Minima:
A Web Audio Playground for Minimalist Music

Brian Ginsburg

Functional Languages,
Portland State University

June 13th, 2017

1 Introduction

Minima is simple web application for experimenting with minimalist music patterns. Minima is written in Elm and JavaScript.

Elm is a purely functional language that targets front-end web development. Elm features functional techniques such as maps, filters, higher-order functions, and abstract data types. These techniques drastically reduce the verbosity of JavaScript and can make code compact and easy to read. In addition, the type safety of Elm eliminates runtime errors and makes front-end web development sensible and enjoyable.

Minima uses a simple web audio synthesizer written in JavaScript. The Elm application treats the synthesizer as "hardware" and sends it note messages through ports provided in the Elm architecture.

The remainder of this paper will describe Minima, introduce the Elm language and architecture, and review the major components of application: the model, view, and update. Through each section, I will highlight functional language concepts and techniques used in this project. A testing section will validate that Minima plays the music we might expect. Finally, I will consider possible improvements to the application, future work, and make a few concluding remarks.

Source code fragments are included throughout this report, but you may find some function definitions are missing. If you would like to reference a missing definition, the source code is listed in Appendix A. In addition, I invite any and all questions on the application.

2 Minima

Minima has four voices, each represented by a row of note patterns. Each voice plays a single pitch, and the voices are arranged vertically by frequency[1]. This arrangement borrows from the typical layout of MIDI tracks in a digital audio workstation. Figure 1 shows the Minima GUI.

The arrows to the side of each row rotate forward and backward through a set of note patterns. The play button, pause button, and a counter indicating the current beat are embedded in the instructions. When the music is playing, the counter updates with the clock, and when the music is stopped the counter reads 0.

Minima is live on the web. Try it out if you would like a demo[2].

[1]The pitches are justly tuned intervals off the second row at 440Hz. From the lowest row: 7/8, 1/1, 3/2, 2/1. The second, third, and fourth rows are tonic, fifth, and octave. The lowest row is a pitch between a major sixth and a minor seventh, about 31.2 cents below a standard minor seventh. If you are interested in learning more about Just Intonation, the Wikipedia page[1] is good, or ask me personally.

[2]Try Minima at http://brianginsburg.com/minima.

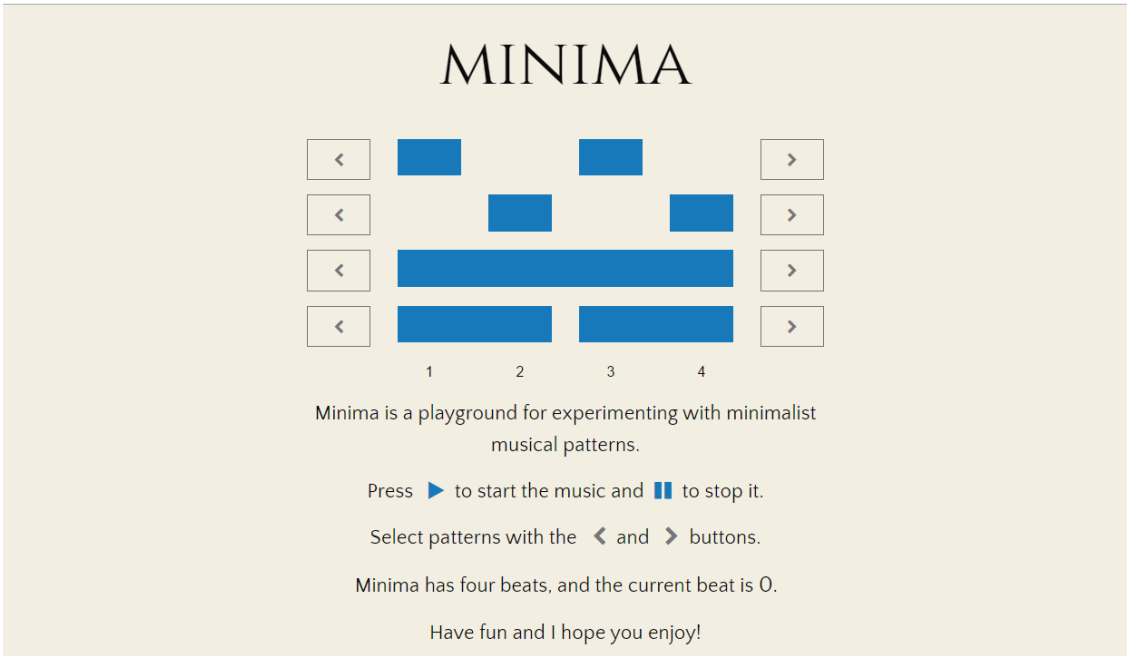

Fig. 1. Minima

3 Elm Language and Architecture

Elm is a purely functional language for building front-end web applications. The language was developed by Evan Czaplicki while working on his thesis *Elm: Concurrent FRP for Functional GUIs*[2]. The Elm compiler produces HTML, CSS, and JavaScript, though it mostly compiles to JavaScript and virtualizes HTML and CSS.

Elm uses a virtual DOM to handle changes to the structure and style of a web page[3]. A virtual DOM is an abstract version of a web page. Elm listens for user interactions and creates a new virtual DOM when changes occur. This new DOM is diffed with the previous DOM, and the changes that need to be made are rendered in the actual DOM in one batch operation. Since Elm is a purely functional language with immutable data, diffing can be handled lazily making updates to the page quite fast.

The architecture of an Elm application consists of a model, update, and view[4]. The model is the state of your application, update is how you modify state, and the view is a representation of your model. Elm uses an event loop that listens for changes from the user and uses an `update` function to modify the model. The runtime then modifies the view based on changes to the model.

When a user interacts with the application, the Elm runtime produces a message that is sent to `update`. Commands are actions generated from `update` and will do things that communicate with the outside world. Subscriptions listen to the outside world for events and tell `update` about them through messages. Together, commands and subscriptions manage all explicit interactions between the pure language and the outside world.

