Csound-expression guide

Welcome to the simplest textual synthesizer.

```
> dac $ osc 440
```

Csound-expression is a Haskell framework for computer music. With the help of the library we can create our instruments on the fly. A couple of lines in the interpreter is enough to get the cool sound going out of your speakers. It can be used for simple daily sound-file processing or for a full-blown live performances. It's available on Hackage.

Let's look at how we can create computer music with Haskell.

Appendix:

WARNING: the library works best within ghci. The real-time sound rendering function `dac` spawns a child process in the background which may continue to execute after you stop the main process that runs the programm. It's not so in vim but it happens in the Sublime Editor and when you invoke `runhaskell`. So the best is to write you program in the separate file and then load it in the ghci and invoke the function `main` (which runs the sound rendering with the function `dac`).

Introduction

Csound-expression is a framework for creation of computer music. It's a Haskell library to make Csound much more friendly. It generates Csound files out of Haskell code.

Csound is an audio programming language. It is really awesome. It features unlimited polyphony, hundreds of synth-units including FM, granular synth, frequency domain transforms and many more. Midi and OSC control, compatible with JACK. With JACK it's easy to use with your DAW of choice. It can run on mobile devices and even in the web browser. It has the support for GUI-widgets.

But Csound is clumsy. It's an old C-style language. We can boost it with functional programming. The Csound-expression gives you eloquence of Haskell combined with power of Csound.

With the help of the library we can create our instruments on the fly. A few lines in the interpreter is enough to get the cool sound going out of your speakers. Some of the features of the library are heavily inspired by reactive programming. We can invoke the instruments with event streams. Event streams can be combined in the manner of reactive programming. The GUI-widgets are producing the event streams as a control messages. Moreover with Haskell we get all standard types and functions like lists, maps, trees. It's a great way to organize code and data.

Let's look at how we can create computer music with Haskell. If you are a Csounder that stumbled upon this page and got interested then it's better to learn some Haskell. The basic level is enough to use the library. I recommend the book Learn you a Haskell for a Great Good by Miran Lipovaca. It's a great book with an elephant on the cover. It's a wonderful introduction to the wisdom and miracles of the Haskell.

Installation guide

Let's install everything. The first thing we need is a csound compiler. When it's installed properly we can type in the terminal:

```
> csound
```

It will print the long message. Ubuntu/Debian users can install the Csound with `apt-get`:

```
> sudo apt-get install csound csound-gui
```

The next thing is a working Haskell environment with `ghc` and `cabal-install` It can be installed with Haskell Platform. If it works to install the `csound-expression` we can type in the terminal:

```
> cabal update
> cabal install csound-expression
```

Let's have a break and take a cup of tea. The library contains a lot of modules to install.

The first sound

Let's start the `ghci` and load the main module `Csound.Base`. It exports all modules:

```
> ghci
Prelude> :m +Csound.Base
Prelude Csound.Base>
```

We can play a sine wave with 440 Hz:

```
> dac $ osc 440
```

Pressing Ctrl+C stops the sound. The expression `osc 440` makes the sine wave and the function `dac` makes a file `tmp.csd` in the current directory invokes the `csound` on it and sends the output to speakers.

WARNING: the library works best within `ghci`. The real-time sound rendering function `dac` spawns a child process in the background which may continue to execute after you stop the main process that runs the program. It's not so in vim but it happens in the Sublime Editor and when you invoke `runhaskell`. So the best is to write you program in the separate file and then load it in the `ghci` and invoke the function `main` (which runs the sound rendering with the function `dac` or another sound rendering function).

Key principles

Here is an overview of the features:

- Keep it simple and compact.
- Try to hide low level Csound's wiring as much as we can (no ids for ftables, instruments, global variables).
- Don't describe the whole Csound in all it's generality but give the user some handy tools to play with sound.
- No distinction between audio and control rates on the type level. Derive all rates from the context. If the user plugs signal to an opcode that expects an audio rate signal the argument is converted to the right rate.
- Watch out for side-effects. There is a special type called `SE`. It functions as `IO` in Haskell.
- Less typing, more music. Use short names for all types. Make library so that all expressions can be built without type annotations. Make it simple for the compiler to derive all types. Don't use complex type classes.
- Make low level opcode definitions simple. Let user define his own opcodes (if they are missing).

- Ensure that output signal is limited by amplitude. Csound can produce signals with HUGE amplitudes. Little typo can damage your ears and your speakers. In generated code all signals are clipped by 0dbfs value. 0dbfs is set to 1. Just as in Pure Data. So 1 is absolute maximum value for amplitude.
- No dependency on Score-generation libraries. Score (or list of events) is represented with type class. You can use your favorite Score-generation library if you provide an instance for the CsdSco type class. Currently there is support for temporal-music-notation library (see temporal-csound package).
- Remove score/instrument barrier. Let instrument play a score within a note and trigger other instruments.
- Set Csound flags with meaningful (well-typed) values. Derive as much as you can from the context.
- Composable GUIs. Interactive instruments should be easy to make.

Acknowledgements

I'd like to mention those who supported me a lot with their music and ideas:

- **music**: entertainment for the braindead, three pandas and the moon, odno no, ann's'annat & alizbar, toe, iamthemorning, atoms for piece / radiohead, loscil, boards of canada, Hozan Yamamoto, Tony Scott and Shinichi Yuize.
- **ideas**: Conal Elliott, Oleg Kiselyov, Paul Hudak, Gabriel Gonzalez, Rich Hickey and Csound's community.
- Thanks a lot to all who patiently answered my questions and provided skillful solutions, encouragement and ideas:

  Iain McCurdy, Victor Lazarini, Rory Walsh, Steven Yi, John Ffitch, Joachim Heintz, Peter Burgess, Dr. Richard Boulanger, Michael Gogins, Oeyvind Brandtsegg, Richard Dobson, Partev Barr Sarkissian, Dave Phillips, Guillermo Senna, Art Hunkins, Ben McAllister, Michael Rhoades, Brian Merchant, Gleb Rogozinsky, Eugene Cherny, Wolf Peuker,HlÃ¶Ã°ver SigurÃ°sson, Aaron Krister Johnson, Andy Fillebrown and friends)

  tell me if I forgot to mention you :)

- <= Home
- => Basic types
- Home

Basic types

Let's look at the basic types of the library.

Signals (Sig)

We are going to make an audio signal. So the most frequently used type is a signal. It's called `Sig`. The signal is a stream of numbers that is updated at a certain rate. Actually it's a stream of small arrays of doubles. For every cycle the audio-engine updates it. It can see only one frame at the given time.

Conceptually we can think that signal is a list of numbers. A signal is an instance of type class `Num`, `Fractional` and `Floating`. So we can treat signals like numbers. We can create them with numeric constants, add them, multiply, subtract, divide, process with trigonometric functions.

We assume that we are in ghci session and the module `Csound.Base` is loaded.

```
$ ghci
> :m +Csound.Base
```

So let's create a couple of signals:

```
> let x = 1 :: Sig
> let y = 2 :: Sig
> let z = (x + y) * 0.5
```

Constants are pretty good but not that interesting as sounds. The sound is time varying signal. It should vary between -1 and 1. It's assumed that 1 is a maximum volume. Everything beyond the 1 is clipped.

Let's study the simple waveforms:

```
osc, saw, tri, sqr :: Sig -> Sig
```

They produce sine, sawtooth, triangle and square waves. The output is band limited (no aliasing beyond Nyquist). The waveform function takes in a frequency (and it's also a signal) and produces a signal that contains wave of certain shape that is repeated with given frequency (in Hz).

Let's hear a sound of the triangle wave at the rated of 220 Hz:

```
> dac $ tri 220
```

We can press `Ctrl+C` to stop the sound from playing. If we know the time in advance we can set it with the function `setDur`:

```
> dac $ setDur 2 $ tri 220
```

Right now the sound plays only for 2 seconds. The `setDur` function should be used only once. Right before the sending output to the `dac`.

We can vary the frequency with slowly moving oscillator:

```
> dac $ tri (220 + 100 * osc 0.5)
```

If we use the `saw` in place of `tri` we can get a more harsh siren-like sound.

We can adjust the volume of the sound by multiplying it:

```
> dac $ mul 0.5 $ saw (220 + 100 * osc 0.5)
```

Here we used the special function `mul`. We could just use the normal Haskell's `*`. But `mul` is more convenient. It can work not only for signals but for tuples of signals (if we want a stereo playback) or signals that contain side effects (wrapped in the monad `SE`). So the `mul` is preferable.

Constant numbers (D)

Let's study two another useful functions:

```
leg, xeg :: D -> D -> D -> D -> Sig
```

They are Linear and eXponential Envelope Generators. They create ADSR-envelopes.

They take in a four arguments. They are:

- **attack time**: time for signal to reach the 1 (in seconds)
- **decay time**: time for signal to reach the sustain level (in seconds)
- **sustain level**: the value for sustain level (between 0 and 1)
- **release time**: how many seconds it takes to reach the zero after release.

We can notice the new type `D` in the signature. It's for constant doubles. We can think that it's a normal value of type `Double`. It's a `Double` that is embedded in the Csound. From the point of implementation we don't calculate these doubles but use them to generate the Csound code.

Let's create a signal that is gradually changes it's pitch:

```
> dac $ saw (50 + 150 * leg 2 2 0.5 1)
```

Notice that signal doesn't reaches the release phase. It's not a mistake! The release happens when we release a key on the midi keyboard. We don't use any midi here so the release never happens.

But we can try the virtual midi device:

```
> vdac $ midi $ onMsg $ \x -> saw (x + 150 * leg 2 2 0.5 1)
```

Right now don't bother about the functions `midi` and `onMsg`. We are going to take a closer look at then in the chapter *User interaction*. That's how we plug in the midi-devices.

The value of type `D` is just like a Haskell's `Double`. We can do all the Double's operations on it. It's useful to know how to convert doubles to `D`'s and how to convert `D`'s to signals:

```
double :: Double -> D
sig    :: D -> Sig
```

There are more generic functions:

```
linseg, expseg :: [D] -> Sig
```

They can construct the piecewise linear or exponential functions. The arguments are:

```
linseg [a, timeAB, b, timeBC, c, timeCD, d, ...]
```

They are alternating values and time stamps to progress continuously from one value to another. Values for `expseg` should be positive (above 0 and not 0).

There are two more generic functions for midi notes:

```
linsegr, expsegr :: [D] -> D -> D -> Sig
```

The two last arguments are the release time and the final value for release stage. They are usefull for midi-instruments.

Another frequently used functions are

```
fadeIn  :: D -> Sig
fadeOut :: D -> Sig

fades   :: D -> D -> Sig
fades fadeInTime fadeOutTime = ...
```

They produce more simple envelopes. The `fadeIn` rises in the given amount of seconds form 0 to 1. The `fadeOut` does the opposite. It's 1 from the start and then it fades out to zero in given amount of seconds but only after release. The `fades` combines both functions.

Strings (Str)

The friend of mine has made a wonderful track in Ableton. I have a wav-file from her and want to beep-along with it. I can use a `diskin2` opcode for it:

```
diskin2 :: Tuple a => Str -> Sig -> a
diskin2 fileName playBackSpeed = ...
```

It takes in a name of the file and playback speed and produces a tuple of signals. We should specify how many outputs are in the record by specifying precise the type of the tuple. There are handy helpers for this:

```
ar1 :: Sig -> Sig
ar2 :: (Sig, Sig) -> (Sig, Sig)
ar3 :: (Sig, Sig, Sig) -> (Sig, Sig, Sig)
ar4 :: (Sig, Sig, Sig, Sig) -> (Sig, Sig, Sig, Sig)
```

Every function is an identity. It's here only to help the type inference. Find a *.wav file (your mileage may vary)

```
$ uname -a
 Linux ... aptosid 4.1-1 (2015-06-22) x86_64 GNU/Linux
$ sudo updatedb
$ locate .wav
...
/usr/share/sounds/alsa/Noise.wav
...
$ file /usr/share/sounds/alsa/Noise.wav
 /usr/share/sounds/alsa/Noise.wav: RIFF (little-endian) data, WAVE audio, Microsoft PCM, 16 bit, mono 48000 Hz
```

With a mono wav-file you can use:

```
> let sample = ar1 $ diskin2 (text "Noise.wav") 1
```

Find a stereo wav-file as above or however:

```
$ cp /usr/share/webkitgtk-1.0/resources/audio/Composite.wav .
$ file Composite.wav
 Composite.wav: RIFF (little-endian) data, WAVE audio, Microsoft PCM, 16 bit, stereo 44100 Hz
```

So we have a stereo wav-file, and we want to play it at normal speed.

```
> let sample = toMono $ ar2 $ diskin2 (text "Composite.wav") 1
```

We don't care right now about the stereo so we have converted everything to mono with function.

```
toMono :: (Sig, Sig) -> Sig
```

The first argument of the `diskin2` is not a Haskell's `String`. It's a Csound's string so it has a special name `Str`. It's just like `D`'s for `Double`'s. We used a converter function to lift the Haskell string to Csound one:

```
text :: String -> Str
```

The `Str` has instance of `IsString` so if we are using the extension `OverloadedStrings` we don't need to call the function `text`.

Ok, we are ready to play along with it:

```
> let sample = toMono $ ar2 $ diskin2 (text "Composite.wav") 1
> let meOnKeys = midi $ onMsg osc
> vdac $ mul 0.5 $ meOnKeys + return sample
```

Notice how simple is the combining midi-devices output with the regular signals. The function `midi` produces a normal signal wrapped in `SE`-monad. We can use it anywhere.

There are useful shortcuts that let us use a normal Haskell strings:

```
readSnd :: String -> (Sig, Sig)
loopSnd :: String -> (Sig, Sig)
loopSndBy :: D -> String -> (Sig, Sig)
readWav :: Sig -> String -> (Sig, Sig)
loopWav :: Sig -> String -> (Sig, Sig)
```

The functions with `read` play the sound files only once. The functions with `loop` repeat over the sample over and over. With `loopSndBy` we can specify the time length of the loop period. The `readWav` and `loopWav` can read the file with given speed. The 1 is a normal speed. The -1 is playing in reverse. Negative speed works only for `loopWav`.

So we can read our friends record like this:

```
let sample = loopSnd "Composite.wav"
```

If we want only a portion of the sound to be played we can use the function:

```
takeSnd :: D -> Sig -> Sig
```

It takes only given amount of seconds from the input signal and fills the rest with silence. It's interesting that we can loop not only with samples but with regular signals too:

```
repeatSnd :: D -> Sig -> Sig
```

It loops the signal over given amount of time (in seconds). We can try it out:

```
> dac $ repeatSnd 3 $ leg 1 2 0 0 * osc 220
```

Tables (Tab)

We have studied the four main waveform functions: `osc`, `tri`, `saw`, `sqr`. But what if we want to create our own waveform. How can we do it?

What if we want not a pure sine but two more partials. We want a sum of sine partials and a first harmonic with the amplitude of 1 the second is with 0.5 and the third is with 0.125.

We can do it with `osc`:

```
> let wave x = mul (1/3) $ osc x + 0.5 * osc (2 * x) + 0.125 * osc (3 * x)
> vdac $ midi $ onMsg $ mul (fades 0.1 0.5) . wave
```

But there is a better way for doing it. Actually the oscillator reads a table with a fixed waveform. It reads it with a given frequency and we can hear it as a pitch. Right now our function contains three `osc`. Each of them reads the same table. But the speed of reading is different. It would be much better if we could write the static waveform with three harmonics in it and read it with one oscillator. It would be much more efficient. Think about waveforms with more partials.

We can achieve this with function:

```
oscBy :: Tab -> Sig -> Sig
```

It creates an oscillator with a custom waveform. The static waveform is encoded with value of type `Tab`. The `Tab` is for one dimensional table of doubles. In the Csound they are called functional tables. They can be created with GEN-routines. We don't need to create the tables directly. Like filling each cell with a value (going through the table in the loop). There are plenty of functions that can create specific tables.

Right now we want to create a sum of partials or harmonic series. We can use the function sines:

```
sines :: [Double] -> Tab
```

Let's rewrite the example:

```
> let wave x = oscBy (sines [1, 0.5, 0.125]) x
> vdac $ midi $ onMsg $ mul (fades 0.1 0.5) . wave
```

You can appreciate the simplicity of these expressions if you try to make it directly in the Csound. But you don't need to! There are better ways and here is one of them.

What if we want not 1, 2, and third partials but 1, 3, 7 and 11? We can use the function:

```
sines2 :: [(PartialNumber, PartialStrength)] -> Tab
```

It works like this:

```
> let wave x = oscBy (sines2 [(1, 1), (3, 0.5), (7, 0.125), (11, 0.1)]) x
```

The table size

What is the size of the table? We can create the table of the given size. By default it's 8196. The more size the better is precision. For efficiency reason the tables size in most cases should be equal to some degree of 2. We can set the table size with one of the functions:

```
lllofi, llofi, lofi, midfi, hifi, hhifi, hhhifi
```

The `lllofi` is the lowest fidelity and the `hhhfi` is the highest fidelity.

We can set the size explicitly with:

```
setSize :: Int -> Tab -> Tab
```

The guard point

If you are not familiar with Csound's conventions you are probably not aware of the fact that for efficiency reasons Csound requires that table size is equal to power of 2 or power of two plus one which stands for guard point (you do need guard point if your intention is to read the table once but you don't need the guard point if you read the table in many cycles, then the guard point is the the first point of your table).

If we read the table once we have to set the guard point with function:

```
guardPoint :: Tab -> Tab
```

There is a short-cut called just `gp`. We should use it with `exps` or `lins`.

Specific tables

There are a lot of GEN-routines available. Let's briefly discuss the most usefull ones.

We can write the specific numbers in the table if we want:

```
doubles :: [Double] -> Tab
```

Linear and exponential segments:

```
consts, lins, exps, cubes, splines :: [Double] -> Tab
```

Reads samples from files (the second argument is duration of an audio segment in seconds)

```
data WavChn = WavLeft | WavRight | WavAll
data Mp3Chn = Mp3Mono | Mp3Stereo | Mp3Left | Mp3Right | Mp3All

wavs :: String -> Double -> WavChn -> Tab
mp3s :: String -> Double -> Mp3Chn
```

Harmonic series:

```
type PartialStrength = DoubleSource
type PartialNumber = DoubleSource
type PartialPhase = DoubleSource
type PartialDC = Double

sines  :: [PartialStrength] -> Tab
sines2 :: [(PartialNumber, PartialStrength)] -> Tab
sines3 :: [(PartialNumber, PartialStrength, PartialPhase)] -> Tab
sines4 :: [(PartialNumber, PartialStrength, PartialPhase, PartialDC)] -> Tab
```

Special cases for harmonic series:

```
sine, cosine, sigmoid :: Tab
```

Side effects (SE)

The `SE`-type is for functions that work with side effects. They can produce effectful value or can be used just for the side effect.

For example every function that generates random numbers uses the type `SE`.

To get the white or pink noise we can use:

```
white :: SE Sig
pink  :: SE Sig
```

Let's listen to the white noise:

```
> dac $ mul 0.5 $ white
```

We can get the random numbers with linear interpolation. The output values lie in the range of -1 to 1:

```
rndi :: Sig -> SE Sig
```

THe first arhument is frequency of generated random numbers. We can get the constant random numbers (it's like sample and hold function with random numbers):

```
rndh :: Sig -> SE Sig
```

We can use the random number generators as LFO. The `SE` is a `Functor`, `Applicative` and `Monad`. We rely on these properties to get the output:

```
> let instr lfo = 0.5 * saw (440 + lfo)
> dac $ fmap instr (20 * rndi 5)
```

There are unipolar variants: `urndh` and `urndi`. The output ranges form 0 to 1 for them.

Note that the function `dac` can work not only signals but also on the signals that are wrapped in the type `SE`.

Let's take a break and listen to the filtered pink noise:

```
> dac $ mul 0.5 $ fmap (mlp (on 50 2500 $ tri 0.2) 0.3) $ pink
```

The function `on` is usefull for mapping the range (-1, 1) to a different interval. In the expression `on 50 2500 $ tri 0.2` oscillation happens in the range `(50, 2500)`. There is another usefull function `uon`. It's like `on` but it maps from the range `(0, 1)`.

The essence of the `SE Sig` type lies in the usage of random values. In the pure code we can not distinguish between these two expressions:

```
x1 = let a = rndh 1 in a + a
x2 = rndh 1 + rndh 1
```

For `x1` we want only one random value but for `x2` we want two random values.

The value is just a tiny piece of code (we don't evaluate expressions but use them to generate csound code). The renderer performs common subexpression elimination. So the examples above would be rendered in the same code.

We need to tell to the renderer when we want two random values. Here comes the `SE` monad (Side Effects for short).

```
x1 = do
  a <- rndh
  return $ a + a

x2 = do
  a1 <- rndh
  a2 <- rndh
  return $ a1 + a2
```

The SE was introduced to express the randomness. But then it was usefull to expres many other things. Procedures for instance. They don't produce signals but do smth usefull:

```
procedure :: SE ()
```

The `SE` is used for allocation of delay buffers in the functions.

```
deltap3 :: Sig -> SE Sig
delayr :: D -> SE Sig
delayw :: Sig -> SE ()
```

The `deltap3` is used to allocate the delay line. After allocation we can read and write to delay lines with `delayr` and `delayw`.

The `SE` is used for allocation of local or global variables (see the type `SERef` in the module `Csound.Control.SE`).

For convinience the `SE Sig` and `SE` of tuples of signals is instance of `Num`. We can sum and multiply the signals wrpapped in the `SE`. That's code is ok:

```
> dac $ white + 0.5 * pink
> dac $ white + return (osc 440)
```

Mutable values

We can create mutable variables. It works just like the normal Haskell mutable variables. We can create a reference and the we should use the functions on the reference to read and write values.

There are two types of the variables: local and global variables. The local variables are visible only within one Csound instrument. The global variables are visible everywhere.

We can create a reference to the mutable variable with functions:

```
newRef          :: Tuple a => a -> SE (Ref a)
newGlobalRef    :: Tuple a => a -> SE (Ref a)
```

They take in an initial value and create a value of the type `Ref`:

```
data Ref a = Ref
    { writeRef :: a -> SE ()
    , readRef  :: SE a
    }
```

We can write and read values from reference.

Tuples (Tuple)

Some of the Csound functions are producing several outputs. Then the output is represented with `Tuple`. It's a special type class that contains all tuples of Csound values.

There is a special case. The type `Unit`. It's Csound's alias for Haskell's `()`-type. It's here for implementation reasons.

We have already encountered the tuples when we have studied the function `diskin2`.

```
diskin2 :: Tuple a => Str -> Sig -> a
```

In Csound the functions can produce varied amount of arguments. The number of arguments is specified right in the code. But Haskell is different. The function can produce only certain number of arguments. To relax this rule we can use the special type class `Tuples`. Now we can return different number of arguments. But we have to specify them with type signature. There are helpers to make it easier:

```
ar1 :: Sig -> Sig
ar2 :: (Sig, Sig) -> (Sig, Sig)
ar3 :: (Sig, Sig, Sig) -> (Sig, Sig, Sig)
ar4 :: (Sig, Sig, Sig, Sig) -> (Sig, Sig, Sig, Sig)
```

The Signal space (SigSpace)

We often want to transform the signal which is wrapped in the other type. It can be a monophonic signal. If it's just a pure `Sig` then it's not that difficult. We can apply a function and get the output. But what if the signal is stereo or what if it's wrapped in the `SE`. But it has a signal(s) that we want to process. We can use different combinations of the function `fmap`. But there is a better way.

We can use the special type class `SigSpace`:

```
class Num a => SigSpace a where
    mapSig :: (Sig -> Sig) -> a -> a
    bindSig :: (Sig -> SE Sig) -> a -> SE a
```

There are lots of instances. For signals, tuples of signals, tuples of signals wrapped in the `SE`, the signals that come from UI-widgets such as knobs and sliders.

If you are too lazy to write `mapSig` there is a shortcut `at` for you. It's the same as `mapSig`. That's how we can filter a noise. The `linseg` creates a stright line between the points `1500` and `250` that lasts for `5` seconds:

```
> dac $ at (mlp (linseg [1500, 5, 250]) 0.1) $ white
```

It let us apply signal transformation functions to values of many different types. The one function that we have already seen is `mul`:

```
mul :: SigSpace a => Sig -> a -> a
```

It can scale the signal or many signals.

There is another cool function. It's `cfd`:

```
cfd :: SigSpace a => Sig -> a -> a -> a
```

It's a crossfade between two signals. The first signal varies in range 0 to 1. It interpolates between second and third arguments.

Also we can use bilinear interpolation with four signals

```
cfd4 :: SigSpace a => Sig -> Sig -> a -> a -> a -> a -> a
cfd4 x y asig1 asig2 asig3 asig4
```

We can imagine that we place four signals on the corners of the unipolar square. we can move within the square with `x` and `y` signals. The closer we get to the corner the more prominent becomes the signal that sits in the corner and other three become quiter. The corner to signal map is:

- `(0, 0)` is for `asig1`
- `(1, 0)` is for `asig2`
- `(1, 1)` is for `asig3`
- `(0, 1)` is for `asig4`

The `cfds` can operate on many signals. The first list length equals the second one minus one.

```
cfds :: SigSpace a => [Sig] -> [a] -> a
```

Another usefull function is weighted sum

```
wsum :: SigSpace a => [(Sig, a)] -> a
```

It's a weighted sum of signals. Can be useful for mixing sounds from several sources.

The signal outputs (Sigs)

It's a tuple of signals. It's for mono, stereo and other sound-outputs.

```
class Tuple a => Sigs a
```

Spectrums (Spec)

We can extract a spectrum from the signal. It's an advanced type. The simplest function to create a spectrum is:

```
toSpec   :: Sig -> Spec
fromSpec :: Spec -> Sig
mapSpec  :: (Spec -> Spec) -> Sig -> Sig
```

With `Spec` we can apply spectral transformations to signal. we can create a vocoder effect with it for instance or scale a pitch or crossfade between several timbres.

We can interpolate between several signals:

```
cfdSpec :: Sig -> Spec -> Spec -> Spec
cfdSpec4 :: Sig -> Sig -> Spec -> Spec -> Spec -> Spec -> Spec
cfdsSpec :: [Sig] -> [Spec] -> Spec
```

To scale the pitch there are handy shortcuts:

```
scaleSpec :: Sig -> Sig -> Sig
scalePitch :: Sig -> Sig -> Sig
```

`scaleSpec` scales the frequency of the signal in Hz ratios but `scalePitch` does it in semitones.

