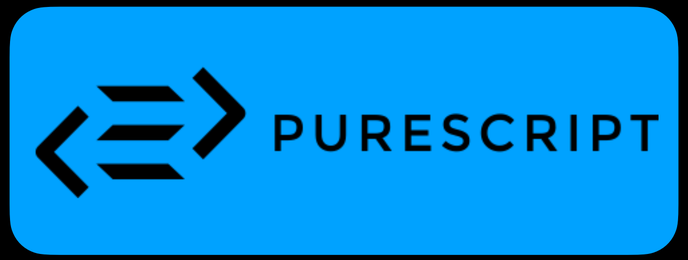
PureScript:
with a chance of Free Monads

Anton Kotenko a.k.a. Elektrokiłka

λ ‘(head 23) meetup @ JetBrains

Why PureScript?



RPD

http://shamansir.github.io/rpd/

RPD — Reactive Patch Development v2.0.0

§ Introduction

(GitHub | Examples | API | Issues | Terminology)

What is RPD?

RPD is the abbreviation for *Reactive Patch Development*...

...or, actually, whatever you decide. It is the library which brings node-based user interfaces to the modern web, in full their power (when you know how to use it) and in a very elegant and minimalistic way. *Node-based* is something like the thing you'll (probably) see above if you move your mouse cursor, or any other pointing device, above the RPD logo — (almost) nothing to do with node.js. Some people also say that with such user interfaces they do *Flow Programming*. If you are wondering yet, what that means, *Node-based* interface is the one where man may visually connect different low-level components using their inputs and outputs and observe the result in real time, take PureData, QuartzComposer, VVVV, NodeBox, Reaktor etc. for example.

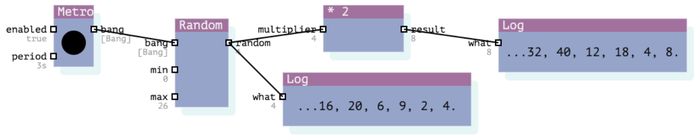
RPD brings DataFlow Programming to the Web both in the *elegant* and *minimal* ways.