Note that the virtual DOM, commands and subscriptions are Elm's way of dealing with the external world of side effects. Elm is purely functional, and much of the uncertainty of the web is handled in the Elm runtime.

An Elm `Program` brings these pieces is set in the `main` function.

```
main =
    Html.program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }
```

The `init` field assigns our initial model, and the other fields tell the `Program` where to look to for the other building blocks of our application.

The syntax of Elm is similar to Haskell, but with a few notable differences[5].

- Type signatures in Elm are written with `:` and cons is written `::`
- Function application is written `|>` and `<|`, where `<|` is the same as $ in Haskell and `|>` goes the other direction. Most often you will see `|>` in Elm.
- The `!` operator makes a tuple from its left and right arguments in the `update` function[3]
- Elm uses `Maybe` in cases where a list operation might fail. You have to account for the `Just a` and `Nothing` cases.
- Elm uses `toString` instead of `show`
- Elm uses `type alias` instead of `type`, and `type` instead of `data`
- Elm does not allow multiple body declarations for functions, `case of` statements are used instead

Many other differences exist, but these should be enough to make the upcoming code examples clear to a Haskell programmer.

The next few sections of this paper will cover the model, view, and update in Minima. Subscriptions and ports for JavaScript interoperation will be briefly discussed in the Update section.

[3]A special case with the type `(!) : model -> List (Cmd msg) -> (model, Cmd msg)`. This will be discussed more in the Update section.

4 Model

We will cover the model first since it defines the types that will be used throughout the application.

A `Model` in Elm is defined as an alias to a record. In Minima, the model consists of a score, voices, ticks, and a clock.

```
type alias Model =
    { score : Score
    , ticks : Int
    , clock : Int
    , one : Voice
    , two : Voice
    , three : Voice
    , four : Voice
    }
```

Minima plays notes from the `score`. The `clock` tracks the current beat up to `ticks` before returning to 1. On each tick of the `clock`, Minima plays all notes in the `score` with a matching value for `tick`. Each note has a frequency, duration, and tick[4].

```
type alias Score =
    List Note

type alias Note =
    { frequency : Float
    , duration : Int
    , tick : Int
    }
```

A `Voice` has a `pattern` of notes and a `frequency`. A `Pattern` is a list of actions, which are read into the `score` as notes or dropped as rests. The option is represented with an algebraic data type `Action`.

[4]Duration is a relative value. Minima defaults to 60bpm, making a duration of 1 equal one second, but a rate controller may be added to a future version of the application. Frequency is the frequency of the soundwave, not the how often a note is played.

```
type alias Voice =
    { id : String
    , frequency : Float
    , pattern : Pattern
    }

type alias Pattern =
    List Action

type Action
    = Play Int
    | Rest Int
```

In my initial version of Minima, I defined `Pattern` as an abstract data type that enumerated all possible patterns as `Whole`, `HalfHalf`, and `HalfRest`, encoding the duration of notes and rests. This `Pattern` turned out to be cumbersome because it led to hard-coding for each option throughout the application. On some good advice, I changed my implementation to the `Pattern` described above.

5 View

The `view` is the representation of the model that the user actually sees. In other words, the HTML and CSS that are rendered to the screen by Elm.

```
view : Model -> Html Msg
view model =
    div []
        [ title
        , showRow model.four
        , showRow model.three
        , showRow model.two
        , showRow model.one
        , showCounter model.ticks
        , instructions model.clock
        ]
```

The view takes a `Model` and returns an `Html Msg`. Note that `Html` is a type constructor with a parameter for the type returned from user interactions with the web page[5]. The parameter in this case is a `Msg` that will be picked up by the update function. User interactions in Minima are click events on the play button, pause button and pattern selection buttons.

[5] `Html` can take other types or type variables[6]. It acts like a `List` or `Maybe`, or perhaps a better comparison would be to the `IO` in Haskell since since it represents a computation for a web browser.

The `view` function defines an Html expression where `div` that takes a list of attributes and a list of Html messages.

```
div : List (Attribute msg) -> List (Html msg) -> Html msg
```

By defining nodes such as `div` this way, Elm builds up the DOM as a tree with a variable number of branches per node[6].

Each node within Minima's `view` function represents a row in the user interface. The rows are displayed from top to bottom, with voices arranged by frequency from highest frequency down.

We will focus on the `showRow` expression since it is the most interesting part of the view. Each row displays a pattern and rotation buttons.

```
showRow : Model.Voice -> Html Msg
showRow voice =
    case .pattern voice of
        [] ->
            div [ class "row align-center" ] []

        acts ->
            div [ class "row align-center" ]
                (rotateVoice voice Model.Left
                    :: List.map renderAction acts
                    ++ [ rotateVoice voice Model.Right ]
                )
```

The `showRow` function takes a `Voice` and returns an `Html Msg`. It uses a `case of` expression to pattern match on the `pattern` field in `Voice`. The `.pattern` expression is a built-in function provided for record types in Elm that gets the value for the matching field.

Each row has a `class` attribute[7] and a list of Html nodes built out by mapping `renderAction` over the actions in the pattern. The left rotate button is prepended with cons and the right rotate button is concatenated at the end. A case for an empty pattern is defined, but this case is not used in practice. Each voice should have some pattern.

Blocks for each action are produced with the `renderAction` function. A block will be filled in for a `Play` action or left empty for a `Rest` action. The `Int` duration of an action defines the width of the block.

[6]This tree is somewhat like a Rose tree, since the number of branches from an Html element is often unbounded.