If we have a spectrum we can process it with many functions from the module Spectral processing.

- <= Introduction
- => Rendering Csound files
- Home

Rendering Csound files

We know how to play the sound live. We can use the function `dac` for it. Also we know how to use virtual midi-device. We can use `vdac` for it. But there are many other ways to render the Csound file. Let's study them. The functions that we are going to look at live in the module Csound.IO.

Producing the Csound code

The csound-expression library at its core is a Csound file generator. The most basic thing it can do is to make as `String` that contains the Csound code.

```
renderCsd :: RenderCsd a => a -> IO String
```

It takes something renderable and produces a `String`. We can write the String to the file with function:

```
writeCsd :: RenderCsd a => String -> a -> IO ()
writeCsd fileName csd = ...
```

These functions are useful if we want to use the Csound code without Haskell. For instance we can take it on some computer that doesn't have the Haskell installed on it and run it with Csound. It can be used on mobile devices inside of the other programs with Csound API. We can send it to our friend by mail. So that he can render it at home and hear the music.

Saving the output to sound-file

We can write the output to wav-file or aiff-file with function:

```
writeSnd :: RenderCsd a => String -> a -> IO ()
writeSnd fileName csd = ...
```

Let's write a 10 seconds of concert A (440 Hz). We can use it for tuning:

```
> writeSnd "A.wav" $ setDur 10 $ osc 440
```

The audio is going to be rendered off-line. The good thing about off-line rendering is that it can happen much faster. It depends on the complexity of the sound units. It's not limited with real-time constraints. So we can render a file with 30 minutes very quickly.

Playing live

We have already seen these functions. You can guess them:

```
dac  :: RenderCsd a => a -> IO ()
vdac :: RenderCsd a => a -> IO ()
```

The `dac` is for sending the sound to sound card and the `vdac` is for playing with virtual midi device.
If you have the real midi-controller you can use it with `dac` function. Just use the plain `midi`-function and everything should work out of the box.

Playing the sound with player

We can render the file to sound file and play it with sound player. Right now only Linux players are supported:

```
mplayer, totem :: RenderCsd a => a -> IO ()
```

Render-able types

It's time to take a closer look at the arguments of the functions. What does type class `RenderCsd` mean?

We have seen how we can play a mono and stereo signals with it. But can we do anything else? Yes, we can.

We can render the signals or tuples of signals.

```
Sig, (Sig, Sig), (Sig, Sig, Sig, Sig)
```

They can be wrapped in the type `SE` (they can contain side effects)

```
SE Sig, SE (Sig, Sig), SE (Sig, Sig, Sig, Sig)
```

We can listen on the sound card ports for input signals. Yes, we can use the Csound as a sound-effect. Then we render a function:

```
(Sigs a, Sigs b) => RenderCsd (a -> b)
(Sigs a, Sigs b) => RenderCsd (a -> SE b)
```

We can render a procedure:

```
SE ()
```

In this case we are using Csound to do something useful but without making any noise about it. Maybe we are going to manipulate some sound-files or receive Midi-messages and silently print them on the screen.

There is also support for GUIs. We are going to encounter it soon. The signal that is wrapped in the UI is also can be rendered:

```
Source Sig, Source (Sig, Sig), ...
```

Options

We don't care much about sound rates for the output or what sound card to use or what size does internal sound buffers have. But if we do?

Can we alter the sample rate? The default is 44100. It's good enough for real-time performance. If we want to produce the high quality audio we need to alter the defaults. That's where the `Options` are handy.

If we look at the module Csound.IO we shortly notice that there are duplicate functions that ends with `By`

```
dacBy       :: RenderCsd a => Options -> a -> IO ()
writeCsdBy  :: RenderCsd a => Options -> String -> a -> IO ()
writeSndBy  :: RenderCsd a => Options -> String -> a -> IO ()
...
```

They take in one more argument. It's `Options` (Csound.Options). With `Options` we can do a lot of fine tuning. we can alter audio sample rate, alter the default size for functional tables, assign settings for JACK-instruments and so on.

That's how we can alter the sound-rates:

```
> let opt = setRates 96000 64
> writeSndBy opt result
```

The sound rates contain two integers. The former is the result audio rate and the latter is for the length of the single audio array. The latter is called blackSize (it's ksmps in Csound). It affects the rate of control signals. We produce the audio signals in frames of the `blockSize` length. When we are done with one frame we can listen for the control signals and then apply them in the production of the next frame.

The cool thing about `Options` is that it's a `Monoid`. We can use the default `Options` and alter only the things we need to alter without the need to redefine the other things. Let's see how we can combine different settings:

```
> let opt = setRates 96000 64 <> def { tabFi = coarseFi 15 }
> writeSndBy opt result
```

We combine the two options with Monoid's `mappend` function. The first option is for rate and the second set's higher degree of fidelity for functional tables. It affects the default table size. By default it's 13th degree of 2. But we have set it to 15.

- <= Basic types
- => Basics of sound synthesis
- Home

Basics of sound synthesis

Let's explore the sound synthesis with Haskell. We are going to study the subtractive synthesis. In subtractive synthesis we start with a complex waveform and then filter it and apply cool effects. That's how we make interesting instruments with subtractive synthesis. Let's look at the basic structure of synthesizer.

Basic structure of synthesizer

Let's imagine that we have a piano midi controller. When we press the key we get the pitch (what note do we press) and volume (how hard do we press it). So the input is the volume and pitch. An instrument converts it to the sound wave. The sound wave should naturally respond to the input parameters. When we hit harder it should be louder and when we press the lower notes it should be lower in pitch and possibly richer in timbre.

To get the sound wave we should ask ourselves: what is an instrument (or timbre)? How it is constructed? What parts should it contain?

The subtractive synthesis answers to these questions with the following scheme:

```
            +----------+     +--------+      +-------+     +--------+
Pitch, ---> | Wave Gen |---->| Filter |----->| Gain  |---->| Effect |---> Sound
Velocity    +----------+     +--------+      +-------+     +--------+

        +-----+   +----+
        | LFO |   | EG |
        +-----+   +----+
```

Synth has six main units:

- **Wave generator (VCO):** It defines the basic spectrum of the sound. It's often defines the pitch of the sound. We create the static sound waveform with the given pitch. Sometimes we produce the noise with it (for percussive sounds).
- **Filter (VCF):** Filter controls the brightness of the timbre. With filter we can vary the timbre in time or make it dynamic.
- **Amplifier or gain (VCA):** With gain we can adjust the volume of the sound (scale the amplitude).
- **Processor of effects (FX):** With effects we can make the sound cool and shiny. It can be delay, reverb, flanger, chorus, vocoder, distortion, name your favorite effect.

Units to make our sounds alive (we can substitute the dumb static numbers with time varied signals that are generated with LFO's or EG's):

- **Low frequency generator (LFO):** A low frequency oscillator generates waves at low frequency (0 to 50 Hz). It's used to change the parameters of the other units in time. LFO's are used to change parameters periodically.

- **Envelope generator (EG):** An envelope creates a piecewise function (linear or exponential). It slowly varies in time. We can describe the steady changes with EG's. It's often used to control the volume of the sound. For example the sound can start from the maximum volume and then it fades out gradually.

Enough with theory! Let's move on to practice! Let's load the csound-expression to the REPL and define the basic virtual midi instrument:

```
> ghci
Prelude> :m +Csound.Base
Prelude> :set -XFlexibleContexts
Prelude Csound.Base> let run f = vdac $ midi $ onMsg f
```

Wave generator

Wave generator defines the timbre content. What spectrum do we need in the sound? There are four standard waveforms: sine, sawtooth, square and triangle. The standard waveforms are represented with functions:

```
osc, saw, sqr, tri :: Sig -> Sig
```

All functions take in a frequency (it can vary with time, so it's a signal).

The most simple is sine wave or pure tone. It represents the sine function. In csound-expression the pure sine is generated with function `osc`. Let's listen to it.

```
> run osc
```

It starts to scream harshly when you press several notes. It happens due to distortion. Every signal is clipped to the amplitude of 1. The function `osc` generates waves of the amplitude 1. So when we press a single note it's fine. No distortion takes place. But when we press several notes it starts to scream because we add several waves and the amplitude goes beyond 1 and clipping results in distortion and leads to the harsh sound. If we want to press several keys we can scale the output sound:

```
> run $ mul 0.25 . osc
```

Pure tone contains only one partial in the spectrum. It's the most naked sound. We can make it a little bit more interesting with different waves. The next wave is triangle:

```
> run $ mul 0.25 . tri
```

Little bit more rich in harmonics is square wave:

```
> run $ mul 0.25 . sqr
```

The most rich is a saw wave:

```
> run $ mul 0.25 . saw
```

All sounds are very 8-bit and computer-like. That's because they are static and contain no variance. But that's only beginning. We can see that we are going to use the scaling all the time so why not to move it inside our runner function. Also we scale the pitch by 2 to make pitch lower:

```
> let run k f = vdac $ midi $ onMsg (mul k . f . (/ 2))
```

Now we can run the saw wave like this:

```
> run 0.25 saw
```

We can make our waves a little bit more interesting with additive synthesis. We can add together several waves (something that resembles harmonic series):

```
run 0.15 $ \x -> saw x + 0.25 * sqr (2 * x) + 0.1 * tri (3 * x)
```

Or we can introduce the higher harmonics:

```
run 0.15 $ \x -> saw x + 0.25 * tri (7 * x) + 0.15 * tri (13 * x)
```

Gain

Gain or amplifier can change the amplitude of the sound. We already did it. When we scale the sound with number it's an example of the gain. But instead of scaling with number we can give the output a shape. That's where the envelope generators come in the play.

Dynamic changes

To make our sounds more interesting we can vary it parameters in time. We are going to study two types of variations. They are slowly moving variations and rapid periodic ones. The former are envelope generators (EG) and former are Low frequency oscillators (LFO). Let's make our sound more interesting by shaping it's amplitude. That's how we change the volume in time.

Envelope generator

Envelope generators produce piecewise functions. Most often they are linear or exponential. In csound-expression we can produce piecewise functions with two function: `linseg` and `expseg`.

```
linseg, expseg :: [D] -> Sig
```

They take in a list of timestamps and values and produce piecewise signal. Here is an example:

Let's look at the input list:

```
linseg [a, t_ab, b, t_bc, c, t_cd, d, ...]
```

It constructs a function that starts with the value `a` then moves linearly to the value `b` for `t_ab` seconds, then goes from `b` to `c` in `t_bc` seconds and so on. For example, let's construct the function that starts at 0 then goes to 1 in 0.5 seconds, then proceeds to 0.5 in 2 seconds, and finally fades out to zero in 3 seconds:

```
linseg [0, 0.5, 1, 2, 0.5, 3, 0]
```

There are two usefull functions for midi instruments:

```
linsegr, expsegr :: [D] -> D -> D -> Sig
```

They take two additional parameters for release of the note. Second argument is a time of the release and the last argument is a final value. All values for `expsegr` should be positive.

For example we can construct a saw that slowly fades out after release:

```
run 0.25 $ \cps -> expsegr [0.001, 0.1, 1, 3, 0.5] 3 0.001 * saw cps
```

We can make a string-like sound with long fade in:

```
run 0.25 $ \cps -> linsegr [0.001, 1, 1, 3, 0.5] 3 0.001 * (tri cps + 0.5 * tri (2 * cps) + 0.1 * sqr (3 * cps))
```

ADSR envelope

Let's study the most common shape for envelope generators. It's attack-decay-sustain-release envelope (ADSR). This shape consists of four stages: attack, decay, sustain and release. In the attack amplitude goes from 0 to 1, in the decay it goes from 1 to specified sustain level and after note's release it fades out completely.

Here is a definition:

```
 adsr a d s r = linseg [0,      a, 1, d, s, r, 0]
xadsr a d s r = expseg [0.0001, a, 1, d, s, r, 0.0001]
```

There are two more function that wait for note release (usefull with midi-instruments):

```
 madsr a d s r = linsegr [0,      a, 1, d, s] r, 0
mxadsr a d s r = expsegr [0.0001, a, 1, d, s] r, 0.0001
```

The functions `madsr` and `mxadsr` are original Csound functions. They are used so often so there are short-cuts `leg` and `xeg`. They are linear and exponential envelope generators.

So we can express the previous example like this:

```
run 0.25 $ \cps -> leg 1 3 0.5 3 * saw cps
```

The EGs are for slowly changing control signals. Let's study some fast changing ones.

Low frequency oscillator

Low frequency oscillator is just a wave form (`osc`, `saw`, `sqr` or `tri`) with low frequency (0 to 20 Hz). It's inaudible when put directly to speakers but it can produce interesting results when it's used as a control signal.

Let's use it for vibrato:

```
run 0.25 $ \cps -> leg 1 3 0.5 0.7 * saw (cps * (1 + 0.02 * osc 5))
```

Or we can make a tremolo if we modify an amplitude:

```
run 0.25 $ \cps -> osc 5 * leg 1 3 0.5 0.7 * saw cps
```

The lfo-frequency can change over time:

```
run 0.25 $ \cps -> osc (5 * leg 1 1 0.2 3) * leg 2 3 0.5 0.7 * saw cps
```

Also we can change the shape of the LFO. We can use `saw`, `tri` or `sqr` in place of `osc`.

With EG's and LFO's we can make our instruments much more interesting. We can make them alive. They can control any parameter of the synth. We are aware of two types of control signals. We can alter pitch (vibrato) or result amplitude (amplitude envelope, tremolo). But there are many more parameters. Let's study new way of controlling sound. Let's study brightness.

There is a special function to make the LFOs more explicit:

```
type Lfo = Sig

lfo :: (Sig -> Sig) -> Sig -> Sig -> Lfo
lfo shape depth rate = depth * shape rate
```

It takes the waveform shape, depth of the LFO and rate as arguments.

Setting the range for changes

The LFOs are ranging in the interval (-1, 1). The EGs are ranging in the interval (0, 1). Often we want to change the range.

We can do t with simple arithmetic:

From `(0, 1)` to `(a, b)`:

```
> let y = a + b * x
```

Or from (-1, 1) to `(a, b)`:

```
> let y = a + b * (x + 1) / 2
```

It happens so often that there are special functions that abstracts these patterns:

From `(0, 1)` to `(a, b)`:

```
uon :: SigSpace a => Sig -> Sig -> a -> a
uon a b x = ...

let y = uon a b x
```

Or from (-1, 1) to `(a, b)`:

```
on :: SigSpace a => Sig -> Sig -> a -> a
on a b x = ...

let y = on a b x
```

The function `on` can be used with LFOs and `uon` can be used with EGs.

Looping envelope generators

Since the version 4.3 we can use a lot of looping envelope generators. They work as step sequencers.

Let's see how we can use LFO's to turn the sound in the patters of notes. Let's take a boring white noise and turn it in to equally spaced bursts:

```
> dac $ mul (usqr 4) white
```

We have multiplied the noise with unipolar square wave. We can change the shape of envelope if we multiply the noise with sawtooth wave:

```
> dac $ mul (usaw 4) white
```

We can reverse the envelope:

```
> dac $ mul (1 - usaw 4) white
```

We can create a simple drum pattern this way:

```
dac $ mul (usaw 2) white + mul (usqr 1 * (1 - usqr 4)) (return $ saw 50)
```

But the real drummer don't kicks all notes with the same volume we need a way to set accents. We can do it with special functions. They take in a list of accents and they scale the unipolar LFO-wave. Let's look at `sqrSeq`. It creates a sequence of squares which are scaled with given pattern:

```
> dac $ mul (sqrSeq [1, 0.5, 0.2, 0.5] 4) $ white
```

We can create another pattern for sawtooth wave:

```
> let b1 = mul (sqrSeq [1, 0.5, 0.2, 0.5] 4) $ white
> let b2 = mul (sawSeq [0, 0, 1] 2) $ white
> let b3 = return $ mul (triSeq [0, 0, 1, 0] 4) $ osc (sqrSeq [440, 330] 1)
> dac $ b1 + b2 + b3
```

We can use these functions not only for amplitudes. We can control other parameters as well.

```
> dac $ tri $ constSeq [220, 220 * 5/4, 330, 440] 8
```

The `constSeq` creates a sequence of constant segments. The cool thing about wave sequencers is that the values in the sequence are signals. We can change them easily.

```
> dac $ tri $ constSeq [220, 220 * 5/4, 330, constSeq [440, 220 * 4/ 3] 1] 8
```

```
> let b3 = return $ mul (triSeq [0, 0, 1, 0] 4) $ osc (stepSeq [440, 330] 0.25)
```

The function `stepSeq` creates a sequence of constant segments. The main difference with `constSeq` is that all values are placed in a single period. The period of `constSeq` is a single line but the period of `stepSeq` is the sequence of const segments. We can create arpeggiators this way:

Let's create a simple bass line:

```
> dac $ mlp (400 + 1500 * uosc 0.2) 0.1 $ saw (stepSeq [50, 50 * 9/ 8, 50 * 6 / 5, 50 * 1.5, 50, 50 * 9 / 8] 1)
```

We are using the function `mlp`. It's a moog low pass filter (the arguments: cut off frequency, resonance and the signal). We modulate the center frequency with LFO.

There are many more functions. We can create looping adsr sequences with `adsrSeq` and `xadsrSeq`. We can loop over generic line segments with `linSeq` and `expSeq`. We can create sample and hold envelopes with `sah`. We can find the functions in the module `Csound.Air.Envelope`.

Let's create a simple beat with step sequencers. The first line is the steady sound of kick drum:

```
> let kick = osc (100 * linloop [1, 0.1, 0, 0.9, 0])
> dac kick
```

The kick is a pure sine wave that is rapidly falls in pitch. We are using the function `linloop` to repeat the pitch changes. The `linloop` is just like `linseg` but it repeats over and over. Let's create a simple snare:

```
> let snare = at (hp 500 23) $ mul (sqrSeq [0, 0, 1, 0, 0, 0, 0.5, 0.2] 4) $ pink
> dac $ return kick + snare
```

We use high pass filtered pink noise. We create the drum pattern with square waves. The function `at` is the generic `map` for signal-like values:

```
at :: (SigSpace a) => (Sig -> Sig) -> a -> a
```

We wrap the kick in the `SE` monad to add it to the snare wave. Let's add a hi-hat. The hi-hat is going to be filtered white noise with sequence of saw envelopes:

```
> let hiHat = at (mlp 2500 0.1) $ mul (sawSeq [1, 0.5, 0.2, 0.5, 1, 0, 0, 0.5] 4) $ white
> dac $ mul 0.5 $ return kick + snare + hiHat
```

Let's add some pitched sounds. Also we can make the kick louder:

```
> let ticks = return $ mul (sqrSeq [0, 0, 0, 0, 1, 1] 8) $ osc 440
> dac $ mul 0.3 $ return (mul 2.4 kick) + ticks + snare + hiHat
```

Using GUIs as control signals

We can change parameters with UI-elements such as sliders and knobs. it's not the place to discuss GUIs at length. But I can show you a couple of tricks.

We have a simple audio wave:

```
> dac $ mlp 1500 0.1 $ saw 110
```

It's a filtered sawtooth wave. Let's plugin a knob to change the volume:

```
> dac $ lift1 (\amp -> mul amp $ mlp 1500 0.1 $ saw 110) $ uknob 0.5
```

The `uknob` creates a knob that outputs a unipolar signal (it belongs to the interval `[0, 1]`). The argument is the initial value of the knob. The `lift1` maps over the value of the `knob`. The `uknob` returns not the signal itself but the signal and the GUI-element. With lift1 we can easily transform control signal to audio wave.

What if we want to change the frequency? It's best to change the frequency with eXponential control signals (the change is not linear but exponential). we can use the function `xknob`:

```
> dac $ hlift2 (\amp cps -> mul amp $ mlp 1500 0.1 $ saw cps) (uknob 0.5) (xknob (50, 600) 110)
```

The `xknob` takes in three values. They are the minimum and maximum values and the initial value. The `hlift2` can join two UI-control signals with functions. It aligns the visual representation horizontally. The `vlift2` aligns visuals vertically.

Let's change the parameters of the filter with sliders:

```
> dac $ vlift2 (\(amp, cps) (cflt, q)  -> mul amp $ mlp cflt q $ saw cps)
    (hlift2 (,) (uknob 0.5) (xknob (50, 600) 110))
    (vlift2 (,) (xslider (250, 7000) 1500) (mul 0.95 $ uslider 0.5))
```

We can see the picture of the talking robot. The `uslider` and `xslider` work just like `uknob` and `xknob` but they lokk like sliders. Notice the scaling of the value of the second slider with `mul`. It's as simple as that.

There are functions `hlift3`, `hlift4` and `hlift4` to combine more widgets. The `hlift2'`, `hlift3'` . Notice the last character also take in scaling parameters for visual objects. We can define four knobs with different sizes:

```
> dac $ mul 0.5 $ hlift4' 8 4 2 1
    (\a b c d -> saw 50 + osc (50 + 3 * a) + osc (50 + 3 * b) + osc (50 + c) + osc (50 + d))
    (uknob 0.5) (uknob 0.5) (uknob 0.5) (uknob 0.5)
```

Another usefull widget is ujoy. It creates a couple of signal which control xy coordinates on the plane:

```
> dac $ lift1 (\(a, b) -> mlp (400 + a * 5000) (0.95 * b) $ saw 110) $ ujoy (0.5, 0.5)
```

To use exponential control signals wwe should try the function `joy`:

```
joy :: ValSpan -> ValSpan -> (Double, Double) -> Source (Sig, Sig)
```

The `ValSpan` can be linear or exponential. Both functions take in minimum and maximum values:

```
linSpan, expSpan :: Dounle -> Double -> ValSpan
```

Let's look at the simple example:

```
> dac $ lift1 (\(amp, cps) -> amp * tri cps) $ joy (linSpan 0 1) (expSpan 50 600) (0.5, 110)
```

Filter

We can control brightness of the sound with filters. A filter can amplify or attenuate some harmonics in the spectrum. There are four standard types of filters:

Low pass filter (LP) attenuates all harmonics higher than a given center frequency.

High pass filter (HP) attenuates all harmonics lower than a given center frequency.

Band pass filter (BP) amplifies harmonics that are close to center frequency and attenuates all harmonics that are far away.

Band reject filter or notch filter (BR) does the opposite to the BP-filter. It attenuates all harmonics that are close to the center frequency.

A filter is very important for the synth. The trade mark of the synth is defined by the quality of its filters.

The strength of attenuation is represented by the ratio of how much decibels the harmonic is weaker per octave from the center frequency. The greater the number the stronger the filter.

In csound-expression there are plenty of filters. Standard filters are:

```
lp, hp, bp, br :: Sig -> Sig -> Sig -> Sig
```

The first parameter is center frequency, the second one is resonance and the last argument is the signal to modify.

There is an emulation of the Moog low pass filter:

```
mlp :: Sig -> Sig -> Sig -> Sig
```

The arguments are: central frequency, resonance, the input signal.

We can change parameters in real-time with EG's and LFO's. Let's create an envelope and apply it to the amplitude and center frequency:

```
> let env = leg 0.1 0.5 0.3 1
> run (0.15 * env) (lp (1500 * env) 1.5 . saw)
```

Normal values for resonance range from 1 to 100. We should carefully adjust the scaling factor after filtering. Filters change the volume of the signal.

We can align the center frequency with pitch. So that if we make pitch higher the center frequency gets higher and we get more bright sounds:

```
> run (0.15 * env) (\x -> lp (x + 500 * env) 3.5 $ saw x)
```

We can make a waveform more interesting with new partials.

```
> run (0.1 * env) (\x -> lp (x + 2500 * env) 3.5 $ saw x + 0.3 * tri (3 * x) + 0.1 * tri (4 * x))
```

We can apply an LFO to the resonance.

```
run (0.15 * env) (\x -> lp (x + 500 * env) (7 + 3 * sqr 4) $ saw x)
```

Also we can apply LFO to the frequency:

```
run (0.15 * env) (\x -> lp (x + 500 * env) (7 + 3 * sqr 4) $ saw (x * (1 + 0.1 * osc 4)))
```

We can increase an order of the resonant filter applying it several times. There is a function `filt` that does it:

```
run (0.15 * env) (\x -> filt 2 lp (x + 500 * env) (3 + 2 * sqr 4) $ saw x)
```

You can find lots of filters in the module `Csound.Air.Filter`.

Effects

We can make our sounds much more interesting with effects! Effect transforms the sound of the instrument in some way. There are several groups of effects. Some of them affect only amplitude, while the other alter frequency or phase or place sound in acoustic environment.

To apply effect to the sound we have to modify our runner function. Right now all arguments control the sound that is produced with the single note. But we want to alter the total sound that goes out of the instrument. It includes the mixed sound from all notes that are played. Let's modify our definition for function `run`:

```
let run eff k f = vdac $ (eff =<< ) $ midi $ onMsg (mul k . f . (/ 2))
```

The first argument now applies some effect to the output signal.

Time/Based

Reverb

Reverb places the sound in some room, cave or hall. We can apply reverb with function `reverTime`:

```
reverTime :: Sig -> Sig -> Sig
```

It expects the reverb time (in seconds) as a first argument and the signal as the second argument.

```
run (return . reverTime 1.5) (0.05 * env) (\x -> lp (x + 500 * env) (7 + 3 * sqr 4) $ saw x)
```

There is also a function `rever1`:

```
rever1 :: Sig -> Sig -> (Sig, Sig)
```

It's base on very cool Csound unit `reverbsc`. It takes in feedback level (0 to 1) and input signal and produces the processed output. There are several ready to use shortcuts: `smallRoom`, `smallHall`, `largeRoom`, `largeHall` and `magicCave`.

Let's place our sound in the magic cave:

```
run (return . magicCave) (0.05 * env) (\x -> lp (x + 500 * env) (7 + 3 * sqr 4) $ saw x)
```

You can hear how dramatically an effect can change the sound.

Delay

Delay adds some echoes to the sound. the simplest function is `echo`:

```
echo :: D -> Sig -> Sig -> SE Sig
echo dt fb asig
```

It takes the delay time, the ratio of signal attenuation (reflections will be weaker by this amount) and the input signal. Notice that the output is wrapped in the `SE`-monad. `SE` means side effect. It describes some nasty impure things. This function allocates the buffer of memory to hold the delayed signal. So thats why the output contains side-effects.

