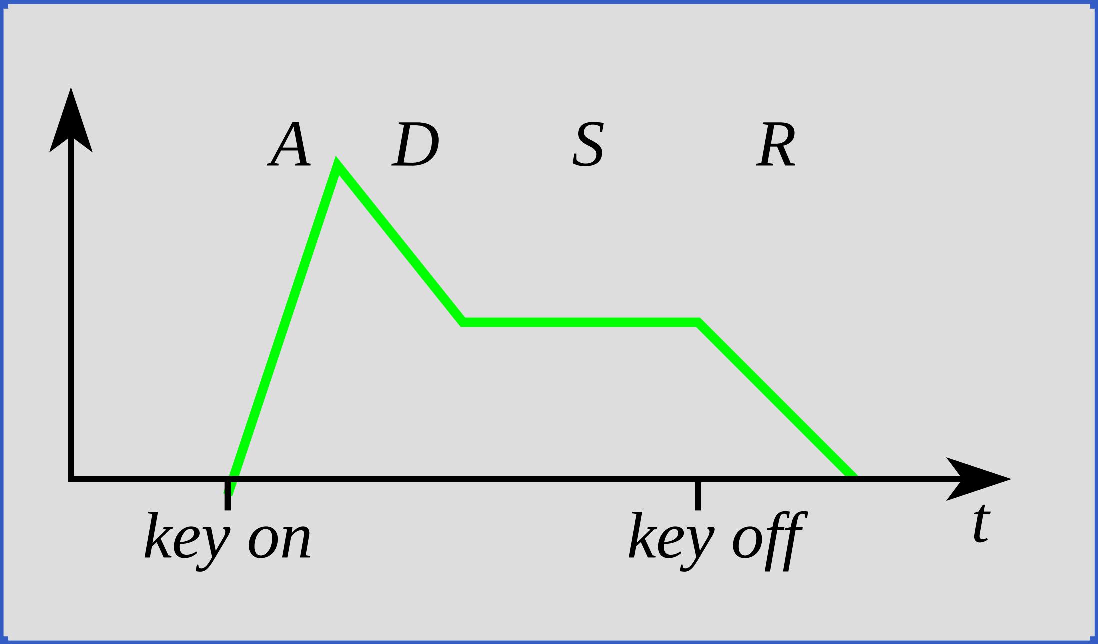
$$constant\ 0$$
$$\ggg\ oscSine\ 5.0$$
$$\ggg\ arr\ (*0.05)$$
$$\ggg\ oscSine\ 440$$

Example 3: 50's Sci Fi



$$
\begin{array}{l}
sciFi :: SF\ ()\ Sample \\
sciFi = \mathbf{proc}\ () \rightarrow \mathbf{do} \\
\quad und \leftarrow arr\ (*0.2) \lll oscSine\ 3.0 \prec 0 \\
\quad swp \leftarrow arr\ (+1.0) \lll integral \prec -0.25 \\
\quad audio \leftarrow oscSine\ 440 \prec und + swp \\
\quad returnA \prec audio
\end{array}
$$

Envelope Generators (1)



$$envGen :: CV \rightarrow [(Time, CV)] \rightarrow (Maybe\ Int)$$
$$\rightarrow SF\ (Event\ ())\ (CV, Event\ ())$$

$$envEx = envGen\ 0\ [(0.5, 1), (0.5, 0.5), (1.0, 0.5), (0.7, 0)]$$
$$(Just\ 3)$$

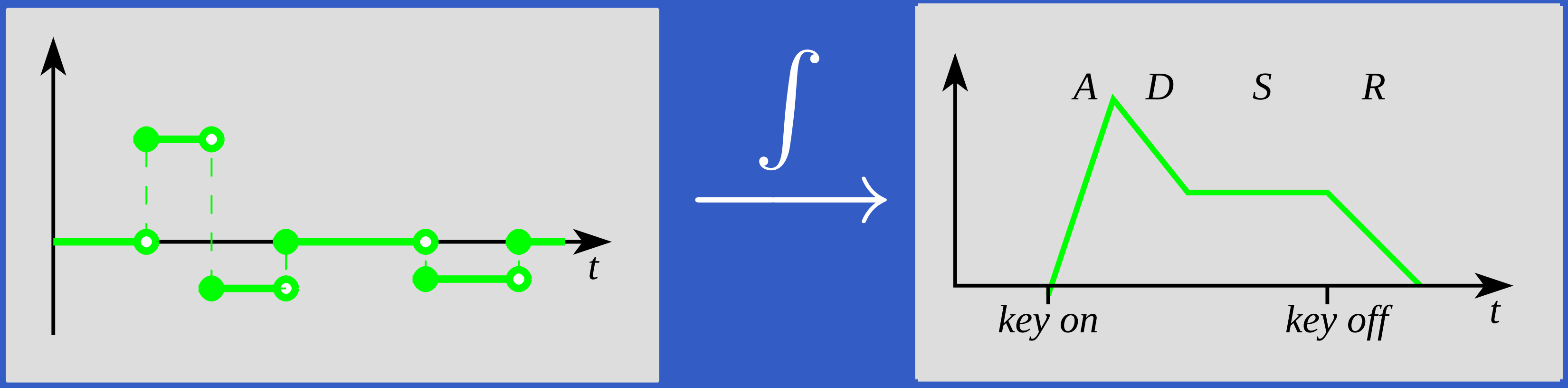
Envelope Generators (2)