[7]The classes `row` and `align-center` reference CSS syles imported from the Foundation UI framework.

```
renderAction : Model.Action -> Html Msg
renderAction action =
    let
        block duration text =
            div [ class ("column small-" ++ toString duration) ] text
        space =
            [ text "\x2002" ] -- comment: unicode space
    in
        case action of
            Model.Play duration ->
                block duration [ a [ class "expanded button" ] space ]

            Model.Rest duration ->
                block duration space
```

The `renderAction` function went through a couple of iterations, using various local `let` definitions to make the code readable and remove repeated Html syntax. The final version defines a `block` that takes the `duration` parameter and `text` for the content in the block. An anchor tag fills the `Play` action, and a `space` is the `text` for either `Play` or `Rest`.[8]

The output of `showRow` and `renderAction` is a nice grid of blocks representing the musical pattern that Minima will play.

6 Update

The update function is where the state of model gets modified. It takes a `Msg` and `Model`, and returns an updated `model` and a command.

```
update : Msg -> Model -> ( Model, Cmd Msg )
```

In Minima, the `update` function subscribes to `Time` events that drive its clock and reacts to `onClick` events from the user to start the music, stop the music, and rotate patterns.

The messages accepted by `update` are defined with an abstract data type.

```
type Msg
    = Tick Time
    | Play
    | Pause
    | Rotate Voice Direction
```

[8]The unicode space is a hack to force the height of each block to a uniform size matching the rotate buttons. There may be a better way to address this in CSS.

Each `Msg` is a case in the `update` function. The first three modify the state of the clock.

```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        Tick time ->
            { model
                | clock = increment model.clock model.ticks
            }
                ! [ Cmd.batch (playNotes model.score model.clock) ]

        Play ->
            { model | clock = 1 } ! []

        Pause ->
            { model | clock = 0 } ! []
        ...
```

The { `model | ...` } statement updates the model. The `!` operator makes a tuple from its left and right arguments.

```
(!) : model -> List (Cmd msg) -> (model, Cmd msg)
(!) model commands =
    (model, batch commands)
```

Note that `model` and `msg` are type variables. The `!` operator could take any `model` and `msg` that can be given as a parameter to `Cmd`. Also, the `!` operator packages the list of commands into the primitive `batch`.

```
batch : List (Cmd msg) -> Cmd msg
batch =
    Native.Platform.batch
```

The tuple produced by `!` is returned by `update`[9].

The `Rotate` message modifies the state of the voices. We will take this message up in the Pattern Selection section.

6.1 Clock

In the first version of Minima, I implemented a single voice that played each successive note based on the duration of the note before it. In other words, there was no clock. Minima just played one note and then the next. While this worked out fine for a single voice, it did not work as easily when more voices were introduced.

[9]Multiple commands can also be batched before `!` evaluates. We use this to play multiple notes at the same time.

The challenge is that when you overlap notes, you cannot measure the start time for a note based on the duration of the previous note. You only have a sense of when the first note will end. A clockless implementation is attractive, because it would allow patterns of arbitrary length to be overlapped, which could generate interesting syncopation and polymeters. I was unable to find a way to make it work in this project, and I implemented a clock[10].

The `clock` is initially set to 0. When a user presses the play button, the `clock` is set to 1 and starts to increment. The `playNotes` function retrieves a list of notes for the current `clock` value and plays them. When a user presses the pause button, the `clock` is set back to 0 and is the music is stopped.

```
increment : Int -> Int -> Int
increment clock ticks =
    case clock of
        0 ->
            0


        _ ->
            (clock % ticks) + 1
```

To drive the clock, we rely on an external time event. The `Time` library[7] provides an `every` function that repeatedly generates a time message at some time interval. Since we are listening for an external event not generated by the user, this function is added to the `subscriptions`. We send a `Tick` message to the update function every second.

```
subscriptions : Model -> Sub Msg
subscriptions model =
    every second Tick
```

Taken together, the increment function and time subscription move our clock along. The update function receives the `Tick` messages and executes `increment` on each tick. `Play` starts the music at 1 and `Pause` stops the music. The `Play` and `Pause` messages come from click events and are defined in the view.

Now that our clock is running, we are ready to use `playNotes` to play some music! We will pass `playNotes` the `score` and the `clock` from the model to generate our note events.

[10] I based my clock off of the clock in the elmkc-melodymaker[8] project. My implementation is a bit different since I do not couple my timing to note sustain.

6.2 Playing Notes

Minima plays the notes in the `score` for the current tick by sending note events to a JavaScript web audio synthesizer.

Recall that `Score` is an alias for `List Note`. The `playNotes` function filters this list for any notes with a `tick` field that matches the value of `clock`. Then the `play` function is mapped across the remaining notes producing a list of commands for the Elm runtime.

```
playNotes : Score -> Int -> List (Cmd msg)
playNotes score clock =
    List.filter (\n -> .tick n == clock) score
        |> List.map play
```

The predicate we pass our filter is constructed with a lambda expression. The anonymous definition for this predicate works since we are making a simple comparison. If our comparison was more complex, we could use a local definition or an entirely separate function. The `|>` applies `List.map play` to the list produced by `List.Filter`[11].

Elm applications interoperate with JavaScript through ports[9], a message passing system that manages the effects of interacting with JavaScript[12]. Since JavaScript does not provide the guarantees about type safety that Elm expects, Elm manages side effects through the ports interface. Ports are like any other `Cmd` and `Sub`, they just provide both for two-way communication. Minima only uses `Cmd`.

Minima sends notes through the port `play` to the web audio synthesizer[13].

```
port play : Note -> Cmd msg
```

On the JavaScript side, we instantiate our Elm application and set up a function to receive the messages.

```
"use strict"
var app = Elm.Main.fullscreen();
...
app.ports.play.subscribe(function (args) {
  playNote(args.frequency, args.duration);
});
```

Each message is received by the subscription to `play`, and the `playNote` JavaScript function is called to play the note.

[11] The Elm documentation points out that the `|>` is like a UNIX pipe[10]. Good point!

[12] Elm also provides an interface to directly import native JavaScript functions, somewhat like `FFI` in Haskell. The use of the native interface is discouraged since it may break the type safety guarantees provided by Elm[11].