Let's try it out:

```
run (echo  0.5 0.4) (0.05 * env) (\x -> lp (x + 500 * env) (7 + 3 * sqr 4) $ saw x)
```

Let's add some reverberation:

```
run (fmap smallHall . echo  0.5 0.4) (0.05 * env) (\x -> lp (x + 500 * env) (7 + 3 * sqr 4) $ saw x)
```

We are using the `fmap` function to apply the next effect in chain to the value with side-effects. The `SE`-wrapper type is `Monad` and hence it's `Applicative` and `Functor`. The `echo` function is a specification of generic function:

```
fdelay :: D -> Sig -> Sig -> Sig -> SE Sig
fdelay len fbk mix asig
```

It takes a delay time, ratio of sound attenuation, the mix level (we add the initial sound with processed one which is scaled by amount of `mix`) and the input signal.

There is the last most generic function `fvdelay`. With it we can vary the delay time:

```
fvdelay :: D -> Sig -> Sig -> Sig -> Sig -> SE Sig
fvdelay maxDelTime delTime fbk mix asig
```

It takes the maximum delay time and the delay time which is signal (it must be bounded by `maxDelTime`). Other arguments are the same.

Multitap delays can be achieved with function

```
fvdelays :: D -> [(Sig, Sig)] -> Sig -> Sig -> SE Sig
fvdelays maxDelTime delTimeAndFbk  mix asig
```

The list holds tuples of delay times and attenuation ratio for each delay line.

Distortion

A distortion can make our sound scream. We can use the function

```
distortion :: Sig -> Sig -> Sig
distortion gain asig
```

It takes a distortion level as first parameter. It ranges from 1 to infinity. The bigger it is the harsher the sound.

Pitch/Frequency

Let's review briefly some other cool effects.

Chorus

Chorus makes sound more natural by adding slightly transformed versions of the original sound:

```
chorus :: Sig -> Sig -> Sig -> SE Sig
chorus rate depth asig
```

Beside the input signal chorus takes two arguments that range from 0 to 1. They represent the chorus rate and depth.

Flanger

The next two effects are useful for creating synthetic sounds or adding electronic flavor to the natural sounds.

The flanger can be applied with function `flange`:

```
flange :: Lfo -> Sig -> Sig -> Sig -> Sig -> Sig
flange lfo fbk mx asig
```

Where arguments are: an LFO signal, feedback level, balance level between pure and processed signals and an input signal.

Let's apply a flanger:

```
run (return . flange (lfo tri 0.9 0.05) 0.9 0.5) (0.05 * env) (\x -> lp (x + 500 * env) (7 + 3 * sqr 4) $ saw x)
```

Phaser

The phaser is a special case of flanger effect. It processes the signal with series of all-pass filters. We can simulate a sweeping phase effect with phaser.

There are three types of phasers. The simplest one is

```
phase1 :: Sig -> Lfo -> Sig -> Sig -> Sig -> Sig
phase1 ord lfo fbk mx asig
```

The arguments are: the order of phaser (an integer value, it represents the number of all-pass filters in chain, 4 to 2000, the better is 8, the bigger the number the slower is algorithm), an LFO for phase sweeps (depth is in range acoustic waves, something around 5000 is good start, the rate is something between 0 and 20 Hz), amount of feedback, the balance between pure and processed signals, the input signal.

There are two more phasers:

```
harmPhase, powerPhase :: Sig -> Lfo -> Sig -> Sig -> Sig -> Sig -> Sig -> Sig
harmPhase ord lfo q sep fbk mx asig = ...
```

The arguments are: order of phaser, LFO-signal for frequency sweep, resonance of the filters (0 to 1), separation of the peaks, feedback level (0 to 1), balance level.

Noise

We can make our sounds more interesting by introducing randomness. There are several ways to create random signals (including noise).

We can create a sequence of random numbers that change linearly with given frequency. Also this unit can be used as LFO.

```
rndi, urndi :: Sig -> SE Sig

 rndi frequency
urndi frequency
```

The `urnds`varies between 0 and 1. The `rnds` varies between -1 and 1.

We can generate colored noises with:

```
white, pink :: SE Sig
```

Let's create a simple wind instrument:

```
> let simpleWind x = do { cfq <- 2000 * urndi 0.5; asig <- white; return $ mlp (x + cfq) 0.6 asig }
```

We filter the white noise with filter. The center frequency randomly varies above the certain threshold. Let's hear the wind:

```
dac $ mul (fadeIn 0.5) $ simpleWind 500
```

Complex waves

Let's study how we can made our waveforms more interesting. We can apply several simple techniques to achieve it.

Reading sound signals from files

We can reuse the sound signals. The music is everywhere and we can take a somebody else's music as a start point.

There are handy functions for reading the sound from files:

```
readSnd :: String -> (Sig, Sig)
loopSnd :: String -> (Sig, Sig)

readSnd fileName = ...
```

The `readSnd` plays the file only once. The `loopSnd` repeats the file over and over again. There is another useful function:

```
loopSndBy :: D -> String -> (Sig, Sig)
```

It takes the duration of the loop-period as a first argument.

These functions can read files in many formats including `wav` and `mp3`. If your sound sample is stored in the `wav` or `aiff` format we can read it with the given speed. The speed is a signal. It can change with time. We can create interesting effects with it:

```
loopWav :: Sig -> String -> (Sig, Sig)
loopWav speed fileName = ...
```

The normal playback is a speed that equals `1`. We can play it in reverse if we set the speed to `-1`.

The output is a stereo signal. If we want to force it to mono we can use the function:

```
toMono :: (Sig, Sig) -> Sig
```

It produces the mean of two signals.

Additive synthesis

The simplest one is additive synthesis. We add two or more waveforms so that they form harmonic series.

```
> run return (0.25 * env) (\x -> saw x + 0.5 * sqr (2 * x) + 0.15 * tri (3 * x))
```

Stacking together several waveforms

When several violins play in the orchestra the timbre is quite different from the sound of the single violin. Though timbre of each instrument is roughly the same the result is different. It happens from the slightly detuned sound of the instruments. We can recreate this effect by stacking together several waveforms that are slightly detuned. It can be achieved with function:

```
chorusPitch :: Int -> Sig -> (Sig -> Sig) -> (Sig -> Sig)
chorusPitch numberOfCopies chorusWidth wave = ...
```

It takes the integer number of copies and chorus width. Chorus width specifies the radius of the detunement.

```
> run return (0.25 * env) (chorusPitch 8 0.5 saw)
```

Ring modulation

Ring modulation can add metallic flavor to the sound. It multiplies the amplitude of the signal by LFO.

```
run return (0.25 * env) (mul (osc (30 * env)) . chorusPitch 8 0.5 saw)
```

Diving deeper

Csound contains thousands of audio algorithms. It's impossible to cover them all in depth in the short guide. But we can explore them. They reside in the separate package `csound-expression-opcodes` that is re-exported by the module `Csound.Base`. Take a look in the docs. there are links to the originall Csound docs. Maybe you can find your own unique sound somewhere in this wonderful forest of algorithms.

The modules `Csound.Typed.Opcode.SignalGenerators`, `Csound.Typed.Opcode.SignalModifiers` and `Csound.Typed.Opcode.SpectralProcessing` are good place to start the journey.

- <= Rendering Csound files
- => User interaction
- Home

User interaction

Let's explore the ways of how we can interact with instruments.

Midi-instruments

The simplest way to create a responsive instrument is to make a midi-instrument. A Midi-instrument is something that expects a midi-message and produces sound output. A midi-message contains pitch and volume of the note and possibly some control data (to change the parameters of the synth).

The midi-message is represented with opaque type:

```
data Msg

cpsmidi :: Msg -> D          -- extract frequency (Hz)
ampmidi :: Msg -> D -> D     -- extract amplitude (0 to second argument)

ampCps  :: Msg -> (D, D)     -- ampmidi & cpsmidi, amplitude is 0 to 1
```

We can extract amplitude and frequency (Hz) with function `ampCps`.

The midi-intrument listens for message on the specified channel (It's an integer from 1 to 16):

```
type Channel = Int
```

The simplest function for midi instruments is:

```
midi :: Sigs a => (Msg -> SE a) -> SE a
```

It creates a signal that is produced from the output of midi-instrument. A midi-instrument listens for messages on all channels.

There are two more refined functions:

```
midin  :: Sigs a => Channel -> (Msg -> SE a) -> SE a
pgmidi :: Sigs a => Maybe Int -> Channel -> (Msg -> SE a) -> SE a
```

They allow to specify a midi-channel and probably a midi-program. Shortly after creation of the Midi-protocol it was understood that 1 to 16 channels is not enough. So there come the programs. You can specify a midi instrument with 16 channels and 128 programs. We can specify a program with function `pgmidi`.

If you have a real midi-keyboard connected to your computer (most often with USB) you can start to play along with it in csound-expression. Just type:

```
> ghci
> :m +Csound.Base
> let instr msg = return $ 0.25 * (fades 0.1 0.5) * (sig $ ampmidi msg 1) * saw (sig $ cpsmidi msg)
> dac $ mixAt 0.25 smallRoom2 $ fmap fromMono $ midi instr
```

If we don't have the midi-device we can test the instrument with virtual one. We need to use `vdac` in place of `dac`:

```
> vdac $ mixAt 0.25 smallRoom2 $ fmap fromMono $ midi instr
```

We have created a simple saw-based instrument. The function `fades` adds the attack and release phase for the instrument. It fades in with time of the first argument and fades out after release with time of the second argument. We used a lot the function `sig :: D -> Sig`. It's just a converter. It constructs signals from the constant values. The function `fromMono` converts mono signal to stereo. The `mixAt` pplies an effect with given dry/wet ratio. The value 0 is all dry and the 1 is all wet.

Yo can notice how long and boring the expression for the instrument is. Instrument expects a midi-message. Then we have to extract amplitude and frequncy and convert it to signals and apply to the instrument. It's a typical pattern that repeats over and over again. There is a type class that converts functions to midi-instruments. It's called `MidiInstr`:

```
class MidiInstr a where
    type MidiInstrOut a :: *
    onMsg :: a -> Msg -> SE (MidiInstrOut a)
```

It converts a value of type `a` to midi-instrument. There are plenty of instances for this class. We can check them out in the docs. Among them we can find the instance for the type:

```
Sig -> Sig
```

It's assumed that single argument is a frequency (Hz). This instrument is a wave-form. To convert it to midi-instrument we apply midi-frequency to it (it's converted to signal) and scale it with midi-amplitude. So we can redefine our instrument like this:

```
> let instr = onMsg $ mul (0.25 * fades 0.1 1) . saw
> dac $ mixAt 0.25 smallRoom2 $ fmap fromMono $ midi instr
```

The function `mul` scales the signal-output like types. They are all tuples of `Sig` probably wrapped in the type `SE`.

Continuous midi-instruments

So far every midi-instrument has triggered the instrument in the separate note instance. In the end we get the sum of all notes. It's polyphonic mode. But what if we want to use synth in monophonic mode. So that frequency and amplitude are continuous signals that we can use in the other instruments.

There are two functions for this mode:

```
data MidiChn = ChnAll | Chn Int | Pgm (Maybe Int) Int

monoMsg :: MidiChn -> D -> D -> SE (Sig, Sig)
monoMsg portamentoTime releaseTime

holdMsg :: MidiChn -> D -> SE (Sig, Sig)
holdMsg portamentoTime
```

Both of them produce amplitude and frequency as time varied signals. The former fades out when nothing is pressed and the latter holds the last value until the next one is present.

The first argument for both of them is specification of the midi channel. The second argument is portamento time. It's time in second that it takes for transition from one value to another. The function `monoMsg` takes another parameter that specifies a release time. Time it takes for the note to fade out or fade in.

Let's play with these functions:

```
> vdac $ fmap smallRoom $ fmap (\(amp, cps) -> amp * tri cps) $ monoMsg ChnAll 0.1 1
> vdac $ fmap smallRoom $ fmap (\(amp, cps) -> amp * tri cps) $ holdMsg ChnAll 0.5
```

Midi-controls

If our midi-device has some sliders or knobs we can send the control-messages. Control messages allow us to change parameters for the instruments during performance.

We can use the function `ctrl7`:

```
ctrl7 :: D -> D -> D -> D -> Sig
ctrl7 chno ctrlId imin imax
```

It expects the channel number (where we listen for the control messages), the identity number of control parameter, and two parameters for minimum and maximum of the output range. Let's apply the filter to the output of the previous example:

```
> vdac $ fmap smallRoom $ fmap (\(amp, cps) -> amp * mlp (ctrl7 1 1 50 5000) (ctrl7 1 2 0.1 0.9) (tri cps)) $ holdMsg 0.5
```

You can try to use the first slider at the virtual midi. It should control the filter parameters in real-time.

Another function that is worth to mention is:

```
initc7 :: D -> D -> D -> SE ()
initc7 chno ctrlId val              -- value ranges from 0 to 1
```

It sets the initial value for the midi control.

```
> let ctrl = 1
> let out = fmap smallRoom $ fmap (\(amp, cps) -> amp * mlp (ctrl7 1 ctrl 50 5000) 0.5 (tri cps)) $ holdMsg ChnAll 0.5
> dac $ do { initc7 1 ctrl 0.5; out }
```

Unfortunately the function `initc7` doesn't work with virtual midi. It's only for real midi-devices.

There are three more functions to make things more easy:

```
midiCtrl7 :: D -> D -> D -> D -> D -> SE Sig
midiCtrl7 chanNum ctrlNum initVal min max
```

It combines the functions `ctrl7` and `initc7`. So that we don't have to specify the same channel number and control number twice.

There are functions for specific ranges

```
midiCtrl, umidiCtrl :: D -> D -> D -> SE Sig
```

They are the same as midiCtrl7, but former sets the range to `[-1, 1]` and the latter to `[0, 1]`.

Basics of GUI

If we don't have real sliders and knobs we can use the virtual ones. It can be done easily with GUI-elements. Csound has support for GUI-widgets. GUI-widgets live in the module `Csound.Control.Gui`.

Let's study how can we use them. First of all let's define the notion of widget. A widget is something that contains graphical representation (it's what we see on the screen) and behaviour (what we can do with it).

A slider for instance is represented as a moving small line segment in the box. It's a graphical representation of the slider. At the same time the slider can give us a time varying signal. It's behaviour of the slider. There are different types of behaviour. Some widgets can produce the values (like sliders or buttons). They are sources. Some widgets can wait for the value (like text box that shows the value on the screen). They are sinks. Some widgets can do all this in the same time and some widgets can do neither (like static text. It's only visible but it can not do anything).

In the Haskell type system we can express it like this:

```
data Gui    -- visual representation

type Widget a b = SE (Gui, Output a, Input b, Inner)

type Input a = a                -- produces a value
type Output a = a -> SE ()      -- waits for a value
type Inner = SE ()              -- does smth useful
type Sink a = SE (Gui, Output a)    -- value consumer
type Source a = SE (Gui, Input a)   -- value producer
type Display = SE Gui               -- static element
```

Let's look at the definition of the slider:

```
slider :: String -> ValSpan -> Double -> Source Sig
slider tag valueRange initValue
```

The slider expects a tag-name, value range and initial value. It produces a `Source`-widget that contains a signal.

The value type specifies the value range and the type of the change of the value (it can be linear or exponential).

```
linSpan, expSpan :: Double -> Double -> ValSpan

linSpan min max
expSpan min max
```

Let's define a slider in the ghci:

```
> let vol = slider "volume" (linSpan 0 1) 0.5
> dac $ do { (gui, v) <- vol; panel gui; return (v * osc 440) }
```

We can control the volume of the concert A note with the slider! To see the slider we have to place it on the window. That is why we used the function `pannel`:

```
pannel :: Gui -> SE ()
```

It creates a window and renders the graphical representation of the GUI on it. You can notice the strange quirk of the slider it updates the values in reverse. The top is lowest value and the bottom is for the highest value. It's strange implementation of the vertical sliders in the Csound. We can only take it for granted.

Ok. That it's good but how about using two sliders at the same time? We can create the second slider and place it right beside the other with function `hor`. It groups a list of widgets and shows them side by side:

```
> let vol = slider "volume" (linSpan 0 1) 0.5
> let pch = slider "pitch" (expSpan 20 3000) 440
> dac $ do { (vgui, v) <- vol; (pgui, p) <- pch ; panel (hor [vgui, pgui]); return (v * osc p) }
```

Try to substitute `hor` for `ver` and see what happens.

The layout functions

We can see how easy it's to use the `hor` and `ver`. Let's study all layout functions:

```
hor :: [Gui] -> Gui
ver :: [Gui] -> Gui

space :: Gui
sca   :: Double -> Gui -> Gui

padding :: Int -> Gui -> Gui
margin  :: Int -> Gui -> Gui
```

The functions `hor` and `ver` are for horizontal and vertical grouping of the elements. The `space` creates an empty space. The `sca` can scale GUIs. The `margin` and `padding` are well .. mm .. for setting the margin and padding of the element in pixels.

We can stack as many sliders as we want. Let's explore the low-pass filtering of the saw waveform.

```
> let cfq = slider "center frequency" (expSpan 100 5000) 2000
> let q = slider "resonance" (linSpan 0.1 0.9) 0.5
> dac $ do {
    (vgui, v) <- vol;
    (pgui, p) <- pch;
    (cgui, c) <- cfq;
    (qgui, qv) <- q;
    panel (ver [vgui, pgui, cgui, qgui]);
    return (v * mlp c qv (saw p))
}
```

I've added some formatting for readability in the last line. But to make it work in the ghci you have to type it a single line.

Applicative style GUIs

Let's turn back to the example with pitch and volume sliders:

```
> let vol = slider "volume" (linSpan 0 1) 0.5
> let pch = slider "pitch" (expSpan 20 3000) 440
> dac $ do { (vgui, v) <- vol; (pgui, p) <- pch ; panel (ver [vgui, pgui]); return (v * osc p) }
```

There is a much more convenient way of writing widgets like this. We can use applicative style GUIs. There are functions that combine visual representation and behavior of the UIs at the same time:

```
lift1 :: (a -> b) -> Source a -> Source b

hlift2 :: (a -> b -> c) -> Source a -> Source b -> Source c
vlift2 :: (a -> b -> c) -> Source a -> Source b -> Source c

hlift3 :: (a -> b -> c -> d) -> Source a -> Source b -> Source c -> Source d
vlift3 :: (a -> b -> c -> d) -> Source a -> Source b -> Source c -> Source d
```

```
hlift4, vlift4 :: ...
hlift5, vlift5 :: ...
```

It takes a function to combine the outputs of the widget and the prefix is responsible for catenation of the visuals. `h` -- means horizontal and `v` -- vertical.

Let's rewrite the last line of our example:

```
> dac $ vlift2 (\v p -> v * tri p) vol pch
```

The `(Sigs a => Source a)` is also renderable type and we can apply `dac` to it. The cool thing is that result of the `vlift2` is an ordinary source-widget. We can apply another `lift`-function to it.

Let's add a filter:

```
> let cfq = slider "center frequency" (expSpan 100 5000) 2000
> let q = slider "resonance" (linSpan 0.1 0.9) 0.5
> let filter = vlift2 (\cps res -> mlp cps res) cfq q
```

Notice that our widget produces a function. Like any real functional programming language Haskell can do it! Let's apply the filter:

```
> let wave = vlift2 (\v p -> v * tri p) vol pch
> dac $ hlift2 ($) filter wave
```

Also there are functions to stack a list of similar widgets:

```
hlifts, vlifts :: ([a] -> b) -> [Source a] -> Source b
```

Let's create a widget to study harmonics:

```
> let harmonics cps weights = mul (1.3 / sum weights) $
        sum $ zipWith (\n w -> w * osc (cps * n))(fmap (sig . int) [1 .. ]) weights
> dac $ mul 0.75 $ hlifts (harmonics 110) (uslider 0.75 : (replicate 9 $ uslider 0))
```

The `uslider` is convenient alias for creation of linear unipolar anonymous sliders. The only value it takes is an initial value. That's it! We have created a widget with ten sliders for harmonic series exploration with just two lines of code! Also there is a function `xslider` for exponential anonymous sliders. It has the arguments:

```
type Range a = (a, a)

xslider :: Range Double -> Double -> Source Sig
xslider (min, max) init
```

The same functions are defined for knobs: `uknob`, `xknob`.

Let's add a couple of controls. We want to change pitch and volume. We already have the required sliders `vol` and `pch`:

```
> let harms = hlifts (flip harmonics) (replicate 10 $ uslider 0)
> dac $ vlift3 (\amp cps f -> amp * f cps) vol pch harms
```

it's ok by the audio but the picture is ugly. The problem is that `vlift3` gives the same amount of space to all widgets. But we want to change the proportions. Right for this task there are functions:

```
hlift2', vlift2' :: Double -> Double -> (a -> b -> c) -> Source a -> Source b -> Source c

hlift3', vlift3' :: Double -> Double -> Double -> (a -> b -> c -> d) -> Source a -> Source b -> Source c -> Source d

hlift4', vlift4' :: ...
hlift5', vlift5' :: ...

vlifts', hlifts' :: [Double] -> ([a] -> b) -> [Source a] -> Source b
```

They take in scaling factors for each widget. Let's see how they can help us to solve the problem:

```
> dac $ vlift3' 0.15 0.15 1 (\amp cps f -> amp * f cps) vol pch harms
```

Widgets

Knobs

There are many more widgets. Let's turn some sliders into knobs. The knob is a sort of circular slider:

```
> let vol = knob "volume" (linSpan 0 1) 0.5
> let pch = knob "pitch" (expSpan 20 3000) 440
> dac $ do {
    (vgui, v) <- vol;
    (pgui, p) <- pch;
    (cgui, c) <- cfq;
    (qgui, qv) <- q;
    panel (ver [vgui, pgui, hor [cgui, qgui]]);
    return (v * mlp c qv (saw p))
}
```

Now the sliders look to big we can change it with function `sca`:

```
    ...
    panel (ver [vgui, pgui, sca 1.5 $ hor [cgui, qgui]]);
    ...
```

Also there are aliases. Produces unipolar linear anonymous knob:

```
uknob :: Double -> Source Sig
uknob initVal = ...
```

Produces exponential anonymous knob:

```
type Range a = (a, a)

xknob :: Range Double -> Double -> Source Sig
xknob (min, max) init
```

Numeric values

Numeric creates a time varying signal like a slider. But it's graphical representation is different. It's a box with a number inside it. You can change the value by dragging the mouse from the box.

```
numeric :: String -> ValDiap -> ValStep -> Double -> Source Sig
numeric tag valueDiapason valueStep initialValue
```

Buttons

Let's create a switch button. We can use a `toggleSig` for it:

```
toggleSig :: String -> Bool -> Source Sig
```

This function just creates a button that produces a signal that is 1 whenthe button is on and 0 when it's off. The button is initialized with value Bool.

```
> let switch = toggleSig "On/Off" true
> dac $ do {
    (sgui, sw) <- switch;
    (vgui, v) <- vol;
    (pgui, p) <- pch;
    (cgui, c) <- cfq;
    (qgui, qv) <- q;
    panel (ver [vgui, pgui, hor [sgui, cgui, qgui]]);
    return (sw * v * mlp c qv (saw p))
}
```

We can make the gradual change wit portamento:

```
...
    return (port sw 0.7 * v * mlp c qv (saw p))
```

```
...
```

Buttons can produce the event streams:

```
button :: String -> Source (Evt Unit)
```

The event stream `Evt a` is something that can apply a procedure of the type `a -> SE ()` to the value when it happens.

There is a function:

```
runEvt :: Evt a -> (a -> SE ()) -> SE ()
```

Also event streams can trigger notes with:

```
trig  :: (Arg a, Sigs b) => (a -> SE b) -> Evt (D, D, a) -> b
sched :: (Arg a, Sigs b) => (a -> SE b) -> Evt (D, a) -> b
```

The function `trig` invokes an instrument `a -> SE b` when the event happens. The note is a triple `(D, D, a)`. It's `(delayTime, durationTime, instrumentArgument)`. The function `sched` is just like `trig` but delay time is set to zero for all events. So that we need only a pair in place of the triple.

Let's create two buttons that play notes:

```
> let n1 = button "330"
> let n2 = button "440"
> let instr x = return $ fades 0.1 0.5 * osc x
> let go x evt = sched (const $ instr x) (withDur 2 evt)
> dac $ do {
    (g1, p1) <- n1;
    (g2, p2) <- n2;
    panel $ hor [g1, g2];
    return $ mul 0.25 $ go 330 p1 + go 440 p2
}
```

The new function `withDur` turns a single value into pair that contsants a duration of the note in the first cell.

We can do it with a little bit more simple expression if we know that events are functors and monoids. With Monoid's append we can get a single event stream that contains events from both event streams.