RPD

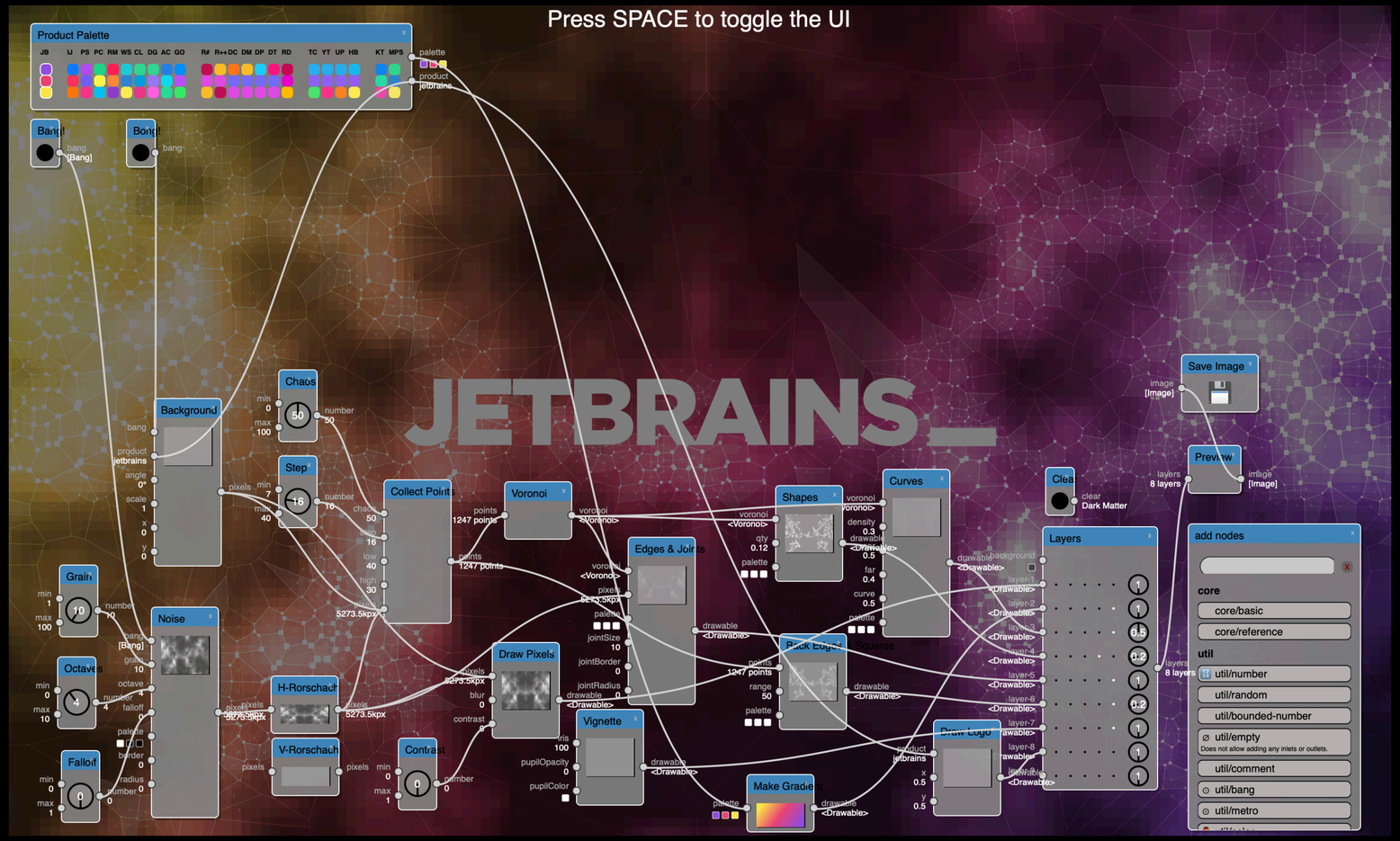
http://shamansir.github.io/rpd/

Random Generator

Random Generator with the help of util toolkit. Following a signal of Metronome, which "bangs" every 3 seconds by default, Random Generator yields a new value between specified minimum and maximum (here: a number from 0 to 26, just for the sake of the example). The result then goes to a multiply-by-two Node, which is produced from a new type defined in the code. Then we Log last five generated random values and last five multiplied value, just to keep track on things.

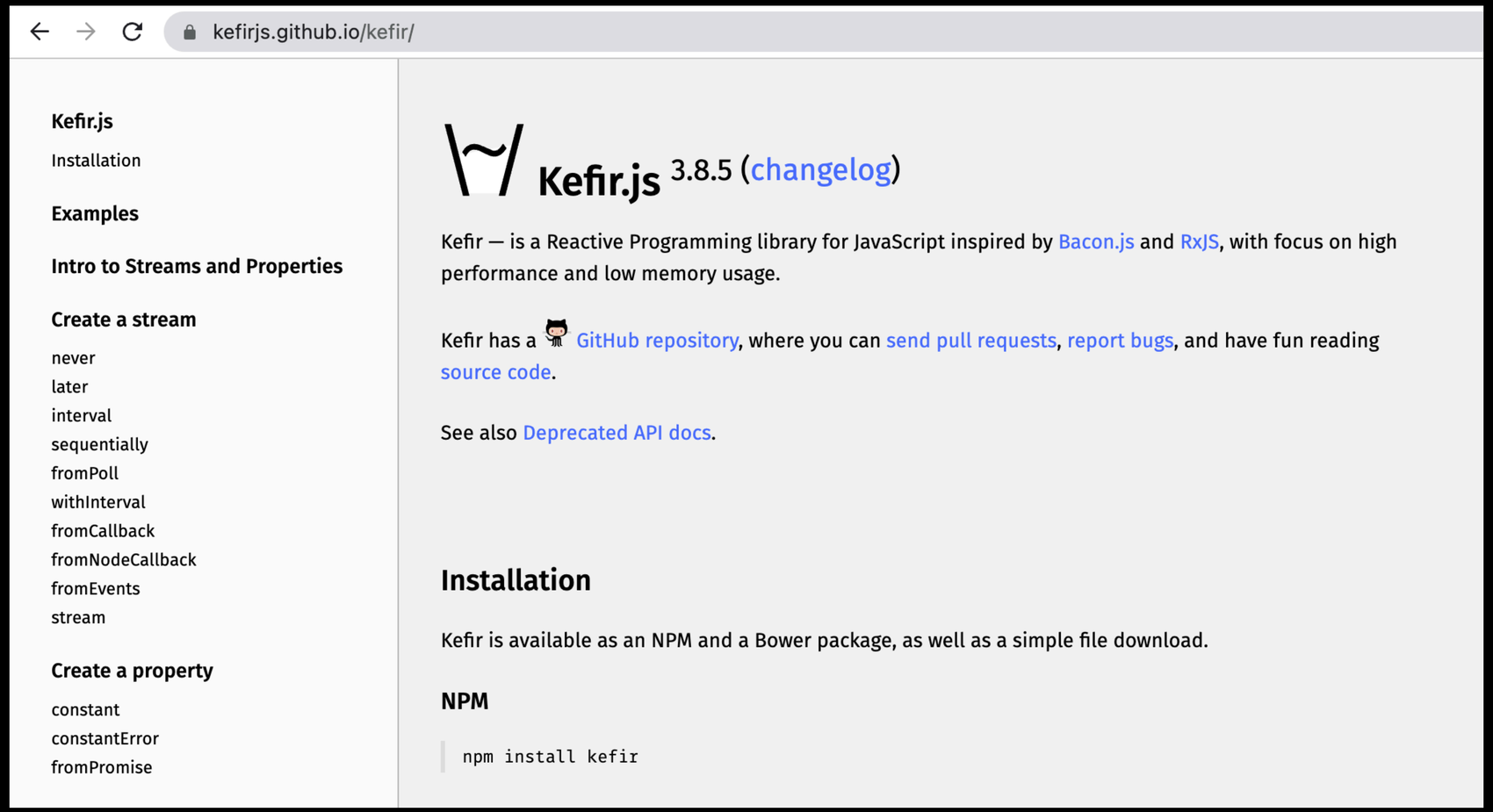


```
Rpd.renderNext('svg', document.getElementById('example-one'),
               { style: 'compact-v' });

var patch = Rpd.addPatch('Generate Random Numbers').resizeCanvas(800, 110);

// add Metro Node, it may generate `bang` signal with the requested time interval
var metroNode = patch.addNode('util/metro', 'Metro').move(40, 10);

// add Random Generator Node that will generate random numbers on every `bang` signal
var randomGenNode = patch.addNode('util/random', 'Random').move(130, 20);
randomGenNode.inlets['max'].receive(26); // set maximum value of the generated numbers

// add Log Node, which will log last results of the Random Generator Node
var logRandomNode = patch.addNode('util/log', 'Log').move(210, 60);
randomGenNode.outlets['out'].connect(logRandomNode.inlets['what']);

// define the type of the node which multiplies the incoming value by two
var multiplyTwoNode = patch.addNode('core/basic', '* 2', {
    process: function(inlets) {
        return {
            'result': (inlets.multiplier || 0) * 2
        }
    }
}).move(240, 10);
var multiplierInlet = multiplyTwoNode.addInlet('util/number', 'multiplier');
var resultOutlet = multiplyTwoNode.addOutlet('util/number', 'result');

// connect Random Generator output to the multiplying node
var logMultiplyNode = patch.addNode('util/log', 'Log').move(370, 20);
resultOutlet.connect(logMultiplyNode.inlets['what']);

// connect Random Generator output to the multiplying node
randomGenNode.outlets['out'].connect(multiplierInlet);
```