[13] While learning how to use ports, I consulted the Elm web audio applications colluder[12] and elmkc-melodymaker[8]. My implementation is a bit like that of elkc-melodymaker, though I send less in my `Note` than in the comparable `PlayBundle`.

6.3 Pattern Selection

Now that the Minima is playing some music, we can introduce some variety by letting the user select new patterns. The rotation buttons on either side of each row shift through a set of patterns[14]. When a rotate button is pressed, the voice for the corresponding row is updated with a new pattern, the UI display is updated, and the pattern is read into the score.

A `Rotate` message is produced when a user clicks on a rotate button. The `Msg` is a case in the `update` function.

```
        Rotate voice direction ->
                    case voice.id of
                        "one" ->
                            { model
                                | one = rotate voice direction
                                , score = read voice direction model.score
                            }
                                ! []

                        "two" ->
                            { model
                                | two = rotate voice direction
                                , score = read voice direction model.score
                            }
                                ! []
                         ...
```

The `voice` is matched on its `id` and updated with a new pattern retrieved from the `rotate` function. Note that `Direction` is an abstract data type `type Direction = Left | Right`.

```
rotate : Voice -> Direction -> Voice
rotate voice direction =
    case elemIndex voice.pattern patterns of
        Just n ->
            case direction of
                Model.Left ->
                    { voice | pattern = getPatternAt ((n - 1) % List.length patterns) }

                Model.Right ->
                    { voice | pattern = getPatternAt ((n + 1) % List.length patterns) }

        Nothing ->
            { voice | pattern = voice.pattern }
```

[14]The set of patterns does not represent every possible pattern, only those which were interesting for this application.

The `rotate` function finds the index for the current pattern in `patterns`[15]. It then checks the case of `Left` or `Right` and retrieves the new pattern by index.

A `Ring` data structure is another possible approach for storing patterns. We could avoid indexing into `patterns` to rotate and define `forward` and `backward` functions that would cover the cases where we wrap around the ends of the list[16].

The blocks in the UI to show the new pattern will be updated by the Elm runtime. Once the changes to the model are made, our `view` function will handle the rendering logic.

The new pattern is read into the `score` by the `read` function.

```
read : Voice -> Direction -> Score -> Score
read voice direction score =
    (readPattern 1 voice (.pattern (rotate voice direction)) score)
        ++ (filterFrequency voice score)
```

We read the notes in the new pattern with `readPattern`. The old pattern is removed from the score by `filterFrequency`, and the remaining notes are concatenated onto the list of the new notes. Remember that `Score` is an alias for a list of notes. We are concatenating two lists of notes to to make our new `Score`.

Notice that `read` calls `rotate`. Although a call to `rotate` is occurs one line above the call to `read` in `update`, these do not happen sequentially. Recall that with each cycle Elm is producing a new program. The `voice` passed into `read` is the one before the rotation occurs, and we must rotate in `read` to assign the correct pattern.

The `readPattern` function transforms our new pattern into a `Score`.

```
readPattern : Int -> Voice -> Pattern -> Score -> Score
readPattern count voice pattern score =
    case pattern of
        [] ->
            []

        p :: ps ->
            case p of
                Model.Play n ->
                    (Note (.frequency voice) n count)
                        :: readPattern (n + count) voice ps score

                Model.Rest n ->
                    readPattern (n + count) voice ps score
```

We recursively add the new notes to the `score` by reading from `pattern`, starting with a count of 1. If the `pattern` has at least one `Action`, we pattern match on the first `Action`. If `pattern` is an empty list, all of the actions have been read and we return a empty list.

[15] `patterns` is the list of all available patterns

[16] We discussed this data structure during the last week of class.

When we have at least one `Action`, we match the first `Action` as a `Play` or a `Rest`. In the `Play` case we create a new note and call `readPattern` with the rest of the `pattern`. In the `Rest` case, we call `readPattern` with the remaining `pattern` without adding a note. Rests are implicit in the `score` and, unlike the view, we do not need to add anything.

With our `update` function complete, we can move onto some thoughtful testing.

7 Testing

We will manually test three aspects of Minima: the clock and looping mechanism, audio quality, and the set of note patterns. All tests were conducted on the online version of Minima in both Chrome and Firefox.

7.1 Clock and Looping

1. When the `Play` button is selected, the counter should display the count over a loop from 1 to 4.

   This test passes with one anomaly. When the play button is selected, the counter starts the count at 4.

2. When the `Pause` button is selected, the counter should display 0 and hold at 0.

   The clock stops as expected.

3. When the `Play` button is selected, Minima should loop over a note pattern, playing the correct note for each time mark.

   A simple arpeggio pattern was used to test audio looping.

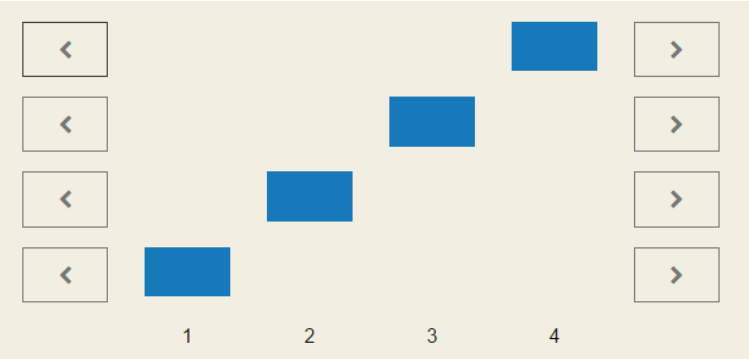

Fig. 2. An arpeggio

   The count starts at 4 as mentioned in first test, but the first note is played when the counter reads 1. One note is played per tick of the counter, and the notes ascend in frequency until the loop resets to 1 and the pattern begins again.

4. When the `Pause` button is selected, no sound should be produced.

   The music stops as expected and regardless of the current state of the clock.

7.2 Audio Quality

1. When no voice displays a note pattern and the clock is running, no sound should be produced.

Fig. 3. No note patterns selected

   The clock was run for a four loops and no sounds were audible. This test confirms that Minima does not produce any unexpected notes. This test passes.