Let's redefine our buttons:

```
> let n1 = mapSource (fmap (const (330 :: D))) $ button "330"
> let n2 = mapSource (fmap (const (440 :: D))) $ button "440"
```

The function `mapSource` maps over the value of the producer widget. Right now every stream contains a value for the frequency with it. Let's merge two streams together and invoke the instrument on the single stream. The result should be the same:

```
> let instr x = return $ fades 0.1 0.5 * osc x
> dac $ do {
    (g1, p1) <- n1;
    (g2, p2) <- n2;
    panel (hor [g1, g2]);
    return $ mul 0.25 $ sched (instr . sig) (withDur 2 $ p1 <> p2)
}
```

Box

With boxes we can just show the user some message.

```
box :: String -> Display
```

Let's say something to the user.

```
> dac $ do {
    gmsg <- box "Two buttons. Here we are."
    (g1, p1) <- n1;
    (g2, p2) <- n2;
    panel (ver [gmsg, hor [g1, g2]]);
    return $ mul 0.25 $ sched (instr . sig) (withDur 2 $ p1 <> p2)
}
```

Radio-buttons

Radio buttons let the user select a value from the set of choices.

```
radioButton :: Arg a => String -> [(String, a)] -> Int -> Source (Evt a)
```

Let's redefine our previous example:

```
> let ns = radioButton "two notes" [("330", 330 :: D), ("440", 440)] 0
> dac $ do {
    (gui, p) <- ns;
    panel gui;
    return $ mul 0.25 $ sched (instr . sig) (withDur 2 p)
}
```

Meter

We have studied a lot of sources. Is there any sink-widgets? The `meter` is the one. It let's us monitor the value of the signal: It shows the output as the slider:

```
> let sa = slider "a" (linSpan 1 10) 5
> let sb = slider "b" (linSpan 1 10) 5
> let res = setNumeric "a + b" (linDiap 2 20) 1 10
> dac $ do {
    (ga, a) <- sa;
    (gb, b) <- sb;
    (gres, r) <- res;
    panel $ ver [ga, gb, gres];
    r (a + b)
}
```

Making reusable widgets

We can make reusable widgets with functions:

```
sink    :: SE (Gui, Output a) -> Sink a
source  :: SE (Gui, Input a) -> Source a
display :: SE Gui -> Display
```

Let's make a reusable widget for a Moog low-pass filter. It's a producer or source. It's going to produce a transformation `Sig -> Sig`:

```
import Csound.Base

mlpWidget :: Source (Sig -> Sig)
mlpWidget = source $ do
    (gcfq, cfq) <- slider "center frequency" (expSpan 100 5000)  2000
    (gq,   q)   <- slider "resonance"        (linSpan 0.01 0.9)  0.5
    return (ver [gcfq, gq], mlp cfq q)
```

Let's save this definition in the file and load it in ghci. Now we can use it as a custom widget:

```
> dac $ do {
    (g, filt) <- mlpWidget;
    panel g;
    return $ mul 0.5 $ filt $ saw 220
}
```

Notice that a widget can produce a function as a value!

Let's define another widget for saw-oscillator:

```
sawWidget :: Source Sig
sawWidget = source $ do
```

```
    (gamp, amp) <- slider "amplitude" (linSpan 0 1) 0.5
    (gcps, cps) <- slider "frequency" (expSpan 50 10000) 220
    return (ver [gamp, gcps], amp * saw cps)
```

Now let's use them together:

```
 dac $ do {
    (gw, wave) <- sawWidget;
    (gf, filt) <- mlpWidget;
    panel $ ver [gw, gf];
    return $ filt wave
 }
```

Open sound control protocol (OSC)

Open sound control is a modern data transfer protocol that should supersede the Midi protocol. It's much more lightweight and efficient. It can be used over network to orchestrate a lot of instruments.

We can send or receive the data over network on the specified port. We should declare the port, the address of the data and the type of the expected data.

The port is an integer. The address is a path-like string:

```
"/foo/bar"
"/note"
```

The type of the data is a string of special characters. The string can contain the characters "cdfhis" which stand for character, double, float, 64-bit integer, 32-bit integer, and string.

There are special type synonyms for all these terms:

```
type OscPort = Int
type OscAddress = String
type OscType = String
type OscHost = String
```

There are two modes. We can listen for the OSC-messages or we can send them.

Listening for messages

To listen for the events we have to create a background process. It waits for messages on the given port:

```
initOsc :: OscPort -> OscRef
initOsc port
```

We can specify an integer port. It gives us a reference to the process which should be used in the function `listenOsc`:

```
listenOsc :: Tuple a => OscRef -> OscAddress -> OscType -> Evt a
listenOsc ref addr type =
```

The function `listenOsc` produces a stream of OSC-messages that are coming on the given port, address and have a certain type.

Sending messages

To send OSC-messages we can use the function `sendOsc`:

```
sendOsc :: Tuple a => OscHost -> OscPort -> OscAddress -> OscType -> Evt a -> SE ()
```

The Osc-messages are coming from the event-stream. We send them to the machine with given host name (an empty string means the local machine). We also specify the OSC-address (it's a path-like string) and type of the messages.

Jack-instruments

With Jack-interface (native for Linux, also there are ports for OSX and PC) we can stream the output of one program to the input of another one. With Jack we can use our Csound instruments in DAW-software (like Ardour, Cubase, Ableton or BitWig).

We can create Jack-instrument if we set the proper options. We have to set the name of the instrument:

```
setJack :: String -> Options
setJack clientName
```

We have to set the proper rates (audio and control rates)

```
setRates :: Int -> Int -> Options
setRates sampleRate blockSize
```

Sample rate is a resolution of the output audio (typical values are 44100 or 48000). It should be the same as for the JACK. The block size is how many samples are in the control period. We have to process the control signals at the lower rate. The `blockSize` specifies the granularity of the control signals (typical values are 64, 128, 256).

We have to set the hardware and software buffers (It's `B` and `b` flags in the Csound):

```
setBufs :: Int -> Int -> Options
setBufs totalBufferSize  singlePeriodSize
```

To send or receive the values from the JACK Csound uses the buffer. We have to define the size of the whole buffer (the first argument) and the one period of the buffer (it should be integer multiplier of the blockSize).

To set all these properties we need to use the `Monoid` instance for `Options`. We need to append all the options:

```
> options = mconcat [ setJack "anInstrument", setRates 44800 64, setBufs 192 64 ]
> dacBy options asig
```

- <= Basics of sound synthesis
- => Scores
- Home

Scores

The type for Score comes from another library `temporal-media`. The score is a bunch of notes for an instrument to be played. Every note has a start time, duration (both in seconds) and some arguments for the instrument. The arguments can carry information about volume and pitch or some timbral parameters.

We can invoke an instrument with functions:

```
sco :: (Arg a, Sigs b) => (a -> SE b) -> Sco a -> Sco (Mix b)
mix :: Sigs a => Sco (Mix a) -> a
```

With `sco` we convert a score of notes to score of unmixed signals. With `mix` we can mix the score of signals to a single signal.

We can notice the two type classes.

- The `Arg` is for arguments. It contains the primitive types `D` (numbers), `Str` (strings) and `Tab` (tables). Also the argument can contain tuples of afore mentioned primitive types.
- The `Sigs` is for tuples of signals. It can be `Sig`, `Sig2`, `Sig4` and so on.

That's how we can play a single note for one second:

```
> let instr x = return $ osc $ sig x
> dac $ mix $ sco instr (temp 440)
```

The function `temp` creates a note that starts right away and lasts for one second. The argument of the function becomes the argument for the instrument to play.

Why do we need two functions? Isn't it better to convert the score of notes to signal? The answer to this question lies in the fact that when we have scores of signals we can combine them together. We can construct scores that contain signals from different instruments:

```
> let oscInstr x = return $ osc $ sig x
> let sawInstr x = return $ saw $ sig x
> dac $ mix $ mel [sco oscInstr (temp 440), rest 1, sco sawInstr (temp 440)]
```

We have created two instruments for pure sine and saw-tooth. Then we create a couple of notes (`temp`), apply the instruments to them (`sco`) and play them one after another (`mel`). We have put a one second rest between the notes. So the mix contains a signals from two different instruments.

Main functions

The main strength of the type `Sco` is that we can build complex scores out of simple primitives. Let's repeat our simple notes four times (`loopBy`) and play it four times faster (`str`):

```
> dac $ mix $ str 0.25 $ loopBy 4 $ mel [sco oscInstr (temp 440), rest 1, sco sawInstr (temp 440), rest 1]
```

Let's study the most important functions for composition (the complete list can be found in the docs for `temporal-media` package, on Hackage).

Primitive functions

Let's start with primitive functions:

```
temp :: a -> Sco a
rest :: D -> Sco a
```

The `temp` creates a single note that starts right away and lasts for one second. The function `rest` creates a pause that lasts for the given amount of time.

Functions for sequential and parallel composition

The next functions can group lists of scores. If we play notes one after another we can get a melody (`mel`). If we play notes at the same time we can get a harmony (`har`). So there are two functions:

```
mel, har :: [Sco a] -> Sco a
```

Let's play a major chord. First we play it in line and then we form a chord:

```
> let notes = fmap temp $ fmap (220 * ) [1, 5/4, 3/2, 2]
> let q = mel [mel notes, har notes]
> dac $ mix $ sco oscInstr q
```

We can hear the buzz in the last chord. It's caused by clipping. All signals for `dac` should have the amplitude less or equal than 1. We can scale the last chord by amplitude with the function `eff`:

```
eff :: (Sigs b, Sigs a) => (a -> SE b) -> Sco (Mix a) -> Sco (Mix b)
```

The `eff` applies an effect to the scores of signals.

```
dac $ mix $ mel [sco oscInstr (mel notes), eff (return . mul 0.2) $ sco oscInstr $ har notes]
```

The cool part of it is that we can treat a block of notes as a single value. We can give it a name, process it with a function or produce it with the function. It's impossible with plain Csound.

Time to delay

We can delay a bunch of notes with function:

```
del :: D -> Sco a -> Sco a
```

It takes a time to delay and a score. Let's play a note after two seconds delay:

```
> dac $ mix $ sco oscInstr $ del 2 $ temp 440
```

Speed up or slow down

We can speed up or slow down the notes playback with function `str` (short for stretch). It stretches the length of notes in time domain. Let's play our previous example four times faster:

```
> dac $ mix $ str 0.25 $ mel [sco oscInstr (mel notes), eff (return . mul 0.2) $ sco oscInstr $ har notes]
```

Loops

We can repeat a score several times with function `loopBy`:

```
loopBy :: Int -> Sco a -> Sco a
```

Functor

Needless to say that Sco is a functor. We can map the notes with `fmap`:

```
fmap :: (a -> b) -> Sco a -> Sco b
```

Twinkle twinkle little star

Let's create a simple melody and play it with a sine instrument. We are going to play a twinkle twinkle little star song. For this tune we have two kind of bars. The first bar contains two notes that are played twice. In the second type of bar one note is played twice and then another is held. We've got two patterns:

```
> let p1 a b = mel $ fmap temp [a, a, b, b]
> let p2 a b = mel [mel $ fmap temp [a, a], str 2 $ temp b]
```

Alsow we have a third pattern. It's more higher level. If we study the song we can see that we always play a first pattern and then we play a second one. So let's create a function for it:

```
> let p3 a b c d = mel [p1 a b, p2 c d]
```

Let's add an amplitude envelope to the instrument:

```
let oscInstr x = return $ mul (linsegr [0, 0.03, 1, 0.2, 0] 0.1 0) $ osc $ sig x
```

Let's also define a synonym for rendering function:

```
> let run = dac . mix . sco oscInstr . fmap cpspch
```

The `cpspch` is csound function that converts numeric values (encoded in Csound) to frequencies. The value `8.00` is a C1, the `8.01` is D#1 the value `8.02` is D1, the `9.00` is C2, and so on. The `8.12` is the same as `9.00`.

Let's listen for the first phrase:

```
> run $ str 0.25 $ p3 8.00 8.07 8.09 8.07
```

And then goes the second phrase:

```
> run $ str 0.25 $ mel [p3 8.00 8.07 8.09 8.07, p3 8.05 8.04 8.02 8.00]
```

We can notice that the third and fourt phrases are the same. And in the last two phrases we are going to repeat first two phrases. Let's give a name to phrases. And combine them in the tune:

```
> let ph1 = p3 8.00 8.07 8.09 8.07
> let ph2 = p3 8.05 8.04 8.02 8.00
> let ph3 = p3 8.07 8.05 8.04 8.02

> let ph12 = mel [ph1, ph2]
> let ph33 = loopBy 2 ph3
> let ph   = mel [ph12, ph33, ph12]

> run $ str 0.25 ph
```

With this approach we can better see the structure of the song.

Let's add chords to the tune. The song is based on three chords: C, F, G7. Let's create a function to play a chord:

```
> let ch a b c = mel [temp a, har [temp b, temp c]]
> let chC = ch 7.00 7.04 7.07
> let chF = ch 7.00 7.05 7.09
> let chG = ch 7.02 7.05 7.07
```

The structure of the chords is the same as the structure of the tune:

```
> let ch1 = mel [chC, chC, chF, chC]
> let ch2 = loopBy 2 $ mel [chG, chC]
> let ch3 = loopBy 2 $ mel [chC, chG]

> let ch12 = mel [ch1, ch2]
> let ch33 = mel [ch3, ch3]

> let ch = mel [ch12, ch33, ch12]
```

We can play the tune with chords. Let's play it twice:

```
> run $ str 0.25 $ loopBy 2 $ har [ch, ph]
```

Here it is! But what about clipping? Some signals are above the 1 in amplitude. We can easily solve this problem by scaling thae output signal. But here we are going to take another approach. We are going to introduce another parameter for the instrument. The instrument was defined for frequencies. Now it's going to get in the amplitudes also:

```
> let oscInstr (amp, cps) =
  return $ mul (sig amp * linsegr [0, 0.03, 1, 0.2, 0] 0.1 0) $ osc $ sig cps
```

We have to update the `run` function also:

```
> let run = dac . mix . sco oscInstr . fmap (\(a, b) -> (a, cpspch b))
```

We transform not the whole argument with `cpspch` but only the second value in the tuple. We have the scores of frequencies. Let's transform them in the scores of pairs! We assume that chords are quieter than the melody:

```
> run $ str 0.25 $ loopBy 2 $ har [fmap (\x -> (0.4, x)) ch,  fmap (\x -> (0.6, x)) ph]
```

Main classes for composition

I have simplified a bit the types for functions. For example, If we try to query the type in the ghci:

```
> :t mel
mel :: Compose a => [a] -> a
```

Or for `del`:

```
> :t del
del :: Delay a => DurOf a -> a -> a
```

The main functions belong to the type class. They are not defined for `Sco` lone. There is an implementation for `del`, `mel`, `har`, `rest`, etc. But later we are going to meet some other types which we can compose with the same functions. We are going to compose with samples (pieces of audio) and signal segments (signals that are limited with event streams).

The only functions that was defined on `Sco` is `temp`:

```
:t temp
temp :: Num t => a -> Track t a
```

We can see that is defined for `Track`s but the `Sco` is a special case for `Track`:

```
type Sco a = Track D a
```

Compose

Let's review the main classes. We can `Compose` things:

```
:i Compose
class Compose a where
  mel :: [a] -> a
  har :: [a] -> a
  (+:+) :: a -> a -> a
  (=:=) :: a -> a -> a
    -- Defined in â€˜Temporal.Classâ€™
```

We can see our good friends `mel` and `har` alongside with corresponding binary equivalents `(+:+)` and `(=:=)`.

There is a function that is based on this class:

```
> :t loopBy
loopBy :: Compose a => Int -> a -> a
```

It's defined as

```
loopBy n a = mel $ replicate n a
```

Delay

We can delay things:

```
> :i Delay
class Delay a where
  del :: DurOf a -> a -> a
    -- Defined in â€˜Temporal.Classâ€™
```

The `DurOf` is a type family. If you don't know what type family is here is the description. Type family is a function defined on types. It means that for any type that is instance of `DurOf` there is a corresponding type that signifies it's duration.

The duration for `Sco` is a constant number `D`.

So the function for delaying is:

```
del :: Delay a => DurOf a -> a -> a
```

Stretch

We can stretch things:

```
> :i Stretch
class Stretch a where
  str :: DurOf a -> a -> a
```

Rest

We can create pauses:

```
> :i Rest
class Compose a => Rest a where
  rest :: DurOf a -> a
    -- Defined in â€˜Temporal.Classâ€™
```

Loops

We can create an infinite loop:

```
class Loop a where
  loop :: a -> a
    -- Defined in â€˜Temporal.Classâ€™
```

Limit

We can limit the length:

```
:i Limit
class Limit a where
  lim :: DurOf a -> a -> a
```

This function is not defined for `Sco`.

- <= User interaction
- => Events
- Home

Events

Events

We have learned how to trigger an instrument with the score. Now we are going to learn how to do it with an event stream. The model for event streams is heavily inspired with functional reactive programming (FRP) though it's not a FRP model in the strict sense, because our signals are discrete and not continuous as FRP requires. But nevertheless it's useful to know the basics of FRP to learn the construction of event streams.

Introduction to FRP

FRP is a novel approach for description of interactive systems. It introduces two main concepts: behaviors and event streams. A behavior can be though as continuous signal of some value. It represents the changes in the life of the value. What's interesting is that it describes the whole life of the value. An event stream contains a value that may happen sometimes. For example if we have a computer mouse. The position of the cursor is a behavior that contains two values (X and Y) and an event stream is a stream of all clicks for the mouse's buttons.

In the traditional callback based approach we have some instrument to register a callback function for the mouse clicks. The function accepts an event that carries the information about which button was pressed and what is the position of the mouse. When something happens we can update some mutable variables.

With FRP we can manipulate event streams as if they are values. We can map over the values that are contained in the events. We can merge two event streams together. We can accumulate some value based on upcoming events. And we can convert the event streams to behaviors. The simplest function that comes into mind is creation of step-wise constant function. When something happens on the event stream we hold the value until the next event fires and updates the value.

```
stepper :: a -> Event a -> Behavior a
stepper initVal events
```

We have an initial value. It lasts while nothing has happened.

More complicated function is a switch function:

```
switch :: Behavior b -> (a -> Behavior b) -> Event a -> Behavior b
switch initVal behaviorProducer events
```

The switch applies some behavior constructor to the value of event when something happens. The resulting behavior lasts until the next event happens. Then we apply the function again and so on.

With this approach we can build complex behaviors from simple ones. The key feature is that a single value can contain a whole event stream! It removes the need for mutable variables. we use mutable values with callbacks when we want to communicate the changes of the value from one callback to another. If we want to use the results of a callback in the rest of the program.

That's how we can count the clicks of the mouse:

```
> showOnScreen $ stepper 0 $ accum 0 (+ 1) $ filter isLeftClick $ mouseClicks
```

It's an imaginary code but it shows the idea. The ides is that we can take the stream of all mouse clicks. Then we can filter it so that we get only clicks for the left button. Then we can accumulate a value over the event stream and in the last function we convert the stream of counter into the continuous signal and show it on the screen.

The callback based solution can look like this (again it's an imaginary imperative code written in Haskell):

```
counter <- newIORef 0
screen <- newScreen

Mouse.registerCallback $ \evt -> do
    if isLeftClick evt then do
        modifyIORef (+1) counter
        pushValuetoScreen screen =<< readIORef counter
    else do
        return ()
```

Triggering instruments with event streams

Let's trigger an instrument with event stream. There is a function:

```
sched :: (Arg a, Sigs b) => (a -> SE b) -> Evt (Sco a) -> b
```

It takes in an instrument and an event stream of scores. Every event contains a score. We have a simple instrument:

```
> let bam _ = mul (fades 0.01 0.3) $ pink
```

It plays a pink noise. It takes no arguments but the `sched` function requires an instrument to be a function so we created an "empty" argument. Let's trigger it with the stream:

```
> dac $ sched bam $ withDur 0.1 $ metro 2
```

The `metro` creates an event stream of ticks that happen with given frequency. We have set the frequency to 2 per second. The function `withDur` creates an event stream of scores out of event stream of values. We can set the duration of every event. The final function `sched` applies an instrument to an event stream. We get the signal as a result.

Let's create an instrument with a parameter. We are going to produce a filtered pink noise:

```
> let bam x = mul (fades 0.01 0.3) $ at (mlp (2500 * sig x) 0.1) $ pink
```

The parameter is responsible for the center frequency. The example introduces an instrument that is not parametrized with an amplitude or frequency but still it can produce a musical result. Let's create a sound:

```
> dac $ sched bam $ withDur 0.1 $ cycleE [1, 0.5, 0.5, 0.25, 1, 0.5, 0.8, 0.65] $ metro 4
```

The function `cycleE` substitutes a values of the event stream with repeating values that are taken from the given list. When something happens it takes a next value from the list and puts it to the event stream when it reaches the last value in the list it starts from the first value and so on. With the example we create a drum pattern.

Also we can create an arpeggio:

```
> let instr x = return $ mul (fades 0.01 0.1) $ tri $ sig x
> let notes = fmap (* 220) [1, 5/4, 1, 3/2, 5/4, 2, 3/2, 10/4, 2, 3, 10/4, 4]
> dac $ mul 0.5 $ sched instr $ withDur 0.1 $ cycleE notes $ metro 8
```

Let's add a couple effects. We add a delay (`echo`) and low pass filter (`mlp`):

```
> dac $ mul 0.25 $ at (mlp 3500 0.1) $ echo 0.25 0.5
    $ sched instr $ withDur 0.1 $ cycleE notes $ metro 8
```

We can recieve the events from the user. Let's create a button:

```
> let btn = button "play"
```

The button produces an event stream of clicks:

```
> :t btn
btn :: Source (Evt Unit)
```

The `Unit` is Csound value that signifies no value or empty tuple. It has to be defined for implementation reasons. We can not just use Haskell empty tuple.

Let's trigger an instrument:

```
> dac $ lift1 (sched instr . withDur 0.1 . fmap (const 440)) btn
```

The fun part of it is that an instrument can contain signals that were created with event streams! Let's abstract away our arpeggios in an instrument:

```
> let arpInstr _ = mul (fadeOut 1) $ at (mlp 3500 0.1) $ echo 0.25 0.5 $ mul 0.25
    $ sched instr $ withDur 0.1 $ cycleE notes $ metro 8
> dac $ lift1 (sched arpInstr . withDur 1) btn
```

There are functions that play an instrument until something happens with another event stream:

```
schedUntil :: (Arg a, Sigs b) => (a -> SE b) -> Evt a -> Evt c -> b
```

Let's create another button for stopping an instrument. We are going to play the `arpInstr` until we press another button.

```
> let stop = button "stop"
> dac $ hlift2 (schedUntil arpInstr) btn stop
```

We can create an event stream of keyboard presses. There are handy functions:

```
charOn, charOff :: Char -> Evt Unit
```

The function takes in a symbolic representation of key and produces an event stream of clicks/ Let's rewrite previous example:

```
> dac $ (schedUntil arpInstr) (charOn 'a') (charOff 'a')
```

Try to press the key `a`. We should focus on the Csounds window.

There is a more generic function `keyIn`:

```
> :t keyIn
keyIn :: KeyEvt -> Evt Unit
> :i KeyEvt
data KeyEvt = Press Key | Release Key
```

And type `Key` contains all special keys. We can find the complete description in the documentation.

There are functions to listen for midi event streams:

```
midiKeyOn, midiKeyOff :: MidiChn -> D -> SE (Evt D)
> :i MidiChn
data MidiChn = ChnAll | Chn Int | Pgm (Maybe Int) Int
```

We are going to study them later.

Main functions for event streams

Let's study the main functions for construction of event streams.

Monoid

Event stream is a `Monoid`. The `mempty` is an event stream that has no events and `mappend` combines to event streams into a single event stream that contains events from both streams. Reminder: `mconcat` is a version of `mappend` that is defined on lists.

We can create an intricate drum pattern:

```
> let bam _ = mul (fades 0.01 0.05) $ pink
> dac $ sched bam $ withDur 0.1 $ mconcat [metro 2, metro 1.5, metro $ 3/7]
```

Try to exclude values from the list or include your own and see what happens.

Functor

an event stream is a functor. We can transform the events of an event stream with a function. We can map over events with `fmap`:

```
fmap :: (a -> b) -> Evt a -> Evt b
```

The function `withDur` that turns values to scores is defined with `fmap`:

```
withDur :: D -> Evt a -> Evt (Sco a)
withDur dur = fmap (str dt . temp)
```

There is another useful function `devt`. It substitutes any value in the stream with the given value:

```
devt :: a -> Evt b -> Evt a
devt a = fmap (const a)
```

We can create pitched beats:

```
> let oscInstr x = return $ mul (fades 0.01 0.1) $ osc $ sig x
> dac $ sched oscInstr $ withDur 0.1 $ mconcat
    [devt 440 $ metro 2, devt 660 $ metro 1.5, devt 220 $ metro 0.5]
```

Picking values from the lists

We already familiar with th function `cycleE` it cycles over the values in the list. Another useful function is `oneOf` it picks a value at random from the list:

```
> dac $ mlp 2500 0.1 $ sched oscInstr $ withDur 0.1 $
    oneOf (fmap (* 220) [1, 9/8, 5/4, 3/2, 2]) $ metro 8
```

The type signatures:

```
cycleE, oneOf :: [a] -> Evt b -> Evt a
```

We can also set the frequencies of repetition for the values in the list:

```
type Rnds a = [(D, a)]
```

```
freqOf :: (Tuple a, Arg a) => Rnds a -> Evt b -> Evt a
```

The type `Rnds` is a list of pairs. They are values augmented with probabilities. The sum of probabilities should be equal to 1.

The most generic function is:

```
listAt :: (Tuple a, Arg a) => [a] -> Evt D -> Evt a
```

It picks values from the list by the event stream of indices.

Accumulation of values

We can create a simple accumulation of values.

The simple function `iterateE` applies a function over and over when something happens on the event stream:

```
iterateE :: Tuple a => a -> (a -> a) -> Evt b -> Evt a
```

Let's listen to the midi notes:

```
> dac $ sched oscInstr $ withDur 0.2 $ fmap cpsmidinn $ iterateE 30 (+1) $ metro 4
```

The function `cpsmidinn` trn an integer number of midi key to frequency.

The function `iterateE` doesn't take into account the value of events. We can run counter that takes values from the event stream:

```
appendE :: Tuple a => a -> (a -> a -> a) -> Evt a -> Evt a
```

The function `appendE` takes in an initial value and a function to apply to the current value and the value of the event. When event happens the function is applied and result is stored as the state. The current value is put into the output stream. We can create a simple synth with two buttons. Left button is for going down the scale and the right button is for going up the scale:

```
> let btnDown = button "down"
> let btnUp   = button "up"
> dac $ hlift2 (\down up -> mlp 1500 0.1 $ saw $ cpsmidinn $ evtToSig 60
    $ appendE 60 (+) $ mconcat [devt 1 up, devt (-1) down])
    btnDown btnUp
```

It's interesting to note how an instrument is controlled with an event stream. We don't trigger any instrument. We convert the event stream to signal. The signal controls the pitch of the filtered saw.

The function `evtToSig` converts an event stream of numbers to a signal:

```
evtToSig :: D -> Evt D -> Sig
evtToSig initVal evt
```

Let's unwind this expressin. First we transform the event streams for buttons so that each button produces 1's or -1's and we merge two streams in the single stream:

```
mconcat [devt 1 up, devt (-1) down]
```

Then we create a running sum. So that when user presses up the value goes up and when the user presses down we subtract the 1.

```
appendE 60 (+) $ previousExpression
```

Then we convert event stream to signal and convert numbers to pitches:

```
cpsmidinn $ evtToSig 0 $ previousExpression
```

At the last expression we apply the pitch to filtered saw and send the output to speakers:

```
mlp 1500 0.1 $ saw $ previousExpression
```

The whole expression is wrapped in the `hlift2` so that we can read the values from UI-widgets and stack the widgets horizontally.