SETI (Safari only)

https://seti.labs.jb.gg/



Kefir

https://kefirjs.github.io/kefir/

Flare

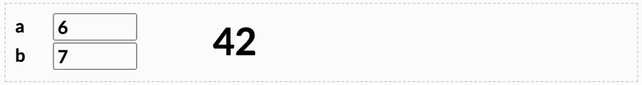
Flare is a *special-purpose* UI library for Purescript. The idea is to define the user interface and the logic of the program at the same time. On the one hand this approach somewhat limits the number of use cases, but on the other hand this allows for a remarkable expressiveness.

A first example

Imagine you want to build a really simple web application that calculates the product $a \times b$ for two numbers a and b which can be entered by the user. Using Flare, we define the *user interface* and the *program logic* in a single expression:

```
lift2 (*) (int "a" 6) (int "b" 7)
```

This code generates the following reactive web-interface[1]:



How does this work? Flare defines the function `int` that takes two arguments: a label and an initial value[2]. An expression like `int "a" 6` serves two purposes. It creates the input field with the label *"a"* and initial value 6, and at the same time it serves as a placeholder for its *current value* in the surrounding expression. If the user changes one of the input fields, the whole expression is re-evaluated.

Pure functions

In functional programming languages, programs are built from small, composable pieces that can be indivually tested. Suppose we have a *pure* function `greet`, that takes a string (a name) and returns a string (a greeting with a fancy version of the name):

PureScript Flare

https://david-peter.de/articles/flare/

```
coloredCircle :: Number -> Number -> Drawing
coloredCircle hue radius =
  filled (fillColor (hsl hue 0.8 0.4)) (circle 50.0 50.0 radius)
```

Note that this is a pure function without any side effects. Again, we can easily turn this into a web interface by creating two Flare components of the appropriate type for the two parameters. Here, we will use `numberSlider label min max step default`. Finally, the higher order function `lift2` takes our two-argument function and applies it to the current values inside the two components:

```
lift2 coloredCircle (numberSlider "Hue"    0.0 360.0 1.0 140.0)
                    (numberSlider "Radius" 2.0  45.0 0.1  25.0)
```

This results in the following interface:

Integration with other signals

The Flare library is built on top of Purescripts Signal library which is inspired by the corresponding module in Elm. In fact, a *Flare* is just a *Signal* with a collection of input fields. In this section, we demonstrate how Flare works together with other types of Signals. As a showcase, we want to integrate time to create a small animation:

Again, we start with a pure function. It takes three parameters, the number of leaves, a boolean flag which enables or disables the shadow, and a rotation angle:

```
plot :: Int -> Boolean -> Number -> Drawing
plot nLeaves shadow phi0 = ...
```