How to implement?

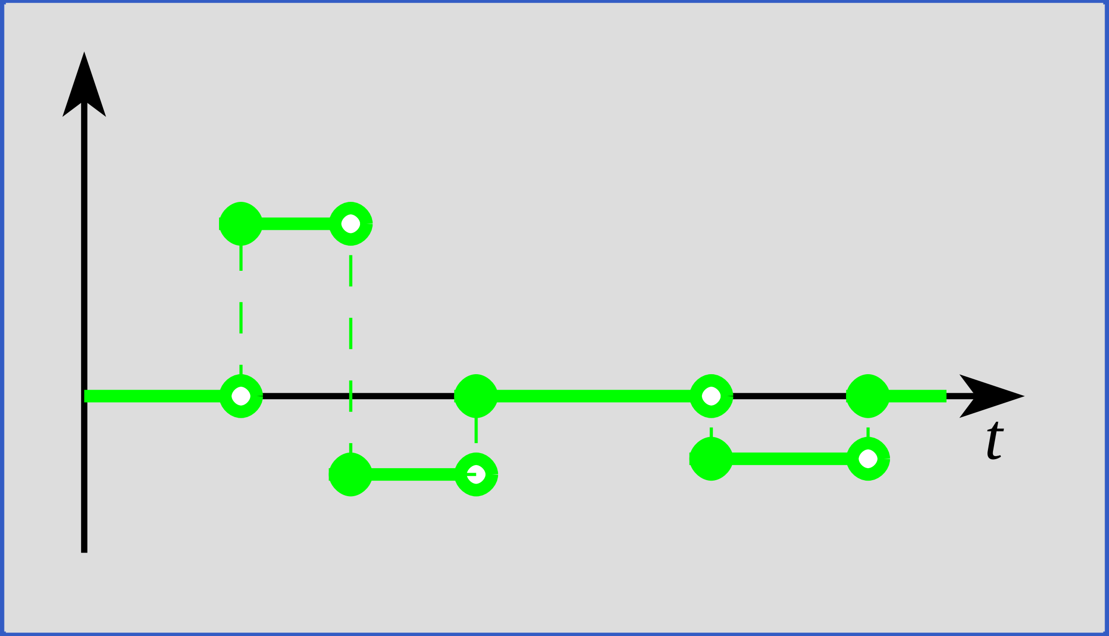
Envelope Generators (2)

How to implement?

Integration of a step function yields suitable shapes:



Envelope Generators (3)



$$afterEach :: [(Time, b)] \rightarrow SF\ a\ (Event\ b)$$

$$hold :: a \rightarrow SF\ (Event\ a)\ a$$

$$steps = afterEach\ [(0.7, 2), (0.5, -1), (0.5, 0), (1, -0.7), (0.7, 0)]$$
$$\ggg hold\ 0$$

Envelope Generators (4)

Envelope generator with predetermined shape:

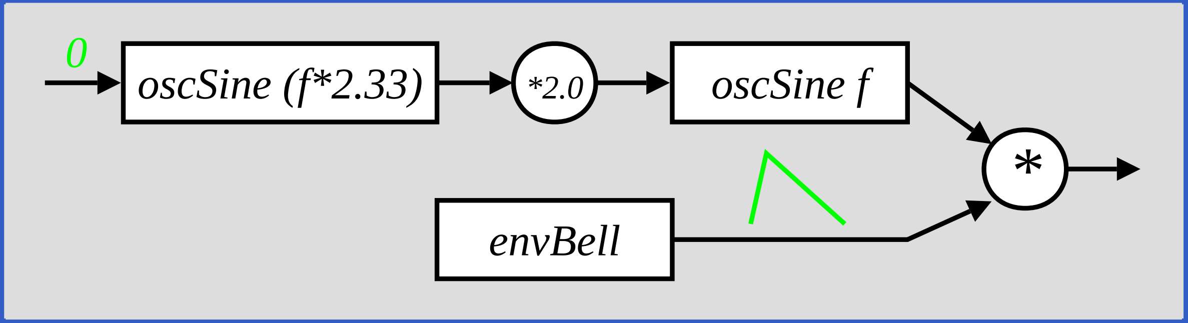
$$
\begin{aligned}
&\mathit{envGenAux} :: \mathit{CV} \rightarrow [(\mathit{Time}, \mathit{CV})] \rightarrow \mathit{SF}\ a\ \mathit{CV} \\
&\mathit{envGenAux}\ \mathit{l0}\ \mathit{tls} = \mathit{afterEach}\ \mathit{trs} \ggg \mathit{hold}\ \mathit{r0} \\
&\qquad\qquad\qquad\qquad \ggg \mathit{integral} \ggg \mathit{arr}\ (+\mathit{l0}) \\
&\quad \textbf{where} \\
&\qquad (\mathit{r0}, \mathit{trs}) = \mathit{toRates}\ \mathit{l0}\ \mathit{tls}
\end{aligned}
$$

Envelope Generators (5)

Envelope generator responding to key off:

$$
\begin{aligned}
&envGen :: CV \rightarrow [(Time, CV)] \rightarrow (Maybe\ Int) \\
&\qquad\qquad \rightarrow SF\ (Event\ ())\ (CV, Event\ ()) \\
&envGen\ l0\ tls\ (Just\ n) = \\
&\quad switch\ (\mathbf{proc}\ noteoff \rightarrow \mathbf{do} \\
&\qquad\quad l \leftarrow envGenAux\ l0\ tls1 \prec () \\
&\qquad\quad returnA \prec ((l, noEvent), noteoff\ `tag`\ l)) \\
&\qquad (\lambda l \rightarrow envGenAux\ l\ tls2 \\
&\qquad\quad \&\&\&\ after\ (sum\ (map\ fst\ tls2))\ ()) \\
&\quad \mathbf{where} \\
&\qquad (tls1, tls2) = splitAt\ n\ tls
\end{aligned}
$$

Example 4: Bell



$$
\begin{aligned}
&bell :: Frequency \rightarrow SF\ ()\ (Sample, Event) \\
&bell\ f = \mathbf{proc}\ () \rightarrow \mathbf{do} \\
&\quad m \leftarrow oscSine\ (2.33 * f) \prec 0 \\
&\quad audio \leftarrow oscSine\ f \prec 2.0 * m \\
&\quad (ampl, end) \leftarrow envBell \prec noEvent \\
&\quad returnA \prec (audio * ampl, end)
\end{aligned}
$$

Example 5: Tinkling Bell

$$
\begin{aligned}
&tinkle :: SF\ ()\ Sample \\
&tinkle = (repeatedly\ 0.25\ 84 \\
&\qquad\qquad \ggg constant\ () \\
&\qquad\qquad\quad \&\&\&\ arr\ (fmap\ (bell \circ midiNoteToFreq)) \\
&\qquad\qquad \ggg rSwitch\ (constant\ 0))
\end{aligned}
$$

Example 6: Playing a C-major scale

$$
\begin{aligned}
&scale :: SF\ ()\ Sample \\
&scale = (afterEach\ [(0.0, 60), (2.0, 62), (2.0, 64), \\
&\qquad\qquad\qquad (2.0, 65), (2.0, 67), (2.0, 69), \\
&\qquad\qquad\qquad (2.0, 71), (2.0, 72)] \\
&\qquad\quad \ggg constant\ () \\
&\qquad\qquad \&\&\&\ arr\ (fmap\ (bell \circ midiNoteToFreq)) \\
&\qquad\quad \ggg rSwitch\ (constant\ 0)) \\
&\qquad \&\&\&\ after\ 16\ ()
\end{aligned}
$$