There are more generic functions for accumulating state:

```
accumE  :: Tuple s => s -> (a -> s ->    (b, s)) -> Evt a -> Evt b
accumSE :: Tuple s => s -> (a -> s -> SE (b, s)) -> Evt a -> Evt b
```

They accumulate state in pure expressions and on expressions with side effects.

Filtering event streams

We can skip some events if we don't like them. We can do it with function:

```
filterE :: (a -> BoolD) -> Evt a -> Evt a
```

The first argument is a predicate, if it's true for the given event it is put in the output otherwise it's left out.

We can also skip events at random:

```
randSkip :: D -> Evt a -> Evt a
```

The first argument is the probability of skip.

There are many more functions we can check them out in the docs (see module `Csound.Control.Evt`).

Signal segments

The signal segments lets us schedule signals with event streams. They are defined in the module `Csound.Air.Seg`. A signal segment can be constructed from a single signal or a tuple of signals:

```
toSeg :: a -> Seg a
```

It plays the signal indefinitely. We can limit the duration of the segment with static length measured in seconds:

```
constLim :: D -> Seg a -> Seg a
```

or with an event stream:

```
type Tick = Evt Unit
```

```
slim :: Tick -> Seg a -> Seg a
```

The signal is played until something happens on the given event stream. When segment is limited we can loop over it:

```
sloop :: Seg a -> Seg a
```

It plays the segment and the replays it again when it comes to an end.

If we several limited signals we can play them in sequence:

```
sflow :: [Seg a] -> Seg a
```

When the first signal stops the next one comes into play and when it stops the next one is turned on.

Also we can play segments at the same time:

```
spar :: [Seg a] -> Seg a
```

The length of the result equals to the longest length among all input segments.

We can delay the segment with an event stream or a static length:

```
sdel     :: Tick -> Seg a -> Seg a
constDel :: D    -> Seg a -> Seg a
```

There is a handy shortcut for playing nothing for the given amount of time:

```
srest     :: Num a => Tick -> Seg a
constRest :: Num a => D    -> Seg a
```

To listen the segment we need to convert it to signal:

```
runSeg :: Sigs a => Seg a -> a
```

That's it. With signal segments we can easily schedule the signals with event streams.

Let's create a button and turn the signal on when it's pressed:

```
> dac $ lift1 (\x -> runSeg $ sdel x $ toSeg $ osc 440) (button "start")
```

Let's create a second button that can turn off the signal.

```
> dac $ hlift2 (\x y -> runSeg $ sdel x $ slim y $ toSeg $ osc 440)
    (button "start")
    (button "stop")
```

When signal stops the program exits. We can repeat the process by looping:

```
> dac $ hlift2 (\x y -> runSeg $ sloop $ sdel x $ slim y $ toSeg $ osc 440)
    (button "start")
    (button "stop")
```

Let's play several signals one after another with `sflow`:

```
> dac $ hlift2 (\x y -> runSeg $ sloop $ slim y
    $ sdel x $ sloop $ sflow $ fmap (slim x . toSeg . osc) [220, 330, 440])
    (button "start")
    (button "stop")
```

Warning: Note that signal release is not working with signal segments.

Samplers

There are handy functions to trigger signals that are based on signal segments. We can look at the module `Csound.Air.Sampler` to find them.

The functions trigger the signals with event streams, keyboard presses and midi messages. Let's look at the functions for keyboard (the rest functions are roughly the same).

There are several patterns of (re)triggering.

- `Trig` -- triggers a note and plays it while the same key is not pressed again

  ```
  charTrig :: Sigs a => String -> String -> a -> SE a
  charTrig ons offs asig = ...
  ```

  It accepts a string of keys to turn on the signal and the string of keys to turn it off.

  Let's try it out:

  ```
  > dac $ at (mlp 500 0.1) $ charTrig "q" "a" $ saw 110
  ```

  Try to hit `q` and `a` keys.

- `Tap` -- is usefull optimization for `Trig` it plays the note only for a given static amount of time (it's good for short drum sounds) `Tap` has the same arguments but the turn off string is substituted with a note's length in seconds (it comes first):

  ```
  charTap :: Sigs a => D -> String -> a -> SE a
  ```

- `Push` -- plays a signal while the key is pressed.

  ```
  charPush :: Sigs a => Char -> a -> SE a
  ```

  Let's create a simple note:

  ```
  > dac $ at (mlp 500 0.1) $ charPush 'q' $ saw 110
  ```

  Let's create a couple of notes:

  ```
  > dac $ at (mlp 500 0.1) $ sum [charPush 'q' $ saw 110, charPush 'w' $ saw (110 * 9 / 8)]
  ```

  Note that only one key (de)press can be registered at the moment. It's current limitation of the library. It's not so for midi events.

- `Toggle` -- uses the same key to turn the signal on/off.

  ```
  > dac $ at (mlp 500 0.1) $ charPush 'q' $ saw 110
  ```

- `Group` -- creates a mini mono-synth. It's give a list of pairs of keys an signals. When key is pressed the corresponding signal starts playing. When the next key is pressed the previous is turned off and the current is turned on.

```
charGroup :: Sigs a => [(Char, a)] -> SE a
```

There are many more functions. You can find them in the module `Csound.Air.Sampler`.

Turning keyboard to DJ-console

Let's create a mini mix board for a DJ. The first thing we need is a cool dance drone:

```
> let snd1 a b = mul 1.5 $ mlp (400 + 500 * uosc 0.25) 0.1 $ mul (sqrSeq [1, 0.5, 0.5, 1, 0.5, 0.5, 1, 0.5] b) $ saw a
```

Let's trigger it with keyboard!

```
> dac $ charTrig "q" "a" (snd1 110 8)
```

Try to press `q` and `a` keys to get the beat going. Let's create another signal. It's intended to be high pitched pulses.

```
> let snd2 a b = mul 0.75 $ mul (usqr (b / 4) * sqrSeq [1, 0.5] b) $ osc a
```

Let's try it out. Try to press `w`, `e`, `r` keys.

```
> dac $ mul 0.5 $ sum [charPush 'w' $ snd2 440 4, charPush 'e' $ snd2 330 4, charPush 'r' $ snd2 660 8]
```

Note that only one keyboard event can be recognized. So if you press or depress several keys only one is going to take effect. It's a limitation of current implementation. It's not so with midi events. Let's join the results:

```
> let pulses = mul 0.5 $ sum [charPush 'w' $ snd2 440 4, charPush 'e' $ snd2 330 4, charPush 'r' $ snd2 660 8]
> let beat = mul 0.5 $ sum [charTrig "q" "a" (snd1 110 8), charTrig "t" "g" $ snd1 220 4]
```

Let's create some drum sounds:

```
> let snd3 = osc (110 * linseg [1, 0.2, 0])
> let snd4 = mul 3 $ hp 300 10 $ osc (110 * linseg [1, 0.2, 0])
> let drums = sum [charTrig "z" "" snd3, charTrig "x" "" snd4]
```

Let's rave along.

```
> dac $ sum [pulses, mul 0.5 beat, mul 1.2 drums]
```

- <= Scores
- => Real-world instruments show case

- Home

Batteries included

There are plaenty of beatiful instruments designed for the library! They can be found at the package `csound-catalog` (available on Hackage).

The main type for the instrument is a `Patch`. The patch is an instrument function to be played and the chain of effects:

```
type CsdNote a = (a, a)
type Instr a b = CsdNote a -> SE b

type Fx a = a  -> SE a
type DryWetRatio = Sig

data FxSpec a = FxSpec
    { fxMix :: DryWetRatio
    , fxFun :: Fx a
    }

data Patch a b = Patch
    { patchInstr :: Instr a b
    , patchFx    :: [FxSpec b]
    }
```

An instrument is a function that takes in a pair of values. The pair contains amplitude and frequency in Hertz. The instrument produces a signal or a tuple of signals wrapped in the `SE`-monad.

An effect is a function that transforms incoming signals and dry/wet ratio. Dry/wet ratio defines how much of the original signal is going to be processed. The `0 is for "dry" unprocessed signal. The1`` is for fully processed signal.

Effects are applied as in math application from last item in the list to the first.

Let's define a simple patch:

```
> let instr (amp, cps) = mul (0.45 * sig amp * fades 0.01 0.1) $
    fmap (mlp (2500 * leg 0.1 0.75 0.65 0.2) 0.1) $
    rndTri (sig cps) + rndTri (2.01 * sig cps)
```

The instrument is a couple of triangular waves with random phases. They are processed with moog low-pass filter, then the siimple envelope (fade in and out) is applied.

The effect is just a bit of reverb:

```
> let fxs = [FxSpec 0.25 (return . smallHall2)]
```

Let's create a patch:

```
> let p = Patch (fmap fromMono . instr) fxs
```

There are plenty usefull functions defined for patches. Let's play the patch with midi keyboard:

```
> vdac $ atMidi p
```

The `atMidi` function takes in a patch and applies all necessary functions to make a midi-instrument. We can play a single held note:

```
> dac $ atNote (0.75, 220)
```

We can play the patch with event stream:

```
> let notes = cycleE [(1, 220), (0.5, 220 * 5/ 4), (0.25, 330), (0.5, 440)]
> dac $ atSched p (withDur 0.15 $ notes $ metro 6)
```

We can also play Scores with patch:

```
> let ns = fmap temp [(0.5, 220), (0.75, 330), (1, 440)]
> let notes = str 0.25 $ mel [mel ns, har ns, rest 4]
> dac $ mul 0.75 $ mix $ atSco p notes
```

We can change the amount of dry/wet apllied to the signal.

```
atMix   :: Sig   -> Patch a -> Patch a
atMixes :: [Sig] -> Patch a -> Patch a
```

The `atMix` changes the ratio only for the last effect. With `atMixes` we can tune all ratios. Also there is a function to play unprocessed dry signal:

```
dryPatch :: Patch a b -> Patch a b
```

We can add effects to the given patch. We can insert the effect in the beginning and to the end of the chain:

```
addPreFx, addPostFx :: DryWetRatio -> Fx b -> Patch a b -> Patch a b
```

Let's add a delay effect to the defined patch:

```
vdac $ atMidi $ addPreFx 1 (at $ echo 0.35 0.65) p
```

There are some usefull functions for Pads:

```
deepPad :: (Fractional a, Sigs b, SigSpace b) => Patch a b -> Patch a b
```

It adds some weight to the timbre. It's defined with more generic function:

```
harmonPatch :: (Fractional a, Sigs b, SigSpace b) => [Sig] -> [a] -> Patch a b -> Patch a b
harmonPatch amplitudes harmonics patch
```

It plays a given patch as harmonic series.

But let's study some predefined patches. We should install the `csound-catalog` package. Then we need to import the module `Csound.Patch` and try some goodies (you can use `dac` instead of `vdac` if you have a real midi device):

```
>:m +Csound.Patch
> vdac $ atMidi vibraphone1

> vdac $ atMidi dreamPad

> vdac $ atMidi $ deepPad razorPad

> vdac $ atMidi epianoBright

> vdac $ atMidi xylophone

> vdac $ atMidi scrapeDahina

> vdac $ atMidi noiz

> vdac $ atMidi mildWind

> vdac $ atMidi toneWheelOrgan

> vdac $ atMidi $ addPreFx 1 (at $ echo 0.35 0.65) banyan

> vdac $ atMidi caveOvertonePad

> vdac $ atMidi flute

> vdac $ atMidi hulusiVibrato

> vdac $ atMidi shortBassClarinet

> vdac $ atMidi $ withDeepBass 0.75 pwBass

> vdac $ atMidi pwEnsemble

> vdac $ atMidi albertClockBellBelfast
```

There are 200+ of other instruments to try out! You can find the complete list in the module `Csound.Patch`.

- <= Events
- => Sound fonts
- Home

SoundFonts

With soundfonts it's very easy to turn your computer in synthesizer. A sound font encodes the timbre of the instrument with samples. There are many free soundfonts with good level of quality.

Midi

We can read soundfonts that are encoded in sf2 format. The function `sfMsg` can turn sound font file in midi instrument:

```
> let sf = Sf "rhodes.sf2" 0 0
> vdac $ midi $ sfMsg sf 0.5
```

We play the file `"rhodes.sf2"` at the bank `0` with program `0`. The sound font can contain many instruments. They are identified with the pair of integers: bank and program number. The second argument of the function `sfMsg` is sustain value of the instrument in seconds.

The funciton `sfMsg` reads the samples with linear interpolation. We can improve the quality with cubic interpolation if we use the function `sfMsg3`.

Non-midi

We are not constrained to midi-frequencies. We can read samples at any frequency with function `sfCps`:

```
> let sf = Sf "rhodes.sf2" 0 0
> dac $ sfCps sf 0 0.5 440
```

The arguments are:

- a soundfont preset.
- sustain.
- the amplitude (it ranges in the iterval [0, 1])
- the frequency in Hz.

We can find a lot of soundfonts in the net. Some links to start:

- Hammer sound

- Rhodes
- And many more

- <= Real-world instruments show case
- => Custom temperament. Microtonal music
- Home

Custom temperament

The temperament in music is the set of exact values in Hz that we assign to notes. In modern western music tradition the main temperament is equal temperament.

Equal temperament divides octave on 12 equal parts (in logarithmic scale) so that each interval with the same number of notes in between is should sound the same no matter from where you place the root of the interval. For instance a C major triad should sound the same as F# major triad. The sound is different in pitch but not in quality or relationships of the notes. It gives a huge advantage for transposition. If you want to sing along with the song but the scale is not good for your voice you can easily transpose the scale and it should sound the same.

But it brings some disadvantages too. The main strength of the equal temperament can become it's main weakness. All major thirds are the same and all minor seconds are the same. In fact all same intervals produce the same sound in all scales. It can wipe away all the colors from the music. The Bach, Chopin, Beethoven and all composers from the Romanticism era used different temperaments. So when we listen Chopin on the modern piano we listen to the music that is not quite the same as Chopin intended it to be.

They used temperaments that have many slightly different triads. It gives the specific colors to the scales and it makes the scale divergence within the composition more profound. Change in scale is not just a trasnposition it can affect the mood of the piece.

Ethnic music enjoys the variety of temperaments. In the Indian classical music octave is divided in 22 notes (or shruties). The musician picks up 5 to 9 notes from the raw material of 22 shruties and each combination can create different mood. For Indian music different scales have not only different sharps and flats but the quality of the note's flatness can be different from scale to scale. For example there can be three different F#.

By default all midi playing utilities use equal temperament. But we can alter this behavior.

Patches

The most common way to play patches is to use the function `atMidi`. It plays the patch with equal temperament. If you have a real midi device you can use the `dac` in place of `vdac`:

```
> ghci
> :m +Csound.Base Csound.Patch
> vdac $ atMidi vibraphone1
```

To change the temperament we can use the function `atMidiTemp` that accepts the temperament as the first argument:

```
> vdac $ atMidiTemp young1 vibraphone1
```

We can try out an ancient Pythagorean tuning:

```
> vdac $ atMidiTemp pythagor1 vibraphone1
```

We have several predefined temperaments to try out: `equal1`, `pythagor`, `meantone`, `just1`, `werckmeister`, `young1`, `young2`, `young3`.

Temperament

Temperament is defined with the base note and the set of relationships for the notes of the scale. The temperament (`Temp`) can be created with function `genTemp`:

```
genTemp :: Double -> Double -> Double -> [Double] -> Temp
genTemp mainInterval baseHz baseMidiKey cents
```

Let's look at the arguments:

- `mainInterval` - The frequency range covered before repeating the grade ratios, for example 2 for one octave, 1.5 for a fifth etcetera.
- `baseHz` - The base frequency of the scale in cycles per second.
- `baseMidiKey` - The integer index of the scale to which to assign `baseHz` unmodified.
- `cents` - the list of ratios for each note of the temperament in cents.

So here is the definition for equal temperament:

```
equal1  = genTemp 2 261.63 60 equalCents1
equalCents1 = [0, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200]
```

The list should include the first note from the next octave (scale's main interval).

There are utility functions that simplify the definition of the temperament:

```
baseC :: [Double] -> Temp
baseC cents
```

The `baseC` creates a temperament with octave interval and modern C as the base note of the temperament. We can rewrite the previous definition as:

```
equal1 = baseC equalCents1
```

There are other useful functions

```
stdTemp, barTemp :: [Double] -> Temp
```

The function `stdTemp` creates a scale so that 9nth note is modern concert A (440 Hz). The `barTemp` creates a temperament with baroque concert A (415 Hz). There are predefined lists of cents for several western temperaments: `equalCents1`, `pythagorCents1`, `meantoneCents`, `werckmeisterCents`, `youngCents1`, `youngCents2`, `youngCents3`. We can use them as an example to define our own temperaments.

Midi instruments

Let's invoke a simple virtual midi instrument:

```
> vdac $ midi $ onMsg $ \cps -> 0.5 * fades 0.01 0.1 * tri cps
```

The `onMsg` function takes in a function of type `Sig -> Sig` and converts it to midi function of the type `Msg -> SE Sig` We can change the temperament with function `onMsg'`

```
> vdac $ midi $ onMsg' just1 $ \cps -> 0.4 * fades 0.01 0.1 * tri cps
```

The `onMsg` takes in a temperament as the first argument. Behind the scenes the function `onMsg` invokes the function `ampCps`. it extracts the amplitude and frequency from the midi message. To change the temperament we can use the the function `ampCps'`. it accepts the temperament as the first argument:

```
ampCps' :: Temp -> Msg -> (D, D)
ampCps temp msg = (amplitude, frequency)
```

The `ampCps'` uses the function `cpsmidi'` to extract frequency with custom temperament:

```
cpsmidi' :: Temp -> Msg -> D
```

Patches

With patches we can use the functions `atMidiTemp` (for polyphonic synths) and `atMonoTemp` (for monophonic synths). Let's lookt at a couple of examples:

```
> vdac $ atMidiTemp young1 dreamPad
```

```
> vdac $ atMonoTemp just1 nightPadm
```

Sound fonts

Also we can use custom temperaments with sound fonts.

```
> vdac $ sfTemp meantone (Sf "/path/to/soundfont/jRhodes3.sf2" 0 0) 0.2
```

Temperament as a note's parameter

It's worth to note that we can pass the temperament as the instrument's argument. It can be used inside the scores or with event streams. The `Temp` type is an instance of the typeclass `Arg`.

More information on the datatype `Temp` and it's functions you can find in the module `Csound.Tuning`.

- <= Custom temperament. Microtonal music
- => Samples
- Home

csound-sampler

A csound-sampler is an easy to use sampler based on csound-expression library.

We can load and play audio files. We can play them back in loops, in reverse, at random segments with different pitch, apply any effects available in Csound. We can arrange them in sequences. We can easily create patterns of audio snippets.

How to install

The library is available on Hackage. So we can install it with cabal-install:

```
> cabal update
> cabal install csound-sampler
```

Also we need to install the Csound compiler. It's software synth. It's going to be our audio engine. When it's properly installed it should be possible to run the csound at the command line. Open up your terminal and type in `csound`. On windows sometimes csound complains on missing `python27.dll`. If it has happened with you download the dll from the web and drop it in the folder `C:\Windows\system32`.

Let's review the main functions of the library.

How to load files and sounds

Let me introduce you to `Sam` (Sam nods). He is the main guy in the library. He can sing samples for you. All samples are in stereo.

We can listen to the audio file:

```
module Main where

import Csound.Base
import Csound.Sam

audio = wav "samples/song.wav"

bpm = 120 * 4

main = dac $ runSam bpm audio
```

Let's load the module in the ghci and invoke the main function.

```
ghci Main
> main
... and the Sam sings ...
```

Press `Ctrl+C` to stop the program. Note that it's the best way to work with csound libs. We can create module with common functions and imports then we load it in the ghci and we can start messing around with samples right in the interpreter!

We load the lossless audio files with function `wav`. If our audio file is mono we should use the function `wav1`.

The function `wav` creates a sample out of file name:

```
wav :: String -> Sam
```

To hear the sample we should run the `Sam`.

```
runSam :: D -> Sam -> SE (Sig, Sig)
runSam bpm sample = ...
```

The first argument of the `runSam` is the Beats Per Minute measure. It's the tempo of the sample playback. When the sample is converted to stereo signal we can hear the result with function `dac`. It's a standard function form csound-expression library.

Playing loops

Ok, we can hear a sample. How can we loop it? There is a function

```
loop :: Sam -> Sam
```

To make things more easy let's create a couple of shortcuts:

```
module Main where

import Csound.Base
import Csound.Sam

run = dac . runSam (120 * 4)

song = wav "samples/song.wav"
beat = wav "samples/beat.wav"
```

Let's save it as Main.hs. From nowon we are going to load the module `Main.hs` in the interpreter. So let's loop over beat:

```
ghci Main.hs
> run $ loop beat
```

Let's add the voice on top of it:

```
> run $ loop beat + song
```

The `Sam` behaves just like a simple number. We can add samples or take a mean of samples:

```
> run $ mean [loop beat, song]
```

Changing the volume of the sample

But the beat is too loud we can not hear the voice properly. Let's fix that:

```
> run $ mean [mul 0.5 $ loop beat, song]
```

The function `mul` comes with library `csound-expression`. It happens that the `Sam` is the instance of `SigSpace`: We can use the the function `mapSig` to apply any signal transforms to it:

```
mapSig :: (SigSapce a) => (Sig -> Sig) -> a -> a
```

The `mul` is just a multiplication by a signal.

```
mul :: (SigSpace a) => Sig -> a -> a
mul k = mapSig (* k)
```

Playing parts of the sample

What if we don't want to hear the whole song but only 8 beats of it. We can use `lim`:

```
> run $ mean [mul 0.5 $ loop beat, lim 8 song]
```

Applying envelopes

We can hear only a part of the song. But now we can hear a nasty clipping. The sound jumps at the and of the sample.

We can fix it with envelope:

```
> run $ mean [mul 0.5 $ loop beat, linEnv 1 1 $ lim 8 song]
```

The `linEnv` takes rise and decay times in BPM and applies a trapezoid envelope to the sample.

There are many more envelopes to explore:

```
-- | Exponential trapezoid
expEnv :: D -> D -> Sam -> Sam
expEnv rise dec = ...

-- | Parabolic envelope
hatEnv :: Sam -> Sam

-- | Linear rise and decay envelopes
riseEnv, decEnv :: Sam -> Sam

-- | Exponential rise and decay envelopes
eriseEnv, edecEnv :: Sam -> Sam
```

Playing samples in reverse

It's cool to reverse audio. The sound becomes mystic and SigurRos'y. We can play audio files in reverse:

```
> let revSong = wavr "samples/song.wav"
run revSong
```

Notice the suffix `r` in the function `wavr`.

Playing one sample after another

Let's play song in two modes. the first time forth and then backwards:

```
> let songLoop = let env = (linEnv 1 1 . lim 8) in loop $ flow [env song, env revSong]
> let beatLoop = mul 0.5 $ loop beat
> run $ mean [beatLoop, songLoop]
```

Notice the function `flow`. It plays a list of samples in sequence. If we want to put some silence between the song samples, we can use the function `rest`. It creates a silent sample:

```
> flow [env song, rest 4, env revSong]
```

Delaying the samples

We want beat's to enter the song first and then after 4 beats delay comes the voice:

```
> run $ mean [beatLoop, del 4 songLoop]
```

We can use the function `del`.

Playing samples at random

What if we want to make our voice track more alive. We can introduce randomness in the choice of the sample:

```
pick :: Sig -> [Sam] -> Sam
pick period sample =
```

The function `pick` plays one sample from the list with the given period:

```
> pick 8 [env song, rest 8, env revSong]
```

That's how we can play song back and forth with random playback modes. There is a function

```
pickBy :: Sig -> [(D, Sam)] -> Sam
```

The `pickBy` plays samples with given frequencies of occurrence. The sum of the frequencies should be equal to 1.

Playing patterns samples

We can play sample in the loop. But what's about more complex patterns? we can create them with function `pat` (short for pattern):

```
pat :: [D] -> Sam -> Sam
```

The first argument is the list of time length for sequence of loops. It's the drum-like pattern:

```
> pat [3, 3, 2] beat
```

It means play the sample `beat` in the loop. The loop spans for 8 beats and it contains three segments. The length of each segments is given in the list.

The pat plays the whole sample. When samples overlap it mixes them together. If we want to play just a parts of the sample we can use the function `rep` (short for repeat). With it we can create complex drum patterns out of simple samples:

```
> rep [3, 3, 1, 2, 1] beat
```

Changing the tempo

We can speed up or slow down the sample playback with

```
str, wide :: D -> Sam -> Sam

str  speedUpRate  = ...
wide slowDownRate = ...
```

It doesn't changes the rate of playback. It changes the BPM measure. The looping or limiting functions will respond to the changes.

Changing the pitch and panning

We can change the pitch of the sample with function:

```
atPch :: D -> Sam -> Sam
```

It lowers (if negative) or heightens the pitch in semitones:

```
> let songLoop = atPch 2 $ loop song
```

We can change the pan with the function

```
atPan :: Sig -> Sam -> Sam
```

The first argument is the panning level. The zero is all left and the one is all right. We can easily create the spinning pan:

```
> let songLoop = atPan (uosc 0.1) $ loop song
```

Playing segments of the audio file

What if we like just one specific spot in the audio file and we want to loop only over it. we can read the segment with function:

```
seg :: D -> D -> String -> Sam
seg startTime endTime fileName = sample
```

The times are measured in seconds. To play the segment in reverse we can use the function `segr`. There are mono variants: `seg1` and `segr1`.

Playing random segments of the audio file

We can create complex sound out of the simple one if we play segments of it at random:

```
rndSeg :: D -> D -> D -> String -> Sam
rndSeg segLength startTime endTime fileName = sample
```

The first argument is the length of the segment. If we want to read segments from the entire audio file we can use the function:

```
rndWav :: D -> String -> Sam
rndWav segLength fileName = sample
```

Applying effects

The type `Sam` is a synonym for generic type:

```
type Sig2 = (Sig, Sig)
```

```
type Sam = Sampler Sig2
```

The `Sampler` is applicative and function. We can easily apply an effect with `fmap`:

```
> dac $ fmap magicCave2 $ loop song
```

We applied a reverb (`magicCave2 :: Sig2 -> Sig2`). It's taken from the library csound-expression.