PureScript Flare

https://david-peter.de/articles/flare/

```
coloredCircle :: Number -> Number -> Drawing
coloredCircle hue radius =
  filled (fillColor (hsl hue 0.8 0.4)) (circle 50.0 50.0 radius)
```

Note that this is a pure function without any side effects. Again, we can easily turn this into a web interface by creating two Flare components of the appropriate type for the two parameters. Here, we will use `numberSlider label min max step default`. Finally, the higher order function `lift2` takes our two-argument function and applies it to the current values inside the two components:

```
lift2 coloredCircle (numberSlider "Hue"    0.0 360.0 1.0 140.0)
                    (numberSlider "Radius" 2.0  45.0 0.1  25.0)
```

This results in the following interface:

Integration with other signals

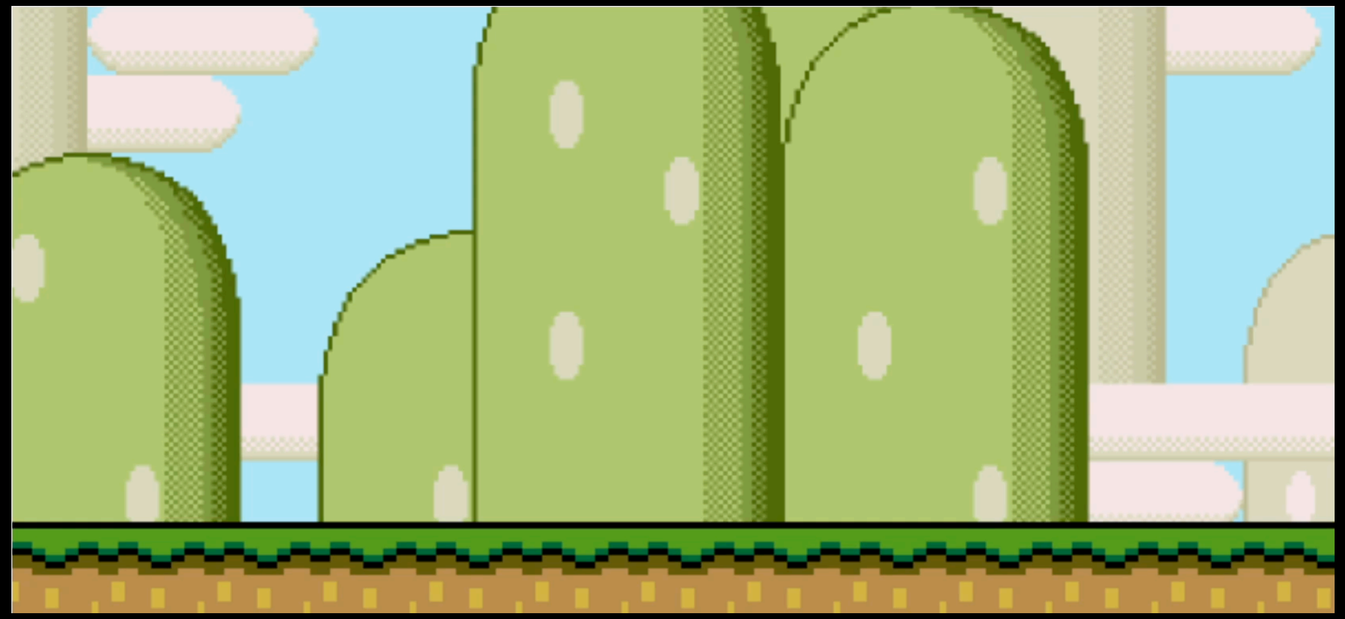
The Flare library is built on top of Purescripts Signal library which is inspired by the corresponding module in Elm. In fact, a *Flare* is just a *Signal* with a collection of input fields. In this section, we demonstrate how Flare works together with other types of Signals. As a showcase, we want to integrate time to create a small animation:

Again, we start with a pure function. It takes three parameters, the number of leaves, a boolean flag which enables or disables the shadow, and a rotation angle:

```
plot :: Int -> Boolean -> Number -> Drawing
plot nLeaves shadow phi0 = ...
```