2. When all voices are set to play a maximum duration note, no distortion in audio quality should be heard.

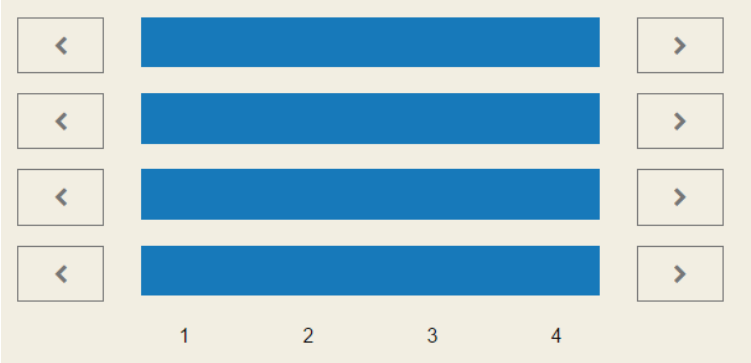

Fig. 4. Four simultaneous whole notes

   The pattern was played over four loops without perceptible audio distortion.

One aspect of audio quality was not tested but deserves to be mentioned. The web audio synthesizer uses an amplitude envelope to gracefully lower the volume at the end of each note. The envelope lowers the volume quickly in order to prevent an audio click produced when the browser stops the sound. Improvements to the web audio synthesizer would be needed to include sounds that release abruptly.

7.3 Note Patterns

The next set of tests check that notes in each pattern play for the correct duration. The envelope was altered to allow the notes to sound for their full length[17].

The note patterns are defined as `patterns` in Minima.

```
patterns : List Pattern
patterns =
    [ [ Play 4 ]
    , [ Play 2, Play 2 ]
    , [ Play 2, Rest 2 ]
    , [ Rest 2, Play 2 ]
    , [ Play 1, Rest 1, Play 1, Rest 1 ]
    , [ Rest 1, Play 1, Rest 1, Play 1 ]
    , [ Play 1, Rest 3 ]
    , [ Rest 1, Play 1, Rest 2 ]
    , [ Rest 2, Play 1, Rest 1 ]
    , [ Rest 3, Play 1 ]
    , [ Rest 4 ]
    ]
```

1. When the note pattern `[ Play 4 ]` is selected, a note should play for four full beats.

   This test uses the note pattern shown in Fig. 5. The notes shown play for the full count as expected.

2. When the note pattern `[ Play 2, Play 2 ]` is selected, two notes with a duration of 2 should be played sequentially.

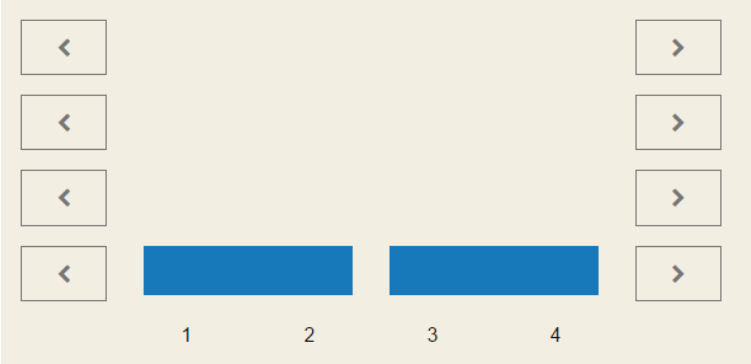

Fig. 5. Two half notes

   Each note was played for a duration of two in sequence as expected.

3. When the note patterns `[ Play 2, Rest 2 ]` and `[ Rest 2, Play 2 ]` are selected, two notes with a duration of 2 should be played sequentially at different frequencies.

[17]The accompanying audio click is acceptable for testing.

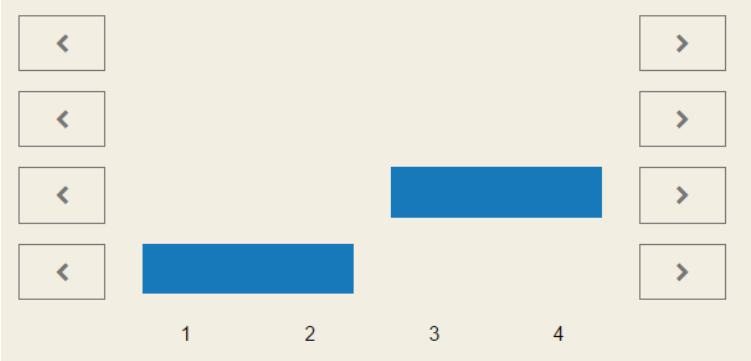

Fig. 6. Half note and half note

Each note was played for a duration of two in sequence as expected. The lower frequency note plays for counts 1 through 2, and the higher frequency note plays for counts 3 through 4.

4. When the note patterns `[ Play 1, Rest 1, Play 1, Rest 1 ]` and `[ Rest 1, Play 1, Rest 1, Play 1 ]` are played, two notes of duration one should be played twice over two frequencies.

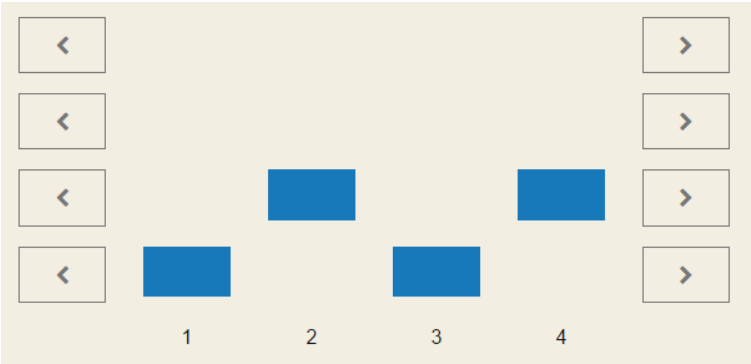

Fig. 7. Alternating quarter notes

Each frequency was played in alternation for a duration of one.