If our effect produces side effects we can use one of the lifting functions:

```
liftSam :: Sample (SE a) -> Sample a
bindSam :: (Sig2 -> SE Sig2) -> Sam -> Sam
```

If we want to now the current BPM we can use functions:

```
mapBpm :: (Bpm -> Sig2 -> Sig2) -> Sam -> Sam
bindBpm :: (Bpm -> Sig2 -> SE Sig2) -> Sam -> Sam
```

And many more

There are many other functions. We can find them all in the docs. Happy sampling!

Signal segments

The signal segments lets us schedule signals with event streams. They are defined in the module `Csound.Air.Seg`. A signal segment can be constructed from a single signal or a tuple of signals:

```
toSeg :: a -> Seg a
```

It plays the signal indefinitely. We can limit the duration of the segment with static length measured in seconds:

```
constLim :: D -> Seg a -> Seg a
```

or with an event stream:

```
type Tick = Evt Unit
```

```
slim :: Tick -> Seg a -> Seg a
```

The signal is played until something happens on the given event stream. When segment is limited we can loop over it:

```
sloop :: Seg a -> Seg a
```

It plays the segment and the replays it again when it comes to an end.

If we several limited signals we can play them in sequence:

```
sflow :: [Seg a] -> Seg a
```

When the first signal stops the next one comes into play and when it stops the next one is turned on.

Also we can play segments at the same time:

```
spar :: [Seg a] -> Seg a
```

The length of the result equals to the longest length among all input segments.

We can delay the segment with an event stream or a static length:

```
sdel     :: Tick -> Seg a -> Seg a
constDel :: D    -> Seg a -> Seg a
```

There is a handy shortcut for playing nothing for the given amount of time:

```
srest     :: Num a => Tick -> Seg a
constRest :: Num a => D    -> Seg a
```

To listen the segment we need to convert it to signal:

```
runSeg :: Sigs a => Seg a -> a
```

That's it. With signal segments we can easily schedule the signals with event streams.

Let's create a button and turn the signal on when it's pressed:

```
> dac $ lift1 (\x -> runSeg $ sdel x $ toSeg $ osc 440) (button "start")
```

Let's create a second button that can turn off the signal.

```
> dac $ hlift2 (\x y -> runSeg $ sdel x $ slim y $ toSeg $ osc 440)
    (button "start")
    (button "stop")
```

When signal stops the program exits. We can repeat the process by looping:

```
> dac $ hlift2 (\x y -> runSeg $ sloop $ sdel x $ slim y $ toSeg $ osc 440)
    (button "start")
    (button "stop")
```

Let's play several signals one after another with `sflow`:

```
> dac $ hlift2 (\x y -> runSeg $ sloop $ slim y
    $ sdel x $ sloop $ sflow $ fmap (slim x . toSeg . osc) [220, 330, 440])
    (button "start")
    (button "stop")
```

Warning: Note that signal release is not working with signal segments.

Samplers

There are handy functions to trigger signals that are based on signal segments. We can look at the module `Csound.Air.Sampler` to find them.

The functions trigger the signals with event streams, keyboard presses and midi messages. Let's look at the functions for keyboard (the rest functions are roughly the same).

There are several patterns of (re)triggering.

- `Trig` -- triggers a note and plays it while the same key is not pressed again

  ```
  charTrig :: Sigs a => String -> String -> a -> SE a
  charTrig ons offs asig = ...
  ```

  It accepts a string of keys to turn on the signal and the string of keys to turn it off.

  Let's try it out:

  ```
  > dac $ at (mlp 500 0.1) $ charTrig "q" "a" $ saw 110
  ```

  Try to hit `q` and `a` keys.

- `Tap` -- is usefull optimization for `Trig` it plays the note only for a given static amount of time (it's good for short drum sounds) `Tap` has the same arguments but the turn off string is substituted with a note's length in seconds (it comes first):

  ```
  charTap :: Sigs a => D -> String -> a -> SE a
  ```

- `Push` -- plays a signal while the key is pressed.

  ```
  charPush :: Sigs a => Char -> a -> SE a
  ```

  Let's create a simple note:

  ```
  > dac $ at (mlp 500 0.1) $ charPush 'q' $ saw 110
  ```

  Let's create a couple of notes:

  ```
  > dac $ at (mlp 500 0.1) $ sum [charPush 'q' $ saw 110, charPush 'w' $ saw (110 * 9 / 8)]
  ```

  Note that only one key (de)press can be registered at the moment. It's current limitation of the library. It's not so for midi events.

- `Toggle` -- uses the same key to turn the signal on/off.

  ```
  > dac $ at (mlp 500 0.1) $ charPush 'q' $ saw 110
  ```

- `Group` -- creates a mini mono-synth. It's give a list of pairs of keys an signals. When key is pressed the corresponding signal starts playing. When the next key is pressed the previous is turned off and the current is turned on.

  ```
  charGroup :: Sigs a => [(Char, a)] -> SE a
  ```

There are many more functions. You can find them in the module `Csound.Air.Sampler`.

Turning keyboard to DJ-console

Let's create a mini mix board for a DJ. The first thing we need is a cool dance drone:

```
> let snd1 a b = mul 1.5 $ mlp (400 + 500 * uosc 0.25) 0.1 $ mul (sqrSeq [1, 0.5, 0.5, 1, 0.5, 0.5, 1, 0.5] b) $ saw a
```

Let's trigger it with keyboard!

```
> dac $ charTrig "q" "a" (snd1 110 8)
```

Try to press `q` and `a` keys to get the beat going. Let's create another signal. It's intended to be high pitched pulses.

```
> let snd2 a b = mul 0.75 $ mul (usqr (b / 4) * sqrSeq [1, 0.5] b) $ osc a
```

Let's try it out. Try to press `w`, `e`, `r` keys.

```
> dac $ mul 0.5 $ sum [charPush 'w' $ snd2 440 4, charPush 'e' $ snd2 330 4, charPush 'r' $ snd2 660 8]
```

Note that only one keyboard event can be recognized. So if you press or depress several keys only one is going to take effect. It's a limitation of current implementation. It's not so with midi events. Let's join the results:

```
> let pulses = mul 0.5 $ sum [charPush 'w' $ snd2 440 4, charPush 'e' $ snd2 330 4, charPush 'r' $ snd2 660 8]
> let beat = mul 0.5 $ sum [charTrig "q" "a" (snd1 110 8), charTrig "t" "g" $ snd1 220 4]
```

Let's create some drum sounds:

```
> let snd3 = osc (110 * linseg [1, 0.2, 0])
> let snd4 = mul 3 $ hp 300 10 $ osc (110 * linseg [1, 0.2, 0])
> let drums = sum [charTrig "z" "" snd3, charTrig "x" "" snd4]
```

Let's rave along.

```
> dac $ sum [pulses, mul 0.5 beat, mul 1.2 drums]
```

- <= Samples
- => Widgets for live performances
- Home

Widgets for live performances

Since the version 4.2.0 there are many widgets tageted at real-time performance. They should make it easy to mix and process audio live.

It's assumed that the library `csound-sampler` is installed.

Playing samples

We can start and stop samples with function `sim`.

```
module Main where

import Csound.Base
import Csound.Sam

a1 = infSig1 $ osc 220
a2 = infSig1 $ osc 330

main = dac $ do
    (g, sam) <- sim 4 [("220", a1), ("330", a2)]
    panel g
    mul 0.5 $ runSam 120 sam
```

For simplicity we use pure sine waves but we can use samples with cool sounds instead.

The first argument for `sim` (it's 4 in the example above) is responsible for syncronization. The samples are started only on every n'th beat.

we can toggle between samples with the function `tog`. The example is the same but write `tog` in place of `sim`. With `tog` only one sample is going to be played.

The widget `live` resembles the session view of the Ableton. the samples are arranged in matrix. We can start all samples in the row by the single click of the mouse and we can toggle samples within each column. Let's look at the example:

```
module Main where

import Csound.Base
import Csound.Sam

b1 = infSig1 $ sqr 220
b2 = infSig1 $ sqr 330
b3 = infSig1 $ sqr 440
```

```
c1 = infSig1 $ tri 220
c2 = infSig1 $ tri 330
c3 = infSig1 $ tri 440

main = dac $ do
    (g, sam) <- live 4 ["triangle", "square"]
        [ c1, b1
        , c2, b3
        , c3, b3]
    panel g
    mul 0.3 $ runSam 120 sam
```

the function `live` takes in the number of beats for syncronization, the names for columns and the list of samples. The number of columns in the matrix is defined with the length of list of the column names.

Using mixer

We can mix several stereo signals together with the widget mixer.

```
mixer :: [(String, SE Sig2)] -> Source Sig2
```

Mixer takes in the list of pairs. The first element of the pair is the name of the instrument and the second element is the actual signal.

Let's balance the sound of the chord:

```
main = dac $ do
    (g, res) <- mixer $ fmap (\x -> mixMono (show x) (osc $ sig $ int x)) [220, 330, 440]
    win "mixer" (600, 300) g
    return $ mul 0.5 $ res
```

Note the function `win`. It constructs the window with the given name, size and content. The function `mixMono` is usefull for mixing mono signals.

We can use mixer with functions `sim` and `tog`:

```
a1 = infSig1 $ osc 220
a2 = infSig1 $ osc 330

run = runSam 120

main = dac $ do
    (g1, sam1) <- tog 4 [("220", a1), ("330", a2)]
    (g2, sam2) <- sim 4 [("220", a1), ("330", a2)]
    (g3, res)  <- mixer [("tog", run sam1), ("sim", run sam2)]
    win "main" (600, 400) $ ver [sca 0.6 $ hor [g1, g2], g3]
    return res
```

Processing signals

There are many widgets to process stereo signals. The sound processing function is a function of the type:

```
type FxFun = Sig2 -> SE Sig2
```

To be truly interesting the sund processing function should depend on parameters which control the behavior of the effect:

```
Sig -> Sig -> ... -> Sig -> FxFun
```

We can create a visual representation of this type with `fxBox`.

```
fxBox :: FxUI a => String -> a -> Bool -> [(String, Double)] -> Source FxFun
fxBox name fx isOn args = ...
```

It expects the name of the widget, the sound processing function the flag that turns on the widget (is it active at the start time) and the list of arguments. The result contains the widget and fx-function.

The class `FxUI` contains the functions like:

```
Sig2 -> SE Sig2
Sig -> Sig2 -> SE Sig2
Sig -> Sig -> Sig2 -> SE Sig2
...

Sig2 -> Sig2
Sig -> Sig2 -> Sig2
Sig -> Sig -> Sig2 -> Sig2
...
```

I hope that you've got the pattern. The arguments are turned into sliders. There are many predefined widgets that implement typical effects (reverbs, distortion, chorus, flanger etc).

```
main = dac $ do
    (gui, fx) <- fxHor
        [ uiFilter False 0.5 0.5 0.5
        , uiChorus False 0.5 0.5 0.5 0.5
        , uiPhaser False 0.5 0.5 0.5 0.5 0.5
        , uiReverb True  0.5 0.5
        , uiGain   True  0.5
        ]
    win "main" (900, 400) gui
    fx $ fromMono $ saw 110
```

We can group the fx-widgets with functions `fxHor`, `fxVer` and `fxSca`. They group widgets horizontaly, verticaly and scale the widgets. There are many more widgets to consider you can find them in the module `Csound.Air.Live`.

- <= Signal segments
- => Padsynth algorithm
- Home

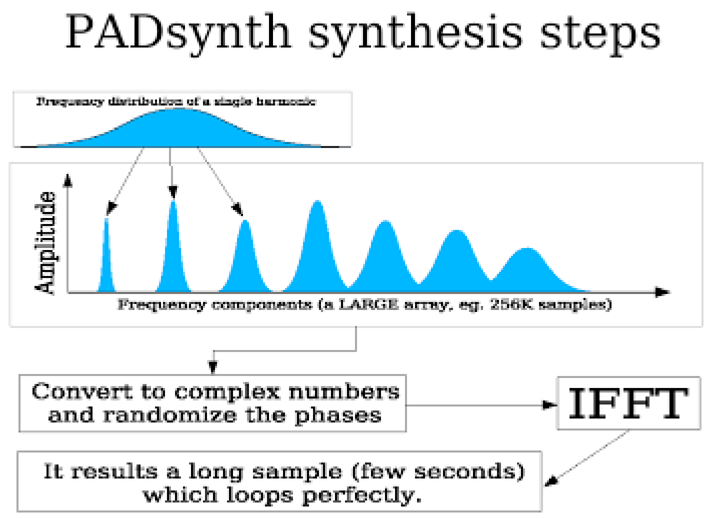
Padsynth algorithm

Padsynth is an interesting technique to make you timbre alive. It's created by Paul Nasca in his famous synthesizer ZynAddSubFX. It was ported to Csound by Michael Gogins. It requires at least Csound 6.05.

The main idea lies in the notion that all cool sounds are inharmonic. They can be harmonic but there are tiny fluctuations and digressions from the ideal shape. The human ear catches those tiny fluctuations and this what can make a difference between dull digital sound and warm analog sound.

The main idea is to add continuous sidebands to each harmonic, then we can apply invere fourier transform to the spectrum and get inharmonic wave. Then we can write the audio wave to table and play it back with an oscillator. It's going to be periodic. But the period of repetition is quite large (about 5-10 seconds).

So in place of single harmonics we get Gaussian curves with peaks at the given harmonics:


PADsynth synthesis steps
Frequency distribution of a single harmonic
Amplitude
Frequency components (a LARGE array, eg. 256K samples)
Convert to complex numbers and randomize the phases
IFFT
It results a long sample (few seconds) which loops perfectly.

The algorithm is described by the author here.

Also you can read the Csound docs.

The Csound provides only GEN routine to create the PADsynth long ftables. But the algorithm is so much useful that I've decided to supply many more functions to make it easy to create beautiful PADsynth-instruments. There are predefined patches that use the algorithm.

Let's start with simplest functions and then we can dive deeper.

PADsynth standard audio waves

There are versions of standard audio waves: pure sine, triangle, square, sawtooth that are enriched by padsynth algorithm. They have the same prefix `bw`:

```
type PadsynthBandwidth = Double
```

```
bwOsc, bwTri, bwSqr, bwSaw :: PadsynthBandwidth -> Sig -> SE Sig
```

The good values for padsynth bandwidth are 0 to 120. When it increases it creates chorus like effect. You can hear the difference between pure sine and sine with bandwidth:

```
> dac $ osc 220
> dac $ bwOsc 15 220
> dac $ bwOsc 45 220
> dac $ bwOsc 85 220
```

Let's listen to the saw filtered with moog-like low pass filter:

```
dac $ at (mlp (2500 * linseg [0, 3, 1, 4, 0]) 0.2) $ bwSaw 45 220
```

Let's modulate the filter with LFO:

```
dac $ at (mlp (1500 * utri (2 + 2 * usaw 1)) 0.2) $ bwSaw 65 70
```

Let's try the same algorithm but with different bandwidth:

```
dac $ at (mlp (1500 * utri (2 + 2 * usaw 1)) 0.2) $ bwSqr 2 70
```

There is an oscillator with given list harmonics:

```
bwOscBy :: PadsynthBandwidth -> [Double] -> Sig -> Sig
```

Stereo waves

The padsynth algorithm can become more alive and natural when we use separate oscillators for each channel. There are "stereo"-versions for most of padsynth-related functions. So there are stereo oscillators:

```
bwOsc2, bwTri2, bwSqr2, bwSaw2 :: PadsynthBandwidth -> Sig -> SE Sig
```

```
bwOscby2 :: PadsynthBandwidth -> [Double] -> Sig -> SE Sig2
```

The signals in each channel have different phase. The phase is random for each note.

PADsynth oscillators

There is a generic PADsynth-oscillator:

```
padsynthOsc :: PadsynthSpec -> Sig -> SE Sig
```

It takes in padsynth initialization parameters and produces an oscillator. Let's look at those parameters:

```
-- | Padsynth parameters.
--
-- see for details: <http://csound.github.io/docs/manual/GENpadsynth.html>
data PadsynthSpec = PadsynthSpec
    { padsynthFundamental     :: Double
    , padsynthBandwidth       :: Double
    , padsynthPartialScale    :: Double
    , padsynthHarmonicStretch :: Double
    , padsynthShape           :: PadsynthShape
    , padsynthShapeParameter  :: Double
    , padsynthHarmonics       :: [Double]
    } deriving (Show, Eq)

data PadsynthShape = GaussShape | SquareShape | ExpShape
```

Wow! Lots of parameters.

- Fundamental -- is the frequency of the note that is stored in the table.
- Bandwidth -- is the bandwidth of harmonic. How wide we should spread the harmonics.
- PartialScale -- Is the ratio with which we increase the bandwidth for each subsequent harmonic. There is a notion that for the sound to sound natural the bandwidth should become bigger when we go from lower harmonics to higher. This parameter declares
- HarmonicStretch -- ratio of stretch of the overtones
- Shape -- shape of the single harmonic (gaussian, square or exponential)
- ShapeParameter -- shape parameter of the curve.
- Harmonics -- list of relative amplitudes of the partials

There seems to be too many parameters to set! But there is a handy function to set reasonable defaults:

```
defPadsynthSpec :: Double -> [Double] -> PadsynthSpec
```

It requires only bandwidth and harmonics. Also you can modify some parameters like this:

```
> (defPArameters 45 [1, 0.5, 0.1]) { padsynthPartialScale  = 2.3  }
```

Let's listen to the sound of some harmonics:

```
> let wave cps = padsynthOsc (defPadsynthSpec 25 [1, 0.5, 0, 0.2]) cps
> dac $ at (mlp (150 + 2500 * uosc 0.25) 0.1) $ wave $ constSeq [110, 137, 165, 220] 6
```

We modify the center frequency of moogladder low-pass filter with LFO. The frequency is created with running sequence of four notes.

It's useful to be able to assign different harmonic content to different regions of frequencies. We can do it with :

```
padsynthOscMultiCps :: [(Double, PadsynthSpec)] -> D -> SE Sig
padsynthOscMultiCps specs frequency = ...
```

The list of pairs contains thresholds for frequencies and padsynth specifications. The given padsynth specification is going to be applied to all notes with frequencies that are below the given threshold and above of the threshold of the previous element in the list.

There is a function that can apply different padsynth specs according to the value of the amplitude.

```
padsynthOscMultiVol :: [(Double, PadsynthSpec)] -> (D, Sig) -> SE Sig
padsynthOscMultiVol specs (amplitude, frequency) = ...
```

There are stereo versions of the padsynth oscillators:

```
padsynthOsc2 :: PadsynthSpec -> Sig -> SE Sig2

padsynthOscMultiCps2 :: [(Double, PadsynthSpec)] -> D -> SE Sig2

padsynthOscMultiVol2 :: [(Double, PadsynthSpec)] -> (D, Sig) -> SE Sig2
```

Low level PADsynth table generator

If the default oscillators are not good for you and you want to implement your own you may beed to create the padsynth ftable first. It's not that hard to do if we understand the `PadsynthSpec` data type (see prev section).

We can create a table with a following function:

```
padsynth :: PadsynthSpec -> Tab
```

PADsynth instruments

The package `csound-catalog` contains many predefined instruments that are based on padsynth algorithm. They take in a spectrum of Sharc instrument and create a padsynth instrument with it:

```
psOrganSharc :: SharcInstr -> Patch2
psPianoSharc :: SharcInstr -> Patch2
psPadSharc :: SharcInstr -> Patch2
psSoftPadSharc :: SharcInstr -> Patch2
```

There are about 30 predefined sharc instruments. The sharc instrument contains spectrum of some orchestral instrument. You can find the full list of sharc instruments in the module Csound.Patch (Section Sharc instruments > Concrete instruments)

Let's listen to some of them (recall that we need to import the `Csound.Patch` module to use the predefined patches):

```
> :m +Csound.Patch
> vdac $ atMidi $ psSoftPadSharc shAltoFlute
> vdac $ atMidi $ psOrganSharc shCello
> vdac $ atMidi $ psPiano shTrumpetC
```

The timbre of an instrument can be altered by changing the bandwidth of padsynth. There are special versions of aforementioned functions that allows to alter specific parameters (The function name stays the same but it's followed by `'`).

```
data PadSharcSpec = PadSharcSpec {
        padSharcBandwidth :: Double,
        padSharcSize      :: Int
    }

psPadSharc' :: PadSharcSpec -> SharcInstr -> Patch2
```

The type `PadSharcSpec` is defined in the module `Csound.Catalog.Wave` (see SHARC section). It contains two parameters:

- Bandwidth -- bandwidth for padsynth ftables
- Size -- number of frequency regions (1 to 40)

The size determines how many tables are going to be used. The default is 15.

There is an instance of `Default` class for `PadSharcSpec`:

```
instance Default PadSharcSpec where
    def = PadSharcSpec 15 8
```

So if we want to alter only bandwidth we can do it like this:

```
vdac $ atMidi $ psSoftPadSharc' (def { padSharcBandwidth = 56 }) shAltoFlute
```

There are many more functions they are related to altering reverb effect for the instruments and the number of frequency regions. We can increase the number of regions if we use the suffix `Hifi`:

```
vdac $ atMidi $ psLargeOrganSharcHifi shAltoFlute
```

Deep pads

The padsynth algorithm is super cool for creation of pads. There are predefined functions that create great pads. They have vedic names:

```
vibhu, rishi, agni, prakriti, rajas, avatara, bhumi :: PadsynthBandwidth -> Patch2
```

The only argument is the bandwidth for underlying tables.

You can try them out:

```
> dac $ atMidi $ vibhu 35
> dac $ atMidi $ vibhu 0.6
```

You can switch `dac` to `vdac` if you don't have the real hardware midi device attached to your computer.

Pads with crossfades

There are cool instruments that allow to morph between several timbres. Right now they are defined only for pads. They have got suffix `Cfd` for morphing of two timbres and `Cfd4` for morphing four timbres:

```
psPadSharcCfd :: Sig -> SharcInstr -> SharcInstr -> Patch2
psPadSharcCfd cfdLevel instr1 instr2 = ...

psPadSharcCfd4 :: Sig -> Sig -> SharcInstr -> SharcInstr -> SharcInstr -> SharcInstr -> Patch2
psPadSharcCfd4 cfdLevelX cfdLevelY instr1 instr2 instr3 instr4 = ...
```

The `cfdLevel` lies in the interval `(0, 1)`. The `0` produces only first instrument and the `1` produces only second instrument. So we have the mixture of two timbres. Also we can create the mixture of four signals. But in this case we have two levels: `cfdLevelX` and `cfdLevelY`. We can imagine that timbres lie at the corners of the rectangle. The levels define the coordinates of the point that lies inside the rectangle. The output timbre is produced with bilinear interpolation of timbres that lie at the corners of the rectangle. The values for levels lie at the interval `(0, 1)`. The `0` means left corner (or bottom) and `1` stands for right corner (or top corner).

Let's create a simple crossfade pad:

```
vdac $ atMidi $ psPadSharcCfd (uosc 0.25) shAltoFlute shCello
```

reminder: the `uosc` produces unipolar sine wave with given frequency.

There are many more functions. They have different prefixes:

```
psSoftPadSharcCfd, psDeepPadSharcCfd, psDeepSoftPadSharcCfd, ...
```

See the full list at the module `Csound.Patch`.

Also there are deep pads with corssfades:

```
vedicPadCfd :: Sig -> SharcInstr -> SharcInstr -> PadsynthBandwidth -> Patch2
vedicPadCfd cfdLevel instr1 instr2 bandwidth = ...

vedicPadCfd4 :: Sig -> Sig -> SharcInstr -> SharcInstr -> SharcInstr -> SharcInstr -> PadsynthBandwidth -> Patch2
vedicPadCfd4 cfdLevelX cfdLevelY instr1 instr2 instr3 instr4 bandwidth = ...
```

They are particularly useful to test timbres with different values for bandwidth (it's the last input argument). Good values lie at the interval `(0.01, 130)`.

There are crossfade versions of specific pads: `vibhuRishi`, `vibhuAgni`, `rishiPrakriti` and so on. They take in the bandwidth and crossfade level:

```
> dac $ mul 2 $ vibhuRajas 45 (uosc 0.25)
```

Also we can use a randomized signal to control the crossfade level:

```
> dac $ do { k <- 0.5 + jitter 0.5 0.1 0.2;  mul 2 $ atMidi $ vibhuRajas 65 k }
```

We can create a timbral shimmer effect if we increase the rate of randomized crossfade level:

```
> dac $ do { k <- 0.5 + jitter 0.5 1 8;  mul 2 $ atMidi $ vibhuRajas 65 k }
```

Padsynth and noise

we can make the pad more interesting if we add some noise. There are two predefined patches that illustrate this idea: `noisyRise` and `noisySpiral`:

```
> vdac $ atMidi noisyRise
> vdac $ atMidi noisySpiral
```

- <= Widgets for live performances
- => Granular synthesis
- Home

Granular synthesis

Granular synthesis is good for creation of atmospheric ambient textures. We can take a plain violin note in the sustain phase and turn it into wonderful soundscape.

The Csound contains a set of functions for granular synthesis. Unfortunately they are very hard to use due to large number of arguments. This module attempts to set most of the arguments with sensible defaults. So that a novice could start to use it. The defaults are implemented with the help of the class `Default`. It's a standard way to implement defaults in the Haskell. The class `Defaults` defines a single constant called `def`. With `def` we can get the default value for the given type. Several csound opcodes are reimplemented so that first argument contains secondary parameters. The type for parameters always has the instance for the class `Default`. The original csound opcodes are defined in the end of the module with prefix `csd`.

Also many granular synthesis opcodes expect the sound file as input. There are predefined versions of the opcodes that take in the file names instead of tables with sampled sound. They have suffix `Snd` for stereo and `Snd1` for mono files.

For example, that's how we can use the @granule@ opcode:

```
> dac $ granuleSnd1 spec [1, 2, 3] grainSize "fox.wav"
```

No need to set all 22 parameters.

The four functions are reimplemented in this way: `sndwarp`, `syncgrain`, `partikkel`, `granule`.