PureScript Flare

https://david-peter.de/articles/flare/



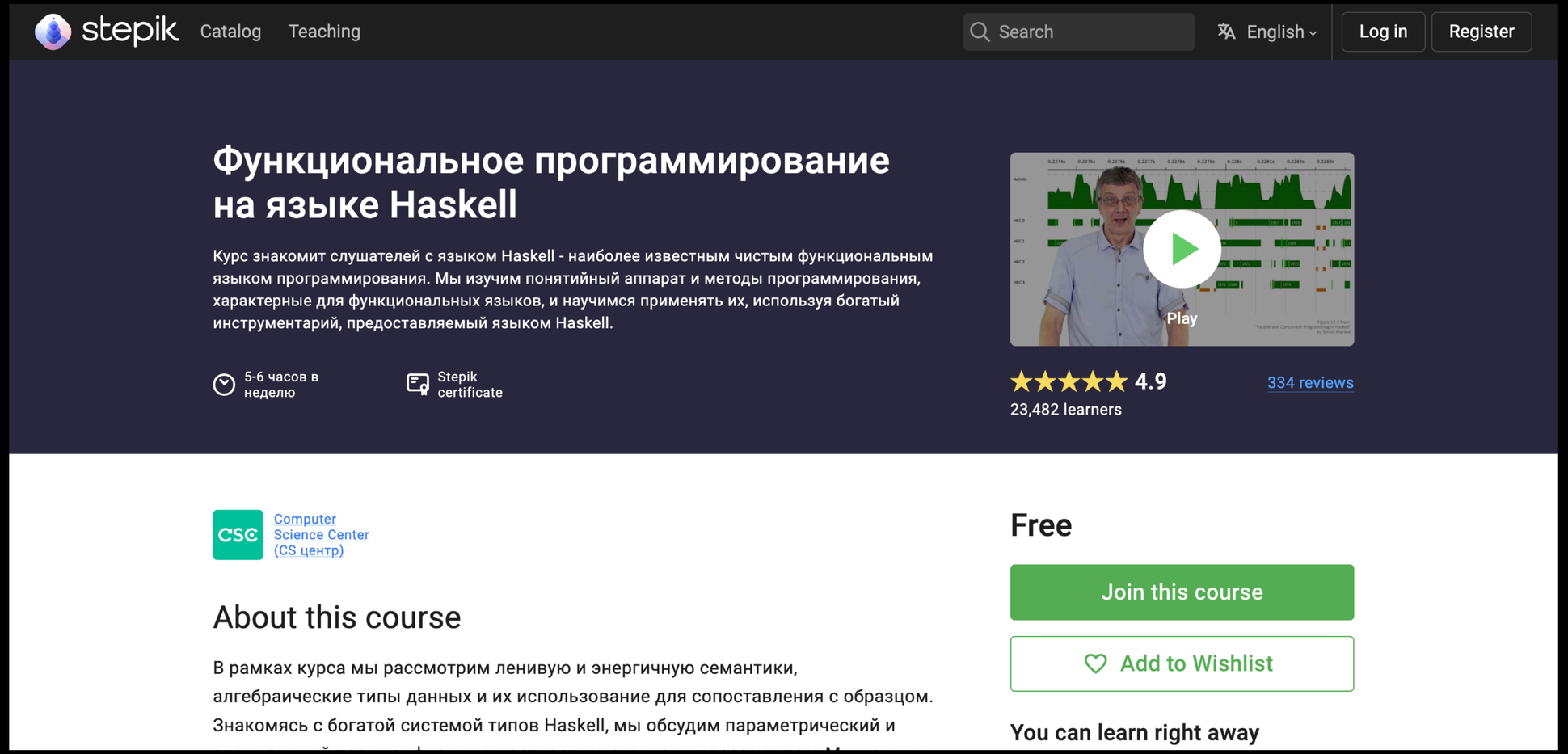
PureScript Mario on Signals

https://github.com/michaelficarra/purescript-demo-mario

```
98  marioLogic :: { left :: Boolean, right :: Boolean, jump :: Boolean } -> Character -> Character
99  marioLogic inputs = velocity <<< applyGravity <<< walk inputs.left inputs.right <<< jump inputs.jump
```