5. When the patterns `[ Play 1, Rest 3 ] , [ Rest 1, Play 1, Rest 2 ] , [ Rest 2, Play 1, Rest 1 ]` and `[ Rest 3, Play 1 ]` are played as an arpeggio, each note should sound for a duration of one and play from low to high frequency.

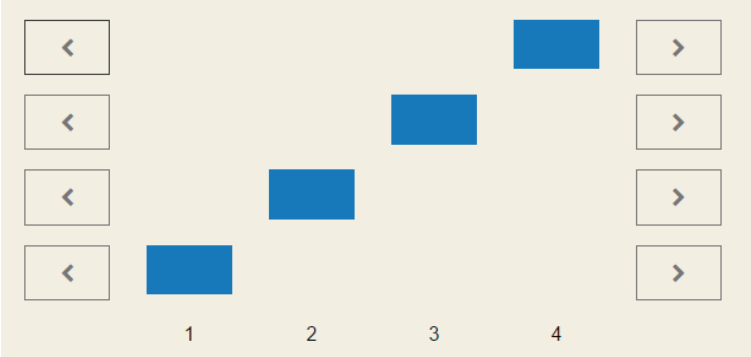

Fig. 8. Arpeggio of quarter notes

Each note was played for a duration of one, and the arpeggiated pattern rose from low to high frequency.

6. When the pattern `[Rest 4]` is played no sounds should be played.

   This note pattern was verified in the first Audio Quality test.

8 Future Work

Minima could be improved with a flexible model for voices and a better clock. Additional features could also be added.

The current implementation hard-codes voices in the `model`. They could instead be put into a data structure with an arbitrary number of values and built-in indexing[18]. An arbitrary number of voices would give some flexibility, voices could be added or removed by the user as desired.

The clock implementation might be improved by using `AnimationFrame`[13] instead of the `Time` library. The `Time` library is not intended for realtime applications and shows its limits when a user has multiple tabs open while Minima is playing. I am not certain that the `AnimationFrame` library would solve this problem, but it seems that animations might face similar rendering issues.

Additional features, such as a rate controller, could be added, but only with restraint. One of my major goals was to make Minima easy to use and approachable. With too many features, Minima would cease to be minimal.

Beyond Minima, Elm and web audio offer exciting possibilities. Both technologies have come around in the last five years and have mostly been taken on by early adopters. Ableton, a vendor of traditional DAWs, recently released a beta version "Get Started Making Music"[19], written with Elm and web audio[14]. Who knows what more may come of these new technologies.

[18]The Elm `Dict` data structure is a good candidate.
[19]Try it out, it's impressive: https://learningmusic.ableton.com/

9 Conclusion

Minima is simple web application that demonstrates the Elm programming language and web audio. Minima shows a small sample of the functional language techniques that Elm brings to front-end web development.

Many thanks to Mark Jones and Katie Casamento for your help and guidance on this project. You both helped me see the ways functional languages can improve expression and reduce complexity.

A Source Code and Build Instructions

The Minima source code is organized as follows.

```
.
|-- elm-package.json
|-- elm-stuff
|   |--...
|-- font-awesome
|   |--  ...
|-- index.html
|-- LICENSE.md
|-- main.js
|-- src
|   |-- Main.elm
|   |-- Model.elm
|   |-- Ports.elm
|   |-- Update.elm
|   |-- View.elm
|-- style.css
```

Note that `elm-stuff` and `font-awesome` are external dependencies. The contents of `elm-stuff` will be installed at build time. The `font-awesome` dependencies are included in the code repository for the project.

Minima can be built in a few simple steps.

1. Install Elm: https://guide.elm-lang.org/install.html
2. Clone the project: `git clone https://github.com/thuselem/minima`
3. Build the project: `elm-make src/Main.js --output=main.js`
4. Accept the plan for installing the Elm dependencies when prompted
5. Open `index.html` in your favorite web browser