The most often used arguments are:

- Scale factors for tempo and pitch: `TempoSig` or `speed` and `PitchSig`. Ranges in 0 to 1.
- Grain size is the size of produced grains in seconds. Good range is 0.005 to 0.01 or even 0.1. The higher the value the more it sounds like the original sound.
- Grain rate. It's the speed of grain production in Hz. If it's in audio range we can no longer perceive the original pitch of the file. Then the pitch is determined with grain rate value.
- Grain gap. It's the gap in samples between the grains. Good values are 1 to 100.
- Grain window function. For the sound to be a grain it have to be enveloped with grain window (some sort of bell shaped envelope). We can use half-sine for this purpose (and it's so in most of the defaults) or we can use a table in the `GEN20` family. In the library they implemented as window tables see the table constructors with prefix `win`.

Usual order of arguments is: `GrainRate`, `GrainSize`, `TempoSig`, `PitchSig`, file `table` or `name`, `poniter` to the table.

Let's study some examples. We assume that there is a file `"fox.wav"` in the current directory.

```
file = "fox.wav"
```

We assume that it contains a long note in the sustain phase. It varies but not so much.

Grainy

The simplest granular function is `grainy`. Grainy is based on the function `partikkel`. It's the most basic version of it. Here is the signature:

```
grainy :: GrainRate -> GrainSize -> TempoSig -> PitchSig -> String -> Sig2
```

As we can see it takes the grain rate and size, scaling factors for tempo and pitch, file name. It produces the stereo signal. It expects the stereo file as input (for mono files there is a function `grainy1`).

Let's see how the grain rate nd grain size affect the sound:

```
> dac $ grainy 200 (linseg [0.1, 5, 0.01]) 1 1 file
> dac $ grainy (linseg [200, 5, 10]) 0.1 1 1 file
```

In the first function we change the grain size. And in the second example we change the grain rate.

Sndwarp

we can change the tempo and pitch of the sound with `sndwarp`. Also we can add a special grainy noise if we change the secondary parameters. secondary parameters are defined in the structure `SndwarpSpec` (short for sndwarp specification). If we are too lazy to care for the parameters we can supply the default value. The `SndwarpSpec` is instance of `Default` so we can use the constant `def`.

```
sndwarp :: SndwarpSpec -> TempoSig -> PitchSig -> Tab -> Sig
```

Let's create a drone sound. We can create the drone if we lower the pitch down an octave. Then we can read a small portion of the file (just half a second). We can control the read position with special function `ptrSndwarpSnd`

```
ptrSndwarpSnd :: SndwarpSpec -> PitchSig -> String -> Pointer -> Sig2
```

It takes in not only a file but also a pointer to the reading position (in seconds). we can create a slow motion of the playhead with function `linseg`:

```
> dac $ ptrSndwarpSnd def 0.5 w2 (linseg [0, 10, 0.5, 10, 0.25])
```

Let's create a more involved example. Let's create continuous sound. We are going to trigger long notes so that the next one starts just several seconds before the current one is stopped. Each note is going to play an audio file with `sndwarp` that scales the pitch with random notes from the given scale.

Let's create an instrument:

```
instr dt file n = do
    a <- random 1.5 (lengthSnd file - 1.5)
    b <- random (-1) 1
    iwin <- random 0.4 1
    let a1 = a
        a2 = a + b
        spec = def { sndwarpWinSize = iwin, sndwarpRandw = iwin / 3 }
    return $ mul (0.5 * env) $ at (mlp (12000 * env) 0.5)
        $ ptrSndwarpSnd spec (sig n) file (linseg [a1, dt, a2])
    where
        env = linseg [0, 0.2 * dt, 1, 0.4 * dt, 1, 0.4 * dt, 0]
```

The instrument takes in a note duration, filename and the pitch scaling factor. It creates a short interval to read from and reads the file. The grains are scaled by pitch.

Let's trigger the instrument:

```
grainOcean :: D -> [D] -> String -> Sig2
grainOcean dt scale file = at largeHall2 $ mul 0.5
    $ sched (instr dt file) $ withDur dt
    $ oneOf scale $ metroE (recip $ sig $ dt * 0.8)
```

Here we create a stream of events with a period that is slightly shorter than the total length of the note (so that there is an intersection of the notes).

```
metroE (recip $ sig $ dt * 0.8)
```

Then we pick pitches at random from the given scale (`oneOf`):

```
oneOf scale $ metroE (recip $ sig $ dt * 0.8)
```

Then we trigger the instrument and add a reverb:

```
at largeHall2 $ mul 0.5
    $ sched (instr dt file) $ withDur dt
    $ oneOf scale $ metroE (recip $ sig $ dt * 0.8)
```

Let's invoke the function:

```
> dac $ grainOcean 16 [1, 9/8, 6/5, 3/2, 2, 0.5, 3/4, (3/2) * (5/4), 6/2] "fox.wav"
```

Granule

With granule we can create a clouds of grains. We can supply a list of pitch scaling factors so that the resulting sound plays a chord:

```
type ConstPitchSig = D
type GrainSize = Sig

granuleSnd :: GranuleSpec -> [ConstPitchSig] -> GrainSize -> String -> Sig2
```

The second argument is a chord of pitches.

Let's study an example:

```
> dac $ granuleSnd def [1, 3/2, 2, 0.5] 0.2 "fox.wav"
```

Syncgrain

The `syncgrain` implements synchronous granular synthesis. The grains are created not at random but with some law. The `syncgrain` can dramatically change the sound:

```
syncgrainSnd :: SyncgrainSpec -> GrainSize -> TempoSig -> PitchSig -> String -> Sig2
```

Here is an example

```
> dac $ smallHall2 $ syncgrainSnd def 0.01 (1.5) 0.3 "fox.wav"
```

Let's study the three parameters. we are going to change them with knobs:

```
> dac $ mul 0.5 $ hlift3 (\a b c -> smallHall2 $
    syncgrainSnd def (0.2 * a) (-2 + 4 * b) (-2 + 4 * c) file)
    (uknob 0.7) (uknob 0.7) (uknob 0.7)
```

There are many more functions to study. Take a look at the module `Csound.Air.Granular`.

- <= Padsynth algorithm
- => Arguments modulation
- Home

Argument modifiers

Argument modifiers make it easy to add an LFO or small amount of noise to parameters of synthesizer.

Let's consider a plain sine wave:

```
> dac $ osc 220
```

Let's add a vibrato:

```
> dac $ osc (220 * (1 + 0.05 * osc 2))
```

What if we want to add a noisy vibrato:

```
> dac $ osc (220 * (1 + 0.05 * white))

<interactive>:6:30:
    Couldn't match expected type â€˜Sigâ€™ with actual type â€˜SE Sigâ€™
    In the second argument of â€˜(*)â€™, namely â€˜whiteâ€™
    In the second argument of â€˜(+)â€™, namely â€˜0.05 * whiteâ€™
```

Ooops, we've got an error! That's because `osc 2` has type `Sig` and `white` has type `SE Sig`. There is a type mismatch and compiler just reminds us about it.

But can we abstract out this pattern of vibrato and devise such a function that we can easily use both types `Sig` and `SE Sig`. There is such a function!

It's defined in the module `Csound.Air.ModArg`. It's called `modArg1`. Let's see it in action:

```
> dac $ modArg1 0.05 (osc 2) osc 220
```

It takes three parameters:

```
> modArg1 depth modSignal function
```

It transforms a function so that the first argument is modulated with `modSignal` with given `depth`. It's defined so that we can use both types `Sig` and `SE Sig` for `modSig`:

```
> dac $ modArg1 0.05 white osc 220
```

It might seem that `modArg1` takes in four arguments but the last argument `220` is the argument for modified function. We may write it like this to clarify it:

```
> let vibrOsc = modArg1 0.05 white osc
> dac $ vibrOsc 220
```

`modArg1` can modify functions with up to four parameters. The output of the function should be one of the following types:

```
Sig, Sig2, SE Sig, SE Sig2
```

Also there are siblings: `modArg2`, `modArg3` and `modArg4`. They can modify second, third and fourth arguments of the function. All functions take in depth of modulation, modulation signal and the function to transform. The functions are defined so that the wiring is hidden from the user. If modulated signal is pure it's just applied to the argument if it contains side effects than function output will have side effects too!

Let's look at another example. Let's modulate the filter's center frequency:

```
> dac $ at (modArg1 0.17 (osc 2) mlp 1750 0.2) $ white
```

We can also modulate second argument too:

```
dac $ at (modArg2 0.4 (osc 8) (modArg1 0.17 (osc 2) mlp) 1750 0.5) $ white
```

We can add some noise to the modulation:

```
> dac $ at (modArg2 0.4 (osc 8) (modArg1 0.17 (mul (uosc 2) white) mlp) 1750 0.5) $ white
```

Delayed modulation

Sometimes we want the modulation to start aftter some initial delay. Take the vibrato for instance. Often there s no vibrato at the attack and then it starts to rise. We can simulate it with the function:

```
delModArg1 delTime riseTime depth modSig function
```

It takes in two more parameters. The first is time of delay and the second is time to rise the modulation depth from zero to the given maximum amount. Let's take a look at the example:

```
> dac $ delModArg1 0.5 1 0.03 (osc 4) osc 220
```

The cool thing to know about modulation signal is that it's a signal. It's parameters can vary too. Let's increase the vibrato rate over time:

```
dac $ delModArg1 0.5 1 0.03 (osc (linseg [3, 3, 8, 4, 4])) osc 220
```

The function `delModArg` is also defined for 1, 2, 3, 4 arguments.

Predefined patterns of modulation

There common ways to modulate signals. Let's look at some of them. For every pattern `N` can be 1, 2, 3 or 4. The full list of functions can be found in the module `Csound.Air.ModArg`.

Oscillators

The modulation most often happens with some LFO. There are predefined functions:

```
oscArgN depth rate function
```

Also there are LFOs with other wave shapes: `triArgN`, `sqrArgN`, `sawArgN`. There are LFOs with random phases: `rndOscArgN`, `rndTriArgN`, `rndSqrArgN`, `rndSawArgN`. There are delayed versions of these functions all of them has prefix `del`.

Let's revrite the vibrato example:

```
> dac $ oscArg1 0.05 4 osc 220
```

Let's delay the vibrato and make it saw-tooth shape:

```
> dac $ delSawArg1 0.5 1 0.05 4 osc 220
```

Noise generators

We can add some noise to parameters to imitate aliveness of the acoustic instruments. There are severl types of noises:

- White Nose: `noiseArgN`.
- Pink noise: `pinkArgN`.
- gauss noise: `gaussArgN`.
- gauss noise with frequency of generation of new random values: `gaussiArgN depth cps`.
- jitter noise: `jitArgN depth cpsMin cpsMax`. It generates random nombers from -1 to 1 within the given interval of frequency of generation of new numbers.

The rest arguments are the same as with oscillators. They are `depth` and `function`. The first argument is always depth of modulation.

It's a common trick to add some liveness to the sound with randomizing the parameters. We add a bit of noise or randomness to the center frequency of the filter or to the resonance. It makes the insturments more interesting.

Let's create a Pad sound with no modulation:

```
> vdac $ midi $ onMsg $ mul (fades 0.5 0.5) . at (mlp 1200 0.15) . saw
```

Let's add a vibrato:

```
> vdac $ midi $ onMsg $ mul (fades 0.5 0.5) . at (mlp 1200 0.15) . delOscArg1 0.3 0.8 4 saw
```

Let's modulate the parameters of the filter:

```
> vdac $ midi $ onMsg $ mul (fades 0.5 0.5) . at ((gaussArg1 0.31 (noiseArg2 0.2 mlp)) 1000 0.15) . delOscArg1 0.3 0.8 0.013 4 saw
```

Let's add a reverb:

```
> vdac $ mixAt 0.25 largeHall2 $ midi $ onMsg $ mul (fades 0.5 0.5) . at ((gaussArg1 0.31 (noiseArg2 0.2 mlp)) 1000 0.15) . delOscArg1 0.3 0.8 0.013 4 saw
```

We can lower the center frequency and increase the volume, to make sound more spacy:

```
> vdac $ mul 2.5 $ mixAt 0.25 largeHall2 $ midi $ onMsg $ mul (fades 0.5 0.5) . at ((gaussArg1 0.31 (noiseArg2 0.2 mlp)) 550 0.15) . delOscArg1 0.3 0.8 0.013 4 saw
```

Envelopes

Also there are predefined functions for common envelopes:

```
 adsrArgN depth att dec sust rel function   -- linear
xadsrArgN depth att dec sust rel function   -- exponential
```

Also there are delayed versions that add initial delay time:

```
 delAdsrArgN delTime depth att dec sust rel depth function   -- linear
delXadsrArgN delTime depth att dec sust rel depth function   -- exponential
```

Note that there is no riseTime as it's the same as attack portion of the envelope. It's often useful to modulate the center frequency of the envelope:

```
> dac $ at (adsrArg1 1 0.5 0.5 0.1 0.3 mlp 1500 0.1) $ saw 110
```

- <= Granular synthesis
- => Csound API. Using generated code with another languages
- Home

Csound API. Using generated code with other languages

The cool thing about Csound is that it's not only a text to audio converter. It's also a C-library! Also it has bindings to many languages! Python, Java, Clojure, Lua, Clojure, Csharp, C++, racket, VB! Also it works on Android, iOS, and RaspPi.

We can create audio engine with Haskell and then we can wrap it in the UI written in some another language! Let's look at how it can be done.

Interaction with generated code.

We can interact with Csound with two main methods.

- Channels for updating values.
- Named instruments for triggering instruments.

Channels

With channel we can update specific value inside Csound code. We can create a global channel and then send write or read values with another program.

We can make a channel and the we can read/write values. We can pass four types of values:

- Constant doubles (`chnGetD` / `chnSetD`)
- Control rate signals. Signals to control the audio (`chnGetCtrl` / `chnSetCtrl`)
- Audio rate signals. Signals that encode the audio output. (`chnGetSig` / `chnSetSig`)
- Strings (`chnGetStr` / `chnSetStr`)

Let's create a pair of channels to control the volume and the frequency of audio signal:

```
module Main where

import Csound.Base

volume      = text "volume"
frequency   = text "frequency"

instr = do
```

```
    vol <- chnGetCtrl volume
    cps <- chnGetCtrl frequency
    return (vol * osc cps)

main = writeCsd "osc.csd" instr
```

So we have a file `main.csd` that encodes our audio engine. Let's create Python program to control our audio. We are going to use Csound API and we need to install the python bindings. On Debian/Ubuntu we can install it with `apt-get`:

```
> sudo apt-get python-csound
```

On Windows it's installed with Csound installer. You can download it from the official Csound site. On OSX we can install it with `brew`.

The source code with examples can be found at the github directory.

How to use channels with Python

There is a cool GitHub project Csound API examples that shows how to ue the Csound with various languages. We can quikly check how to use the audio engine that we have generated with our language of choice. We are going to illustrate the Csound API workflow with Python. The python examples is based on the information from this repo.

```
import csnd6

class Controll:
    def __init__(self, volume, frequency):
        engine = csnd6.Csound()
        engine.SetOption("-odac")
        engine.Compile("osc.csd")

        thread = csnd6.CsoundPerformanceThread(engine)
        thread.Play()

        self.engine = engine
        self.thread = thread

        self.set_volume(volume)
        self.set_frequency(frequency)

    def set_volume(self, volume):
        self.engine.SetChannel("volume", volume)

    def set_frequency(self, frequency):
        self.engine.SetChannel("frequency", frequency)

    def close(self):
        self.thread.Stop()
        self.thread.Join()
```

We create an object that can start a Csound engine and update volume and frequency. In the initialization step we create an audio engine? load file "osc.csd" to it and start csound in the separate thread:

```
        engine = csnd6.Csound()
        engine.SetOption("-odac")
        engine.Compile("osc.csd")

        thread = csnd6.CsoundPerformanceThread(engine)
        thread.Play()
```

Then we save the state for the object:

```
        self.engine = engine
        self.thread = thread
```

and set the initial values for frequency and volume:

```
        self.set_volume(volume)
        self.set_frequency(frequency)
```

These functions update values for csound channels. So with channels we can propagate changes from python to csound:

```
    def set_volume(self, volume):
        self.engine.SetChannel("volume", volume)
```

The last method `close` stops the engine:

```
    def close(self):
        self.thread.Stop()
        self.thread.Join()
```

What's interesting with thism code is that we can control our engine within the python interpreter. It's very simple skeleton for creation of Live coding with python and haskell combo! Let's try some commands. Navigate to the directory with our python file `oscil.py`:

```
$ python
> from oscil import Controll
> c1 = Controll(0.5, 220)
> c1.set_frequency(440)
> c1.set_volume(0.3)
> c1.set_volume(0.1)
> c1.close()
```

We can instantiate several Csound audio engines!

```
> c1 = Controll(0.5, 220)
> c2 = Controll(0.3, 330)
> c3 = Controll(0.6, 110)
> c3.set_frequency(150)
>
> for c in [c1, c2, c3]:
>   c.close()
```

With channels we can update a continuous signal. With Csound API we can also trigger the instruments with notes or messages.

Messages

We can send messages to instruments. To send the message we need to know the numeric identifier of the instrument. When we use Csound directly we know what numbers do we assign to the instruments. But Haskell wrapper hides this process from us.

Csound also provides named instruments. We can assign not only unique numeric value to the instrument, but also a name as a string. There is no need to use the named instruments in the haskell wrapper since we can use plain haskell values to construct instruments and framework will take care about allocation of integer identifiers. But named insturments can help us when we want to trigger instrument with program that is written in another language through Csound API.

There is a function:

```
trigByName :: (Arg a, Sigs b) => String -> (a -> SE b) -> SE b
trigByName name instrument = aout
```

It takes an instrument name and instrument definition and creates an instrument with the given name. We can not use this instrument with Haskell. There are no way to trigger it. But we can trigger it with Csound API.

All basic Csound API functions can be found in the module `Csound.Control.Instr` (see the API section).

Let's write a simple program:

```
module Main where

import Csound.Base

instr :: (D, D) -> SE Sig
instr (amp, cps) = return $ (sig amp) * fades 0.01 0.1 * osc (sig cps)

main = writeCsd "message.csd" $ trigByName "osc" instr
```

If we run this code with `runhaskell` it will produce the `message.csd` file that contains the definition of our audio engine.

We create an instrument that has name `osc`. It takes in amplitude and frequency and produces mono output.

The Csound API the csound thread object has a method `InputMessage`. That takes in a string with Csound note-triggering expression. If you know the Csound the syntax of `i-score` statment should be straightforward to you. But don't skip the next section. It explains not only the Csound syntax but also how it's related to Haskell code.

Csound i-score statment

The Csound musicians trigger insturmnets with `i`-score statements. It can look like this:

```
i "osc" 0 10 0.5 220
```

The `i` is special syntax for `i`-statement. Then goes the list of arguments that are separated with spaces. The first argument is the instrument identifier. It's an integer number or string (note the mandatory double quotes). Then we can see two parameters that are hidden from the haskell user. It's delay to trigger the note and note duration. Both are in seconds. In the example we have a note with no delay that lasts for 10 seconds. Then we can see the arguments that our haskell-instrument takes in. They are amplitude value and frequency value.

The `InputMessage` code for our python code looks like this:

```
    def play(self, delay, duration, volume, frequency):
        self.thread.InputMessage("i \"%s\" %f %f %f %f" % ("osc", delay, duration, volume, frequency))
```

We use python string-formating syntax to substitute `f`'s with floats and `s`'s with strings. Note the escaped double quotes in the python code!

Example continued

Now we are ready to look at the python code:

```
import csnd6

class Audio:
    def __init__(self):
        engine = csnd6.Csound()
        engine.SetOption("-odac")
        engine.Compile("message.csd")

        thread = csnd6.CsoundPerformanceThread(engine)
        thread.Play()

        self.engine = engine
        self.thread = thread

    def play(self, delay, duration, volume, frequency):
        self.thread.InputMessage("i \"%s\" %f %f %f %f" % ("osc", delay, duration, volume, frequency))

    def close(self):
        self.thread.Stop()
        self.thread.Join()
```

The initialization and termination of audio engine are the same as in the previous example. The new funtion is `play`. The syntax is already explained. We take in dleay to trigger the note, note's duration and pair of our Haskell parameters (amplitude and frequency).

Let's try out our engine in the python interpreter:

```
$ python
> from message import Audio
> c = Audio()
> c.play(1, 3, 0.5, 220)
> c.play(0, 2, 0.3, 330)
> c.close()
```

Triggering instruments as procedures

Sometimes we don't want to produce the sound as the response to messages. Sometimes we want to update some parameters. You can imagine a drone sound going on or arpeggiator and we want to update a note or LFO rate with message. To do it we can use the function:

```
trigByName_ :: (Arg a) => String -> (a -> SE ()) -> SE ()
trigByName_ name instrument = aout
```

Note the underscore at the end. It creates a named procedure. The procedure can be called with Csound API just in the same way as an ordinary instrument. It's useful to know the `turnoff` function. It turns the instrument off. By default all Csound instrument last for some time. With `turnoff` we can simulate instant reaction procedure. It does some work (robably updates the global parameters) and then it turns itself off. The pattern of usage looks like this:

```
procedure args = do
    doSomeStuff
    turnoff

main = trigByName_ "update_param" procedure
```

Creation of MIDI-controlled instruments

If we want to create a VST plugin we want to be able to control our csound insturment in the MIDI-like manner. We want to send note on and note off messages. This functionality can be simulated with `trigByName_` function and global variables. There are predefined library function that already implement this behavior:

```
trigByNameMidi :: (Arg a, Sigs b) => String -> ((D, D, a) -> SE b) -> SE b
trigByNameMidi name instrument = ...
```

The instrument takes in two mandatory arguments: pitch and amplitude midi-keys. It produces an audio signal as output. We can use it with Csound API just as in previous examples. We have special format for Csound arguments to simulate note-on/off behavior:

```
i "givenName" delay duration 1 pitchKey volumeKey auxParams     -- note on
i "givenName" delay duration 0 pitchKey volumeKey auxParams     -- note off
```

Alongside with delay and duration we have another hidden argument. It's the fourth argument. It's `1` for note on and `0` for note off. Which note to turn off is determined by pitch key.

There is a procedure version of the function:

```
trigByNameMidi_ :: (Arg a, Sigs b) => String -> ((D, D, a) -> SE ()) -> SE ()
trigByNameMidi_ name instrument = ...
```

Monophonic MIDI-controlled instruments

The monophonic instruments need special treatment:

```
trigNamedMono :: D -> D -> String -> SE (Sig, Sig)
trigNamedMono portamentoTime releaseTime name = ...
```

The function is located at the module `Csound.Control.Midi` (see section Mono-midi synth).

The argument list for Csound is the same as for normal midi instruments.

```
i "givenName" 1 delay duration pitchKey volumeKey     -- note on
i "givenName" 0 delay duration pitchKey volumeKey     -- note off
```

MIDI-controlled patches

There are predefined midi-like named functions for patches (see section Csound API at the module `Csound.Air.Patch`):

```
patchByNameMidi :: (SigSpace a, Sigs a) => String -> Patch D a -> SE a
patchByNameMidi name patch = ...

monoPatchByNameMidi :: (SigSpace a, Sigs a) => String -> Patch Sig a -> SE a
monoPatchByNameMidi name patch = ...

monoSharpPatchByNameMidi :: (SigSpace a, Sigs a) => String -> Patch Sig a -> SE a
monoSharpPatchByNameMidi name patch = ...
```

If you are interested in non-trivial application that uses Csound API you can look at the python synthesizer called tiny-synth. It uses functions for named midi-controlled patches. It features 100+ patches from the standard collection of `csound-expression` instruments.

Example: Audio player

Let's create a command line audio player. We are going to create 3 instruments. One for playing wavs and aiffs, another one for playing mp3s and the last one to stop player.

```
-- the file Player.hs
module Main where

import Csound.Base

declick :: Sig2 -> Sig2
declick = mul (fades 0.01 0.1)

playWav :: Str -> SE Sig2
playWav file = return $ declick $ diskin2 file 1

playMp3 :: Str -> SE Sig2
playMp3 file = return $ declick $ mp3in file

stop :: Unit -> SE ()
stop _ = do
    turnoffByName "wav" 0 0.1
    turnoffByName "mp3" 0 0.1
    turnoff

main = writeCsd "player.csd" $ do
    wavs <- trigByName "wav" playWav
    mp3s <- trigByName "mp3" playMp3
    trigByName_ "stop" stop
    return $ wavs + mp3s
```

Let's take this file apart. The first thing we create is declicking envelope so that playback starts and fades without clicks:

```
declick :: Sig2 -> Sig2
declick = mul (fades 0.01 0.1)
```

Next we define an instruemnt to play wavs and aiffs:

```
playWav :: Str -> SE Sig2
playWav file = return $ declick $ diskin2 file 1
```

We define an instrument to play mp3s:

```
playMp3 :: Str -> SE Sig2
playMp3 file = return $ declick $ mp3in file
```

We define an instrument to turn off any notes for all instruments that play wavs and mp3s.

```
stop :: Unit -> SE ()
stop _ = do
    turnoffByName "wav" 0 0.1
    turnoffByName "mp3" 0 0.1
    turnoff
```

It uses the new function `turnoffByName`. The function is defined to turnoff named instruments. The first argument is the name of the instrument. The next is the code for turning off. Zero means turnoff all instances. The last argument is for release time (in seconds).