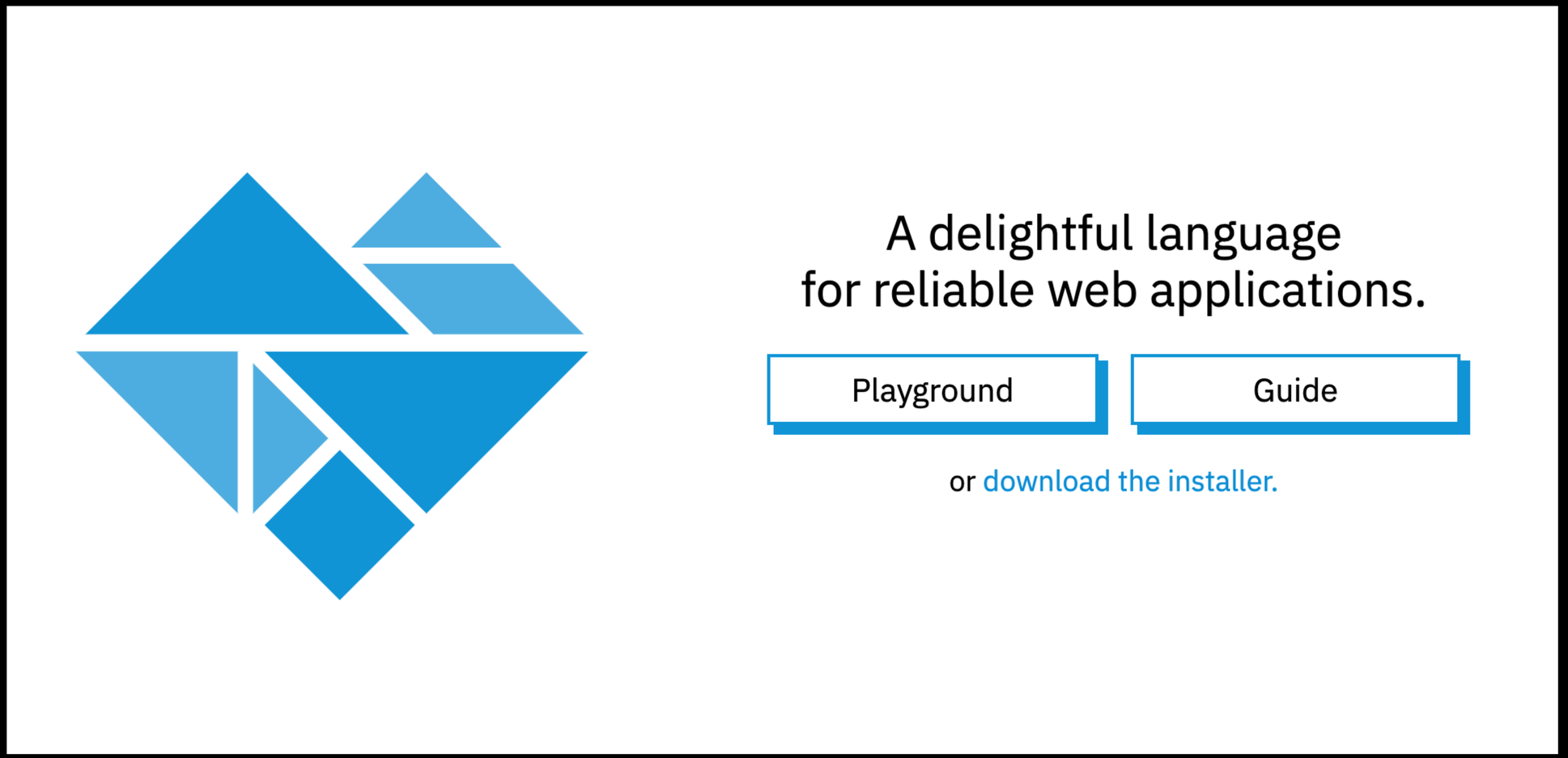
PureScript Mario on Signals

https://github.com/michaelficarra/purescript-demo-mario