The full source code for Minima is included below. Each file is noted with a `----` followed by the file name. `LICENSE.md` is a standard BSD three-clause license, copyright Brian Ginsburg, and is not included in the source code listing.

```
---- Main.elm

module Main exposing (..)

import Html exposing (Html)
import Time exposing (every, second)
import Model exposing (Model, model)
import Update exposing (..)
import View exposing (view)


main : Program Never Model Msg
main =
    Html.program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }


init : ( Model, Cmd Msg )
init =
    ( model
    , Cmd.none
    )


subscriptions : Model -> Sub Msg
subscriptions model =
    every second Tick


-- Model.elm

module Model exposing (..)


type alias Model =
    { score : Score
    , ticks : Int
    , clock : Int
    , one : Voice
    , two : Voice
    , three : Voice
    , four : Voice
    }


model =
    { score =
        [ Note 385 2 1
        , Note 385 2 3
        , Note 440 4 1
        , Note 660 1 2
        , Note 660 1 4
        , Note 880 1 1
        , Note 880 1 3
        ]
    , one = Voice "one" 385 [ Play 2, Play 2 ] -- 7/8
    , two = Voice "two" 440 [ Play 4 ] -- 1/1
    , three = Voice "three" 660 [ Rest 1, Play 1, Rest 1, Play 1 ] -- 3/2
    , four = Voice "four" 880 [ Play 1, Rest 1, Play 1, Rest 1 ] -- 2/1
    , ticks = 4
    , clock = 0
    }
```

```
-- SCORE


type alias Score =
    List Note


type alias Note =
    { frequency : Float
    , duration : Int
    , tick : Int
    }



-- VOICE


type alias Voice =
    { id : String
    , frequency : Float
    , pattern : Pattern
    }


type alias Pattern =
    List Action


type Action
    = Play Int
    | Rest Int


type Direction
    = Left
    | Right


patterns : List Pattern
patterns =
    [ [ Play 4 ]
    , [ Play 2, Play 2 ]
    , [ Play 2, Rest 2 ]
    , [ Rest 2, Play 2 ]
    , [ Play 1, Rest 1, Play 1, Rest 1 ]
    , [ Rest 1, Play 1, Rest 1, Play 1 ]
    , [ Play 1, Rest 3 ]
    , [ Rest 1, Play 1, Rest 2 ]
    , [ Rest 2, Play 1, Rest 1 ]
    , [ Rest 3, Play 1 ]
    , [ Rest 4 ]
    ]
```

```
-- View.elm


module View exposing (..)

import Html exposing (..)
import Html.Attributes exposing (..)
import Html.Events exposing (onClick)
import Model exposing (Model, Voice, Score, Direction)
import Update exposing (..)


view : Model -> Html Msg
view model =
    div []
        [ title
        , showRow model.four
        , showRow model.three
        , showRow model.two
        , showRow model.one
        , showCounter model.ticks
        , instructions model.clock
        ]


title : Html Msg
title =
    div [ class "row" ]
        [ div [ class "columns" ]
            [ h1 [] [ text "minima" ] ]
        ]


showRow : Model.Voice -> Html Msg
showRow voice =
    case .pattern voice of
        [] ->
            div [ class "row align-center" ] []

        acts ->
            div [ class "row align-center" ]
                (rotateVoice voice Model.Left
                    :: List.map renderAction acts
                    ++ [ rotateVoice voice Model.Right ]
                )


renderAction : Model.Action -> Html Msg
renderAction action =
    let
        block duration text =
            div [ class ("column small-" ++ toString duration) ] text
        space =
            [ text "\x2002" ] -- unicode space
    in
        case action of
            Model.Play duration ->
                block duration [ a [ class "expanded button" ] space ]

            Model.Rest duration ->
                block duration space
```

```
-- View.elm (continued)

rotateVoice : Voice -> Direction -> Html Msg
rotateVoice voice direction =
    div [ class "column small-1" ]
        [ case direction of
            Model.Left ->
                a [ class "expanded hollow secondary button fa fa-chevron-left", onClick (Rotate voice direction

            Model.Right ->
                a [ class "expanded hollow secondary button fa fa-chevron-right", onClick (Rotate voice directio
        ]


showCounter : Int -> Html Msg
showCounter last =
    div [ class "counter row align-center" ]
        (counter 1 last)

counter : Int -> Int -> List (Html Msg)
counter current last =
    case current > last of
        True ->
            []

        False ->
            [ div [ class "column small-1" ] [ text (toString current) ] ]
                ++ counter (current + 1) last


instructions : Int -> Html Msg
instructions clock =
    div [ class "row align-center" ]
        [ div [ class "column small-6" ]
            [ p []
                [ text "Minima is a playground for experimenting with minimalist musical patterns." ]
            , p []
                [ text "Press "
                , a [ class "control fa fa-play", onClick Play ] []
                , text "to start the music and"
                , a [ class "control fa fa-pause", onClick Pause ] []
                , text "to stop it."
                ]
            , p []
                [ text " Select patterns with the "
                , a [ class "control secondary fa fa-chevron-left" ] []
                , text "and  "
                , a [ class "control secondary fa fa-chevron-right" ] []
                , text "buttons."
                ]
            , p []
                [ text "Minima has four beats, and the current beat is "
                , span [ class "clock" ] [ text (showClock clock ++ ".") ]
                ]
            , p []
                [ text "Have fun and I hope you enjoy!" ]
            ]
        ]


showClock : Int -> String
showClock clock =
    case clock of
        0 ->
            "0"

        1 ->
            "4"

        _ ->
            toString (clock - 1)
```

```
---- Update.elm

module Update exposing (..)

import List.Extra exposing (..)
import Time exposing (Time)
import Model exposing (..)
import Ports exposing (..)


type Msg
    = Tick Time
    | Play
    | Pause
    | Rotate Voice Direction


update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        Tick time ->
            { model
                | clock = increment model.clock model.ticks
            }
                ! [ Cmd.batch (playNotes model.score model.clock) ]

        Play ->
            { model | clock = 1 } ! []

        Pause ->
            { model | clock = 0 } ! []

        Rotate voice direction ->
            case voice.id of
                "one" ->
                    { model
                        | one = rotate voice direction
                        , score = read voice direction model.score
                    }
                        ! []

                "two" ->
                    { model
                        | two = rotate voice direction
                        , score = read voice direction model.score
                    }
                        ! []

                "three" ->
                    { model
                        | three = rotate voice direction
                        , score = read voice direction model.score
                    }
                        ! []

                "four" ->
                    { model
                        | four = rotate voice direction
                        , score = read voice direction model.score
                    }
                        ! []

                _ ->
                    model ! []
```

```
---- Update.elm (continued)

increment : Int -> Int -> Int
increment clock ticks =
    case clock of
        0 ->
            0

        _ ->
            (clock % ticks) + 1


playNotes : Score -> Int -> List (Cmd msg)
playNotes score clock =
    List.filter (\n -> .tick n == clock) score
        |> List.map play



-- READ


read : Voice -> Direction -> Score -> Score
read voice direction score =
    (readPattern 1 voice (.pattern (rotate voice direction)) score)
        ++ (filterFrequency voice score)


readPattern : Int -> Voice -> Pattern -> Score -> Score
readPattern count voice pattern score =
    case pattern of
        [] ->
            []

        p :: ps ->
            case p of
                Model.Play n ->
                    (Note (.frequency voice) n count)
                        :: readPattern (n + count) voice ps score

                Model.Rest n ->
                    readPattern (n + count) voice ps score


filterFrequency : Voice -> Score -> Score
filterFrequency voice score =
    List.filter (\n -> .frequency n /= .frequency voice) score


-- ROTATE


rotate : Voice -> Direction -> Voice
rotate voice direction =
    case elemIndex voice.pattern patterns of
        Just n ->
            case direction of
                Model.Left ->
                    { voice | pattern = getPatternAt ((n - 1) % List.length patterns) }

                Model.Right ->
                    { voice | pattern = getPatternAt ((n + 1) % List.length patterns) }

        Nothing ->
            { voice | pattern = voice.pattern }
```