Example 7: Playing simultaneous notes

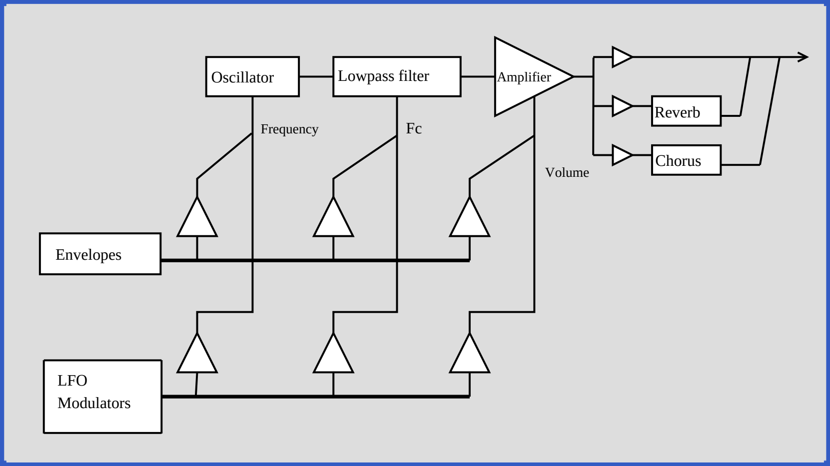
$$\begin{aligned}
&\mathit{mysterySong} :: \mathit{SF}\ ()\ (\mathit{Sample}, \mathit{Event}\ ())\\
&\mathit{mysterySong} = \mathbf{proc}\ _ \rightarrow \mathbf{do}\\
&\quad t \ \leftarrow \mathit{tinkle} \quad \prec ()\\
&\quad m \leftarrow \mathit{mystery} \prec ()\\
&\quad \mathit{returnA} \prec (0.4 * t + 0.6 * m)
\end{aligned}$$

A polyphonic synthesizer (1)

Sample-playing monophnic synthesizer:

- Read samples (instrument recordings) from SoundFont file into internal table.
- Oscillator similar to sine oscillator, except sine func. replaced by table lookup and interpolation.

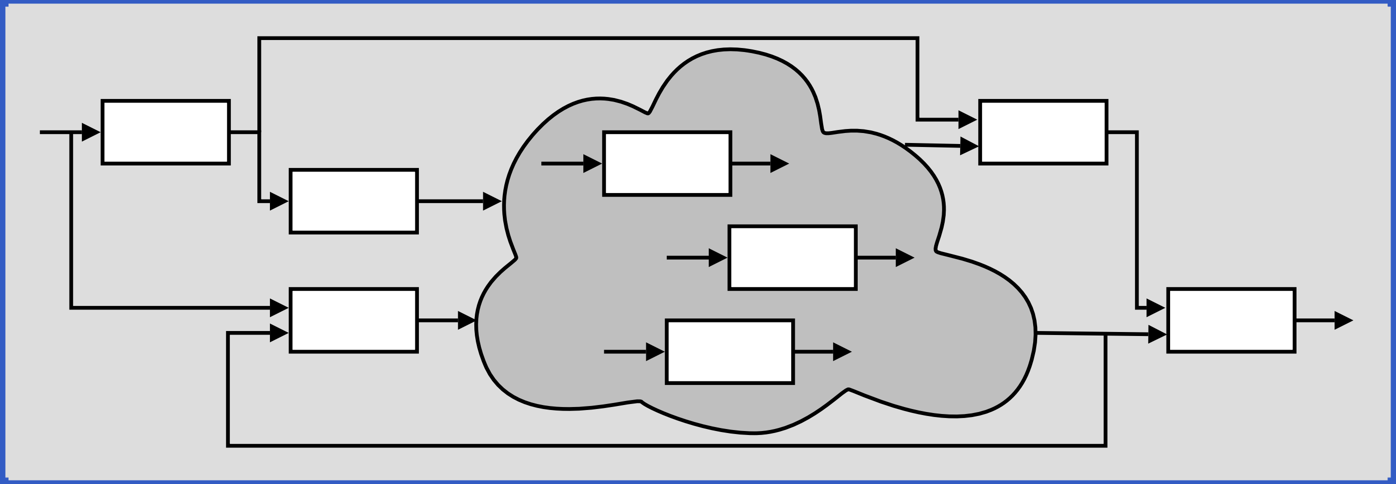
SoundFont synthesizer structure:



A polyphonic synthesizer (2)

Exploit Yampa's switching capabilities to:

- create and switch in a mono synth instance is response to each note on event;
- switch out the instance in response to a corresponding note off event.



Switched-on Yampa?

Switched-on Yampa?

Software and paper: `www.cs.nott.ac.uk/~ggg`