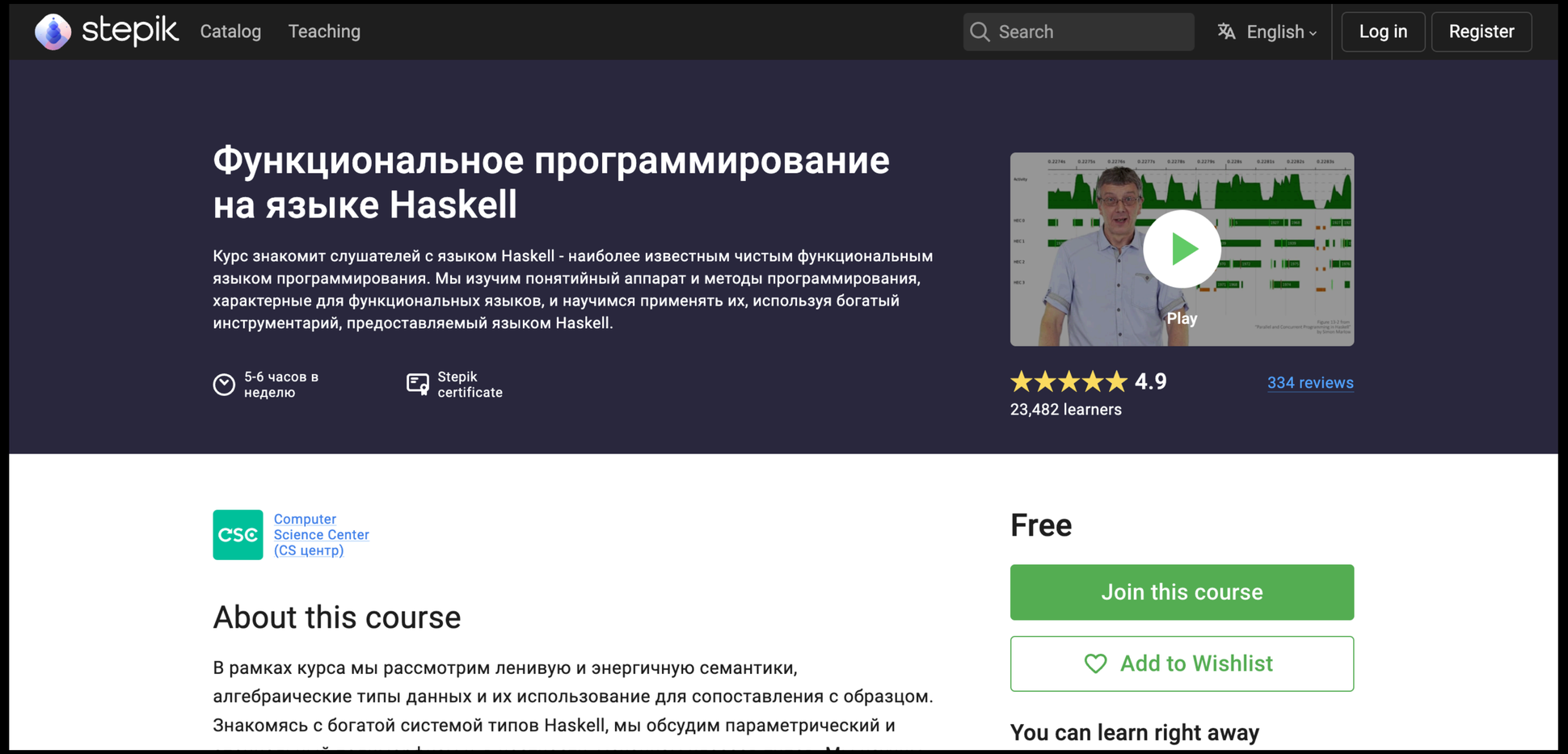
Haskell course @ Stepik

https://stepik.org/course/75/



Elm language

https://elm-lang.org/

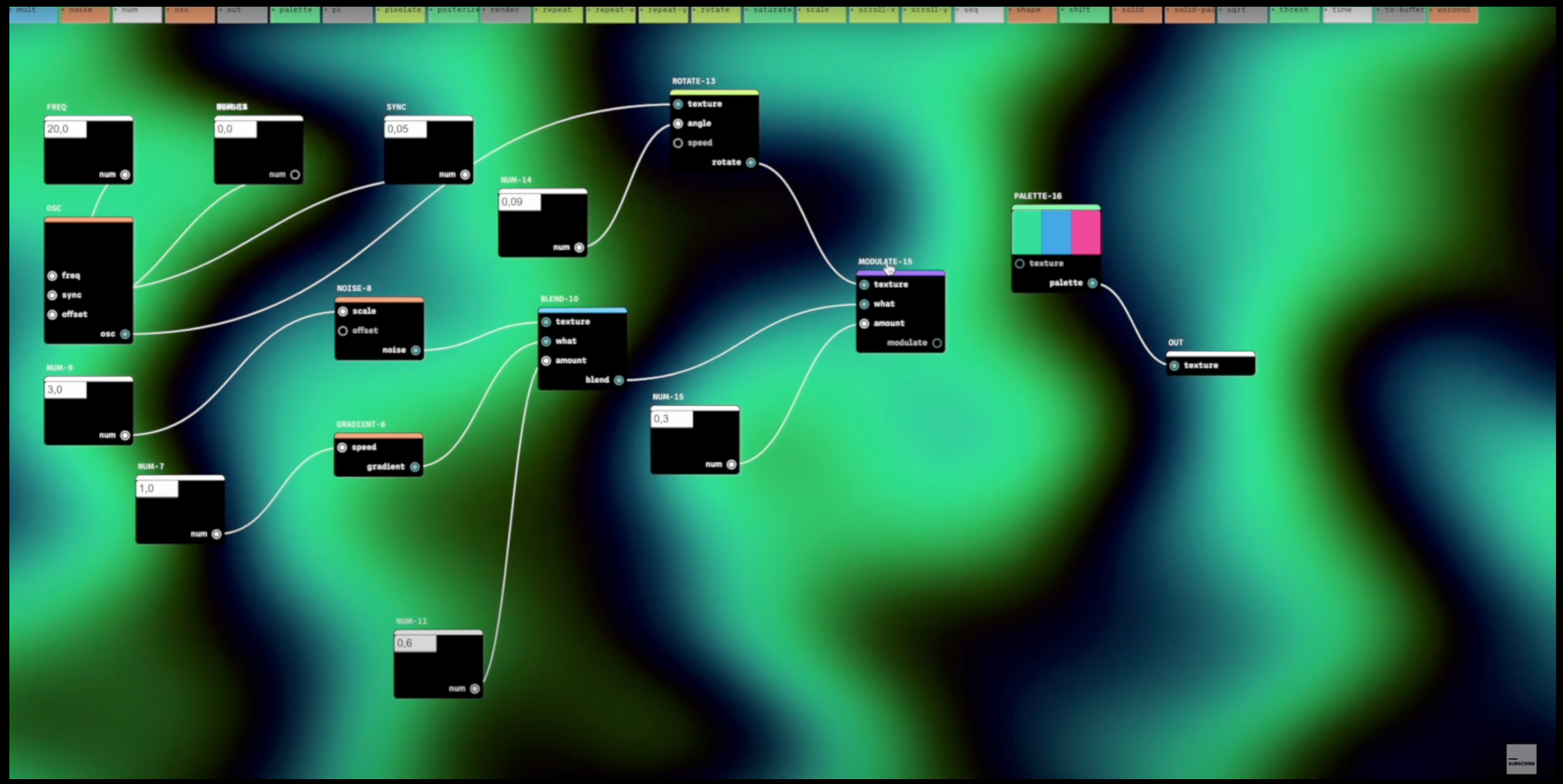