At the main function we assign names to instruments and direct the output to speakers.

```
main = writeCsd "player.csd" $ do
    wavs <- trigByName "wav" playWav
    mp3s <- trigByName "mp3" playMp3
    trigByName_ "stop" stop
    return $ wavs + mp3s
```

So we can create a file with audio engine and give it a name `player.csd` with command:

```
> runhaskell Player.hs
```

Let's look at the python code:

```
import csnd6, os.path, time

def is_mp3(filename):
    filename, file_extension = os.path.splitext(filename)
    return file_extension == '.mp3'

class Player:
    def __init__(self):
        engine = csnd6.Csound()
        engine.SetOption("-odac")
        engine.Compile("player.csd")

        thread = csnd6.CsoundPerformanceThread(engine)
        thread.Play()
        self.engine = engine
        self.thread = thread

    def play_file_by_ext(self, ext, file):
        self.thread.InputMessage("i \"%s\" 0 -1 \"%s\"" % (ext, file))

    def stop(self):
        self.thread.InputMessage("i \"stop\" 0 0.01")
        time.sleep(0.02)

    def play(self, file):
        self.stop()
        if is_mp3(file):
            self.play_file_by_ext("mp3", file)
        else:
            self.play_file_by_ext("wav", file)

    def close(self):
        self.thread.Stop()
        self.thread.Join()
```

The initialization and termination are the same as in previous examples. In the body of the instrument we use a trick to play note forever. To play note forever in the Csound we have to invoke it with negative duration. Look at the code for triggering the notes:

```
    def play_file_by_ext(self, ext, file):
        self.thread.InputMessage("i \"%s\" 0 -1 \"%s\"" % (ext, file))
```

Notice the duration of the note is set to `-1`. It's going to held the note forever. In the `play` function we stop all previous instances and then start new note. We determine the file type by extension:

```
    def play(self, file):
        self.stop()
        if is_mp3(file):
            self.play_file_by_ext("mp3", file)
        else:
            self.play_file_by_ext("wav", file)
```

Let's try it out in the terminal:

```
$ python
> from player import Player
> p = Player()
> p.play("muzzy.wav")
> p.stop()
> p.play("song.mp3")
> p.close()
```

- <= Arguments modulation
- => Creating VST-plugins with Cabbage
- Home

Creating plugins with Cabbage

(since version 5.1)

WARNING: Right now Cabbage is not stable on my environment (Ubuntu) so it's hard for me to test things out. It's hard to use. But when it will improve we are going to have the set of tools to create cabbage instruments.

The Cabbage is a cool program that let's you run Csound instruments and effects as VST-plugins. Moreover it let's you run Csound on Android. It defines it's own way to define UI-widgets in Csound. Cabbage is very easy to learn, the Haskell implementation faithfully represents the cabbage. So for better understending you should get the taste of cabbage on the official site. Do have a look at the docs or watch the video tutorials. But take the light view on Csound stuff since we have the haskell way of doing this just get acquinted with the cabage way of UI-declaration.

The native cabbage declaration is a list of widget declarations. Each declaration takes it's own line. The first goes the name of the widget and then on the same line we can write the properties of the widget:

```
<Cabbage>
form size(100, 100), pluginid("plugin")
button bounds(10, 10, 80, 80), channel("button-id"), text("Click Me"), colour:0(150, 30, 0), colour:1(30, 150, 12)
</Cabbage>
```

With markup language enclosed in `Cabbage` tag we define the UI. And in the audio engine code we can read the values from channels. We define the name of the channel with `channel` property.

To use the cabbage we need to import it separately. It's supposed to be imported qualified to aviod name-clashes with csound-functions:

```
import Csound.Base
import qualified Csound.Cabbage as C
```

The Haskell EDSL for Cabbage is inspired with `blaze-html` library. We represent the lists of widgets and properties with monads (scary word). It means that we can use the next line and identation as delimiter for widgets and properties:

```
import Csound.Base
import qualified Csound.Cabbage as C
```

```
ui = do
    C.cabbage $ do
        C.form $ do
            C.size 100 100
            C.pluginid "plugin"
        C.button $ do
            C.bounds 10 10 80 80
            C.channel "button"
            C.text1 "Click me"
            C.colour0 (C.Rgb 150 30 0)
            C.colour1 (C.Rgb 30 150 12)
    res <- chnCtrlGet "button"
    return res

main = dac $ do
    btn <- ui
    return $ btn * osc 220
```

So in Haskell the properties are delimited by indentation. There are some differences that are copuled with Haskell restrictions on names and type-system:

- Notice that in Haskell the function can not take variable number of arguments so we use `text1` for `text` with one argument and `text2` for text with to arguments.

- We can not use colon in the identifiers so `colour:0` becomes just `colour0`.

- The haskell has strict types. But in Cabbage there are two ways to represent colours. We can pass strings (web-hash codes) and we can pass triplets (RGB-values). To emulate this behaviour in Haskell there is a special type with two cases:

  ```
  data Col = Hash String | Rgb Int Int Int
  ```

We use the `chnCtrlGet` to get the control signal from the button with named channel.

If you are accustomed to Cabbage way of writing properties you can use the function `sequence_`:

```
C.form $ sequence_ [C.size 100 100, C.pluginid "plugin"]
```

What makes Haskell embedding really great is that it's not a spearate block of specific markdown. It's a code.

And we can abstract away the common blocks of code:

```
colors = do
    C.colour0 (C.Rgb 150 30 0)
    C.colour1 (C.Rgb 30 150 12)

C.cabbage $ do
```

```
    C.form $ do
        C.size 200 100
        C.pluginid "plugin"
    C.button $ do
        C.bounds 10 10 80 80
        C.channel "button1"
        C.text1 "Hi"
        colors
    C.button $ do
        C.bounds 110 10 80 80
        C.channel "button2"
        C.text1 "Bye"
        colors
```

We can write functions to avoid duplication:

```
colors = do
    C.colour0 (C.Rgb 150 30 0)
    C.colour1 (C.Rgb 30 150 12)

mkButton name id (x, y) = C.button $ do
        C.bounds x y 80 80
        C.channel id
        C.text1 name
        colors

C.cabbage $ do
    C.form $ do
        C.size 200 100
        C.pluginid "plugin"
    mkButton "Hi"  "button1" (10, 10)
    mkButton "Bye" "button2" (110, 10)
```

We can share the variables between audio-engine and markup. It can be useful to store the names for channels:

```
ui = do
    C.cabbage $ do
        C.form $ do
            C.size 200 100
            C.pluginid "plugin"
        mkButton "Hi"  btn1 (10, 10)
        mkButton "Bye" btn2 (110, 10)
    b1 <- chnCtrlGet btn1
    b2 <- chnCtrlGet btn2
    return (b1, b2)
    where
```

```
        btn1 = "button1"
        btn2 = "button2"
```

After the csound file is rendered we can load it to cabbage and then use it as VST or AU plugin or load it to the Cabbage App on android. We can find out how to do it o the official web-site.

- <= Csound API. Using generated code with another languages
- => Happy Haskelling / Csounding
- Home

Introduction to Csound for Haskell users

We are going to make electronic music. But what is Csound? And why should we use it? Csound is a domain specific programming language. It helps you to define synthesizers and make some music with them. Csound was born in 1985 (a bit older than Haskell) at MIT by Barry Vercoe. It's widely used in the academia. It has a long history. So with Csound we get a lot of music DSP-algorithms ready to be used. It's written in C. So it's very efficient. It's driven by text, so we can generate it. Csound's community is very friendly (what a coincidence!). Csound is very well documented.

We don't need to know Csound to use this library but it's helpful to know the main features of the Csound. How can we create music with Csound in general What design choices were made, basic features and quirks. Csound belongs to the MUSIC N family of programming languages. What does it mean? It means that description of the music is divided in two parts:

- Orchestra. User defines instruments
- Scores. User triggers instruments with a list of notes

An instrument is something that listens to notes and converts them to signals. Note is a tuple: (instrument name, start time, duration, parameters). Parameters cell is a tuple of primitive types: numbers (`D`), strings (`Str`) and tables or arrays of numbers (`Tab`).

Instruments

An instrument is represented with function that takes a tuple of primitive values (`Arg`) and converts it to the tuple of signals (`Sigs`) wrapped in the type `SE`:

```
(Arg a, Sigs b) => a -> SE b
```

The `SE` means Side Effect. It's like `IO`-monad but for Csound.

Events

With instruments we can convert the bunch of notes to the plain signals. There are several ways to do it. We can trigger an instrument:

- With score
- With event stream
- With midi-device

The Score

The score is a list of events with some predefined total duration. An event is a triple that contains:

```
(t0, dt, args)
```

Where `t0` is a start time, `dt` is a duration of the event, `args` is a list of arguments for the instrument. The Score is represented with the type:

```
data CsdEventList a = CsdEventList
    { csdEventListDur       :: Double
    , csdEventListNotes     :: CsdEvent a }

type CsdEvent a = (Double, Double, a)
```

The start time and duration are in seconds. To invoke an instrument with Score we can use the functions:

```
sco :: (CsdSco f, Arg a, Sigs b) => (a -> SE b)  -> f a -> f (Mix b)
mix :: (CsdSco f, Sigs a) => f (Mix a) -> a
```

The type `CsdEventList` is not to be used directly. It's a canonical representation of the Csound score. We should use something more higher level. That's why we don't see it in the signatures. It's referenced indirectly with type class `CsdSco`. The types of the type class `CsdSco` are things that can be converted to the canonical representation.

```
class CsdSco f where
    toCsdEventList :: f a -> CsdEventList a
    singleCsdEvent :: CsdEvent a -> f a
```

The method `toCsdEventList` converts a given score representation to the canonical one. The method `singleCsdEvent` constructs a scores that contains only one event. it lasts for one second.

The function `sco` applies an instrument to the score and produces the score of signals. Then we can apply the function mix` to get the mixed signal.

Scores are very simple yet powerful. Csound handles polyphony for us. If we trigger several notes at the same time on the same instrument we get three instances of the same instrument running in parallel. It's very cool feature (not so easy thing to do with Pd).

The event stream

An event stream is something that produces the notes. The score contains the predefined notes but event stream can produce the in real time.

The event stream is represented with the type:

```
newtype Evt a = Evt { runEvt :: Boom a -> SE () }

type Boom a = a -> SE ()
```

An event stream is a function that takes a procedure of type `a -> SE ()` and applies it to all events in the stream.

We have some primitive constructors:

```
metroE :: Sig -> Evt ()
```

It takes a frequency of the repetition. An empty tuple happens every now and then. We can process the events with functions:

```
repeatE :: a -> Evt b -> Evt a
filterE :: (a -> BoolD) -> Evt a -> Evt a
cycleE  :: Arg a => [a] -> Evt b -> Evt a
oneOf   :: Arg a => [a] -> Evt b -> Evt a
...
```

For example, we can substitute all the events with the constant value (`repeatE`), filter an event stream with predicate or repeat elements in the list (`cycleE`) or take elements at random (`oneOf`). There are many more functions.

The `BoolD` is a Csound boolean value. It's instance of the type classes from the package `Boolean`. There is another boolean type `BoolSig` for the signals of boolean values.

We can trigger instruments on the event streams with functions:

```
trig  :: (Arg a, Sigs b) => (a -> SE b) -> Evt (D, D, a) -> b
sched :: (Arg a, sigs b) => (a -> SE b) -> Evt (D, a)    -> b
```

The function `trig` applies an instrument to the event stream of notes. A note contains a delay of the event, the event duration and the arguments for the instrument. The function `sched` is the same as `trig` but all events happen immediately.

The Midi devices

We can trigger an instrument with midi devices:

```
midi   :: (Sigs a) => (Msg -> SE a) -> SE a
midin  :: (Sigs a) => Int -> (Msg -> SE a) -> SE a
pgmidi :: (Sigs a) => Int -> Maybe Int -> (Msg -> SE a) -> SE a
```

The function `midi` starts to listen for the midi-messages (`Msg`) on all channels. With function `midin` we can specify the concrete channel (it's an integer from 1 to 16). The function `pgmidi` is for assigning an instrument to the midi-program (the first argument) and possible channel (the second argument).

We can query midi-messages for amplitude, frequency and other parameters (we can see the complete list in the module `Csound.Opcode.RealtimeMIDI`):

```
cpsmidi :: Msg -> D
ampmidi :: Msg -> D -> D
```

```
...
```

The second argument of `ampmidi` is a scaling factor or maximum value for amplitude.

Flags and options

Music is defined in two parts. They are Orchestra and Scores. But there is a third one. It's used to set the global settings like sample rate or control rate values (block size). In this library you can set the initial values with `Csound.Options`.

Features and quirks

Audio and control rates

Csound has made a revolution in electronic music technology. It introduced two types of signals. They are audio rate and control rate signals. The audio rate signals is what we hear and control rate signals is what changes the parameters of sound. Control rate is smaller then audio rate. It speeds up performance dramatically. Let's look at one of the sound units (they are called opcodes)

```
ares buthp asig, kfreq [, iskip]
```

It's a Butterworth high pass filter as it defined in the Csound. a-sig - means sig at audio rate. k-freq means freq at control rate (for historical reasons it is k not c). iskip means skip at i-rate. i-rate means init time rate. It is when an instruments instance is initialized to play a note. i-rate values stays the same for the whole note. So we can see that signal is filtered at audio rate but the center frequency of the filter changes at the control rate. In this library the types are merged together (`Sig`). If you plug a signal into `kfreq` we can infer that you want this signal to be control rate. In Csound some opcodes exist go in pairs. One that produces audio signals and one that produces control rate signals. By default if there is no constraint for the signal it is rendered at the audio rate except for those units that produce sound envelopes (like `linseg` or `expseg`).

You can change this behaviour with functions `ar` and 'kr'. They set the signal-like things to audio or control rate. For instance if you want your envelope to run at control rate, write:

```
env = ar $ linseg [0, idur/2, 1, idur/2, 0]
```

Table size

For speed table size should be the power of two or power of two plus one (all tables for oscillators). In this library you can specify the relative size (see `Csound.Options`). I've tried to hide the size definition to make sings easier.

How to read the Csound docs

You'd better get acquainted with Csound docs. Docs are very good. How to read them? For instance you want to use an oscillator with cubic interpolation. So you dig into the `Csound.Opcode.SignalGenerators` and find the function:

```
oscil3 :: Sig -> Sig -> Tab -> Sig
```

From Hackage we can guess that it takes two signals and table and returns a signal. It's a clue but a vogue one. Let's read along, in the docs you can see a short description (taken from Csound docs):

```
oscil3 reads table ifn sequentially and repeatedly at a frequency xcps.
The amplitude is scaled by xamp. Cubic interpolation is applied
for table look up from internal phase values.
```

and here is the Csound types (the most useful part of it)

```
> ares oscil3 xamp, xcps, ifn [, iphs]
> kres oscil3 kamp, kcps, ifn [, iphs]
```

We see a two versions of the opcode. For audio and control rate signals. By default first is rendered if we don't plug it in something that expects control rates. It's all about rates, but what can we find out about the arguments?

First letter signifies the type of the argument and the rest is the name. We can see that first signal is amp with x rate. and the second one is cps with x rate. We can guess that amp is the amplitude and cps is cycles per second. This unit reads the table with given amplitude (it is a signal) and frequency (it is a signal too). Or we can just read about it in the docs if we follow the link that comes at the very last line in the comments:

```
doc: <http://www.csounds.com/manual/html/oscil3.html>
```

We now about a-, k- and i-rates. But what is the x-rate? Is it about X-files or something? X means a-rate or k-rate. You can use both of them for this argument. Let's go through all types that you can find:

- `asig` -- audio rate (`Sig`)
- `ksig` -- control rate (`Sig`)
- `xsig` -- audio or control rate (`Sig`)
- `inum` -- constant number (`D`)
- `ifn` -- table or 1D-array (`Tab`). They are called functional tables in Csound.
- Sfile -- string, probably a file name (`Str`)
- fsrc -- spectrum (`Spec`). Yes, you can mess with sound in the space domain.

Often you will see the auxiliary arguments, user can skip them in Csound. So we can do it in Haskell too. But what if we want to supply them? We can use the function `withInits` for this purpose.

Example (a concert A)

```
module Main where

-- imports everything
import Csound.Base

-- Let's define a simple sound unit that
-- reads in cycles the table that contains a single sine partial.
-- oscil1 is the standard oscillator with linear interpolation.
-- 1 - means the amplitude, cps - is cycles per second and the last argument
-- is the table that we want to read.
myOsc :: Sig -> Sig
myOsc cps = oscili 1 cps (sines [1])

-- Let's define a simple instrument that plays a sound on the specified frequency.
-- We use sig to convert a constant value to signal and then plug it in the osc unit.
-- We make it a bit quieter by multiplying with 0.5.
pureTone :: D -> SE Sig
pureTone cps = return $ 0.5 * (myOsc $ sig cps)

-- Let's trigger the instrument from the score section.
-- It plays a three notes. One starts at 0 and lasts for one second with frequency of 440,
-- another one starts at 1 second and lasts for 2 seconds, and the last note lasts for 2 seconds
-- at the frequency 220 Hz.
res = sco pureTone $ CsdEventList 5 [(0, 1, 440), (1, 2, 330), (3, 2, 220)]

-- Renders generated csd-file to the "tmp.csd", invokes the csound on it
-- and directs the sound to speakers.
main :: IO ()
main = dac $ mix res
```

More examples

You can find many examples at:

- Examples in the archive with the source code of the library.
- A translation of the Amsterdam catalog of Csound computer instruments

References

Got interested in Csound? Csound is very well documented. There are good tutorials, read about it at:

- Reference manual --
- Floss tutorials --
- Amsterdam catalog of Csound computer instruments --
- Lots of wonderful real-time examples by Iain McCurdy --
- Outdated but short manual on Csound

- Home

Overview of the library

There are many functions in the library. Let's list the most useful ones:

The converters:

```
double :: Double -> D
int    :: Int    -> D
text   :: String -> Str
sig    :: D -> Sig
```

The primitive wave forms

They convert the time varied frequency to the signal:

```
osc :: Sig -> Sig       -- pure tone
saw :: Sig -> Sig       -- sawtooth
sqr :: Sig -> Sig       -- square wave
tri :: Sig -> Sig       -- triangle wave

oscBy :: Tab -> Sig -> Sig     -- oscillator with a specified wave
blosc :: Tab -> Sig -> Sig     -- a generic band-limited oscillator
```

There are unipolar variants of the waveforms (a unipolar signal varies from 0 to 1): `uosc`, `usaw`, `usqr`, etc

Envelopes:

Release is a time to linger the signal after note is over (useful with midi). Two additional parameters are: time of the release and the final value. Exponential envelopes should be above zero (we can use the small numbers to imitate the zero)

```
linseg :: [D] -> Sig                -- linear envelope
expseg :: [D] -> Sig                -- exponential envelope

                                    -- with release:
linsegr :: [D] -> D -> D -> Sig     -- linear envelope
expsegr :: [D] -> D -> D -> Sig     -- exponential envelope
```

Filters

Parameters: the last argument is always the signal to filter the first parameter is cut-off frequency for `lp` and `hp` and the center frequency for `bp` and `br`. The second argument is a band-width (resonance).

```
lp  :: Sig -> Sig -> Sig -> Sig     -- low pass
hp  :: Sig -> Sig -> Sig -> Sig     -- high pass
bp  :: Sig -> Sig -> Sig -> Sig     -- band pass
br  :: Sig -> Sig -> Sig -> Sig     -- band reject
```

There are Butterworth variants of the filters: `blp`, `bhp`, `bbp`, `bbr`.

Reverberation

```
nreverb  :: Sig -> Sig -> Sig
```

It takes a signal to process, the delay time and the speed of decay (0 to 1).

```
reverbsc :: Tuple a => Sig -> Sig -> Sig -> Sig -> a
reverbsc aleft aright feedBackLevel cutOffFrequency
```

It's a stereo processing. It takes a two signals the feedback level (0 to 1) and cut off frequency of the low pass filter (usually it's 10000).

Reading the files

```
readSnd :: String -> (Sig, Sig)  -- read once
loopSnd :: String -> (Sig, Sig)  -- read in loop
loopSndBy :: D -> String -> (Sig, Sig) -- read in loop with given period (in seconds)
```

If we have a wav (or aiff) file we can read it with the given speed (the first argument):

```
readWav :: Sig -> String -> (Sig, Sig)
loopWav :: Sig -> String -> (Sig, Sig)
```

Note that speed is a Signal and can vary with time. Use negative values to read in reverse. When speed equals one it's normal reading.

Low-level functions:

```
mp3in   :: Tuple a => Str -> a
diskin2 :: Tuple a => Str -> Sig -> a
```

The function `diskin2` reads only wav-files. The additional parameter is the speed of the playback.

Constructing the arrays

```
sines :: [Double] -> Tab    -- list of the partials to sine harmonics
lins  :: [Double] -> Tab    -- array of linear segments
                            -- (parameters are like in `linseg`)
exps  :: [Double] -> Tab    -- array of exponential segments
                            -- (parameters are like in `expseg`)
```

Noises

The first parameter is the amplitude, the second one is beta for the low pass filter (-1 to 1) for the function `noise`, and the frequency of the random values for the function `randi`.

```
noise   :: Sig -> Sig -> SE Sig     -- white noise
randi   :: Sig -> Sig -> SE Sig     -- random linear segments
pinkish :: Sig -> SE Sig            -- pink noise
```

Simplified noises:

```
white, pink :: SE Sig
```

Events

Defined in the module `Csound.Control.Evt`

The event stream `Evt` is a `Functor` and `Monoid`

```
metroE   :: Sig -> Evt ()
filterE  :: (a -> BoolD) -> Evt a -> Evt a
repeatE  :: a -> Evt b -> Evt a
cycleE   :: (Arg a) => [a] -> Evt b -> Evt a
oneOf    :: (Arg a) => [a] -> Evt b -> Evt a
randSkip :: D -> Evt a -> Evt a
```

Invoking the instruments

```
-- renderes the midi instrument
midi    :: Sigs a => (Msg -> SE a) -> SE a

-- renderes the midi instrument
-- on the given channel
```

```
midin   :: Sigs a => Int -> (Msg -> SE a) -> SE a

-- mix the signals from score
mix     :: (CsdSco f, Sigs a) => f (Mix a) -> a

-- invokes an instrument on the score
sco     :: (CsdSco f, Arg a, Sigs b) => (a -> SE b) -> f a -> f (Mix b)

-- applies an effect to the score
eff     :: (CsdSco f, Sigs a, Sigs b) => (a -> SE b) -> f (Mix a) -> f (Mix b)

-- invokes an instrument on the event stream

-- event stream contains duration of the note
sched   :: (Arg a, Sigs b) => (a -> SE b) -> Evt (D, a) -> b

-- event stream contains delay time and duration of the note
trig    :: (Arg a, Sigs b) => (a -> SE b) -> Evt (D, D, a) -> b

-- triggers an instrument with the first event
-- stream and holds the note while the second event stream is silent
schedUntil :: (Arg a, Sigs b) => (a -> SE b) -> Evt a -> Evt c -> b
```

Truncating/repeating the signal

```
takeSnd   :: Sigs a => Double -> a -> a
repeatSnd :: Sigs a => D      -> a -> a
```

With `takeSnd` we can truncate the signals to the given amount of seconds. We use only first `n`-seconds from the signals. The `repeatSnd` repeats the signal with the given period.

Rendering the Csound files

Defined in the module `Csound.IO`

The type class `RenderCsd` contains the sings that can be rendered to file. It's something that produces the sound or triggers the Csound procedures.

```
-- plays a signal in real time
dac     :: RenderCsd a => a -> IO ()

-- plays a signal in real time with virtual midi-keyboard
vdac     :: RenderCsd a => a -> IO ()
```

```
-- saves the csound file with the given name
writeCsd :: RenderCsd a => String -> a -> IO ()

-- saves the csound file to 'tmp.csd' and invokes the csound on it
csd :: RenderCsd a => a -> IO ()
```

Options

Defined in the module `Csound.Options`.

With options we can set the global settings of the rendering process. The type `Options` is a monoid with the meaning that we can concatenate partially defined options and get more specified ones.

To specify the options we use the rendering functions with suffix `By`:

```
options = mconcat [setRates 44800 64, setDac, setAdc]

main = csdBy options asignal
```

The most common options:

```
-- sets the sample rate and the block size
setRates :: Int -> Int -> Options

-- sets the buffer sizes (the define the granularity of the real-time performance)
setBufs :: Int -> Int -> Options

-- where to direct the output
setOutput :: String -> Options

-- from where to recieve the input
setInput :: String -> Options

-- sets the specific output and input

-- directs the output to dac
setDac :: Options

-- recieves the input from adc.
setAdc :: Options

setThru = mappend setDac setAdc
```

Score

Defined in the module `Temporal.Music.Score` from the package `temporal-music-notation`:

```
-- Constructs a score with the single note (it lasts for one second)
temp :: a -> Score a

-- Constructs a score that contains nothing and lasts for some time.
rest :: Double -> Score a

-- Stretches the score in the time domain with the given coefficient.
-- It gets faster or slower.
str  :: Double -> Score a -> Score a

-- Delays all events with the given amount of time.
del  :: Double -> Score a -> Score a

-- A sequential composition of scores. It's short for melody.
-- It plays the scores one after the other.
mel  :: [Score a] -> Score a

-- A parallel composition. It's short for harmony.
-- It plays all scores at the same time.
har  :: [Score a] -> Score a

-- Repeats the score several times.
loop :: Int -> Score a -> Score a
```

GUI

Main elements:

```
--         Label     Diapason   Init      Result
--                   of the     value
--                   value

knob   :: String -> ValSpan -> Double -> Source Sig
slider :: String -> ValSpan -> Double -> Source Sig

button :: String -> Source (Evt Unit)
toggle :: String -> Source (Evt D)

--                      Label     Alternatives     Id of the
--                                                 default
--                                                   value
radioButton :: Arg a => String -> [(String, a)] -> Int -> Source (Evt a)

-- shows a static text
box    :: String -> Display
```

Creating value spans:

Linear and exponential spans with the give bounds:

```
linSpan :: Double -> Double -> ValSpan
expSpan :: Double -> Double -> ValSpan
```

The linear unit span:

```
uspan :: ValSpan
uspan = linSpan 0 1
```

Layout

```
-- horizontal placement
hor :: [Gui] -> Gui

-- vertical placement
ver :: [Gui] -> Gui

-- scaling of the element within the group
-- (element is contained in the horizontal
-- or vertical container)
sca :: Double -> Gui -> Gui
```

Creating a windows with GUIs

Creates a single window

```
panel :: Gui -> SE ()
```

Creates a single window that is listening for keyboard events

```
keyPanel :: Gui -> SE ()
```

Creates a single windows and we can specify title and size of the window:

```
panelBy :: String -> Maybe Rect -> Gui -> SE ()
```

Keyboard events

```
data KeyEvt = Press Key | Release Key
data Key = CharKey | F1 | F2 | ...

keyIn   :: KeyEvt -> Evt Unit
```

Press and release a simple key:

```
charOn  :: Char   -> Evt Unit
charOff :: Char   -> Evt Unit
```

We should create a window to be able to listen on keyboard events with the function `keyPanel` or `keyPanelBy`.

- Home