—— Update.elm (continued)

```
getPatternAt : Int -> Pattern
getPatternAt index =
    case patterns !! index of
        Just pattern ->
            pattern

        Nothing ->
            []
```

—— Ports.elm

```
port module Ports exposing (..)

import Model exposing (Note)

port play : Note -> Cmd msg
```

—— elm-package.json

```
{
    "version": "1.0.0",
    "summary": "A web audio playground for mininmalist music",
    "repository": "https://github.com/thuselem/minima.git",
    "license": "BSD3",
    "source-directories": [
        "src"
    ],
    "exposed-modules": [],
    "dependencies": {
        "elm-community/list-extra": "6.1.0 <= v < 7.0.0",
        "elm-lang/core": "5.1.1 <= v < 6.0.0",
        "elm-lang/html": "2.0.0 <= v < 3.0.0"
    },
    "elm-version": "0.18.0 <= v < 0.19.0"
}
```

—— index.html

```
<!doctype HTML>
<html>
    <head>
      <title>minima</title>
      <meta charset="utf-8" />
      <link href="https://fonts.googleapis.com/css?family=Cinzel|Quattrocento+Sans" rel="stylesheet">
      <link href="https://cdnjs.cloudflare.com/ajax/libs/foundation/6.3.1/css/foundation-flex.min.css" \
       rel="stylesheet">
      <link rel="stylesheet" href="font-awesome/css/font-awesome.min.css">
      <link rel="stylesheet" href="style.css">
    </head>

    <body>
        <div id="main"></div>
        <script type="text/javascript" src="main.js"></script>
        <script>
         "use strict"
         var app = Elm.Main.fullscreen();

         var context = new (window.AudioContext || window.webkitAudioContext)();

         var waveform = 'sine',
             filterType = 'lowpass',
             cutoff = 3000,
             gain = 0.2;

         var filter = context.createBiquadFilter();
         filter.type = filterType;
         filter.frequency.value = cutoff;
         filter.connect(context.destination);


         function playNote (frequency, duration) {

           // intstantiate and initialize nodes
           var osc = context.createOscillator();
           var amp = context.createGain();
           osc.type = waveform;
           osc.frequency.value = frequency;
           amp.gain.value = gain;

           // routing
           osc.connect(amp);
           amp.connect(filter);

           // set timing and amp envelope
           var now = context.currentTime;
           amp.gain.cancelScheduledValues(now);
           amp.gain.linearRampToValueAtTime(gain + 0.05, now + 0.03);  // attack
           amp.gain.setTargetAtTime(gain, now + 0.03, 0.01);  // decay
           amp.gain.setValueAtTime(gain, now + 0.03);  // set mark for release
           amp.gain.exponentialRampToValueAtTime(0.0001, now + duration);  // release

           osc.start();
           osc.stop(now + duration);  // overshoot duration for release
         };

         app.ports.play.subscribe(function (args) {
           playNote(args.frequency, args.duration);
         });

        </script>
    </body>

</html>
```

```
-- style.css

body {
  background: #f2eee2;
}

div.counter {
  text-align: center;
  padding-bottom: 1em;
}

p {
  font-family: 'Quattrocento Sans', sans-serif;
  font-size: 1.4rem;
  text-align: center;
}

h1 {
  font-family: 'Cinzel', serif;
  font-size: 4.5rem;
  text-align: center;
  margin: 1rem 0 2rem 0;
}

a.control {
  margin: 0 0.5em 0 0.5em;
}

a.secondary {
  color: #767676;
}

span.clock {
  display: inline-block;
  width: 0.8em;
  font-size: 1.5rem;
}
```

References

[1] Just Intonation. Retrieved from https://en.wikipedia.org/wiki/Just_intonation

[2] Czaplicki, E. (2012). Elm: Concurrent FRP for Functional GUIs. Retrieved from https://www.seas.harvard.edu/sites/default/files/files/archived/Czaplicki.pdf

[3] Czaplicki, E. (2014). Blazing Fast HTML. Retrieved from (http://elm-lang.org/blog/blazing-fast-html.

[4] The Elm Architecture. Retrieved from https://guide.elm-lang.org/architecture/.

[5] haskell-to-elm. The comparison between Elm and Haskell syntax was inspired and informed by this guide. Retrieved from https://github.com/eeue56/haskell-to-elm

[6] Html msg vs HTML Msg A discussion on Reddit with some code examples using `Html`. Retrieved from https://www.reddit.com/r/elm/comments/63zvja/html_msg_vs_html_msg/

[7] Time. Time library API from elm-lang core. Retrieved from http://package.elm-lang.org/packages/elm-lang/core/latest/Time

[8] elmkc-melodymaker. Basic web audio in Elm example. Retrieved from https://github.com/Fedreg/elmkc-melodymaker/blob/master/Main.elm

[9] JavaScript Interop. Elm language official guide for JavaScript Interops https://guide.elm-lang.org/interop/javascript.html

[10] Elm Syntax. Retrieved from http://elm-lang.org/docs/syntax

[11] Stance on native APIs for 0.18? Native code is discouraged, though it is hard to track down an official statement. In fact, official documentation on Native itself is hard to find. Retrieved from https://groups.google.com/forum/#!topic/elm-discuss/nuR4NnCVcMs

[12] Colluder. A collaborative music environment. https://github.com/knewter/colluder

[13] AnimationFrame. An animation library for repainting things as fast as possible. http://package.elm-lang.org/packages/elm-lang/animation-frame/1.0.1/AnimationFrame

[14] Twitter post from Ableton Dev. Make music in any modern browser with learningmusic.ableton.com. Uses #WebAudio and Tone.js, with interactive components written in #Elm. Retrieved from https://twitter.com/abletondev/status/861580662620508160?lang=en