Haskell course @ Stepik

https://stepik.org/course/75/

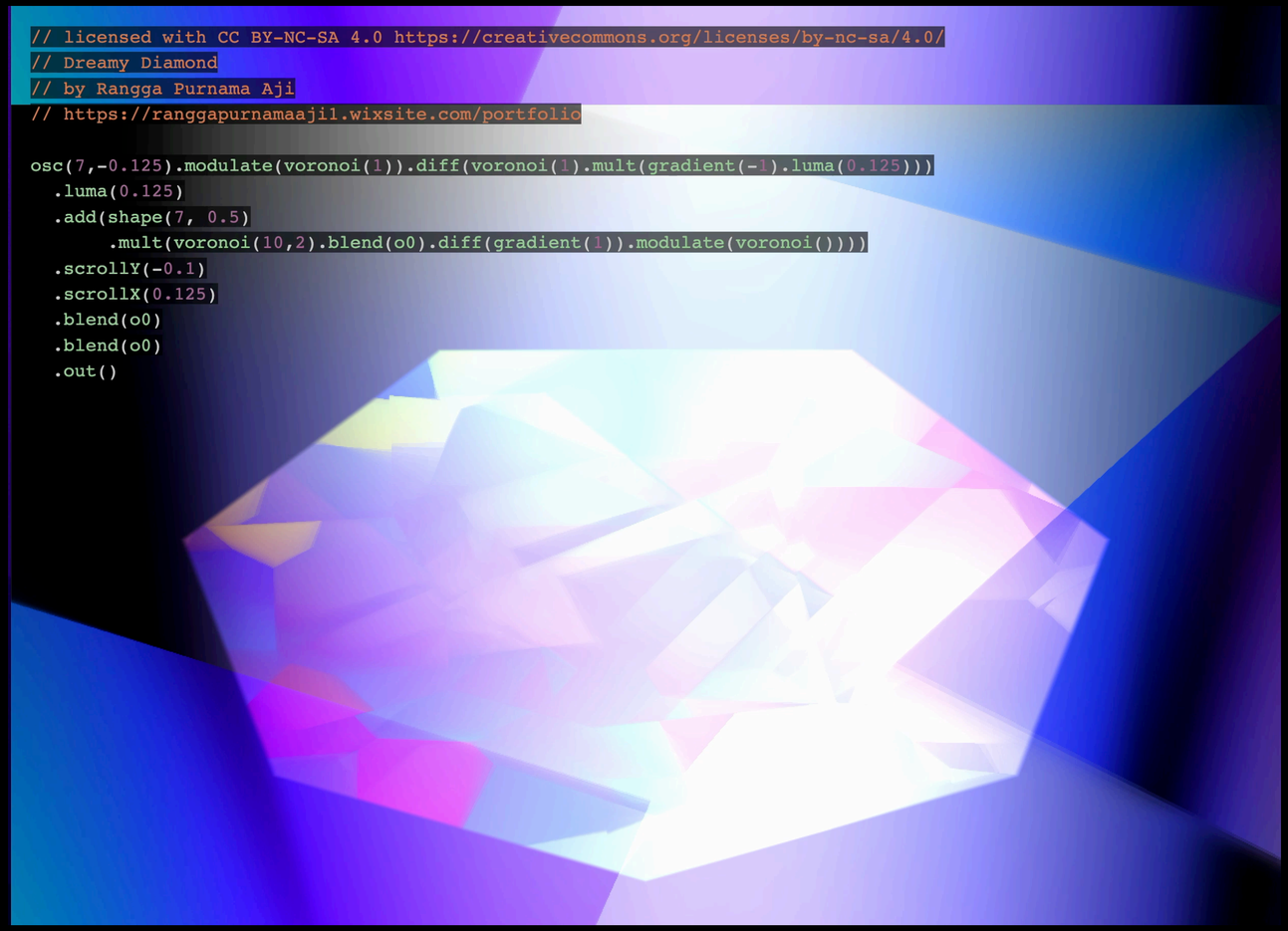
Noodle

https://noodle.labs.jb.gg/



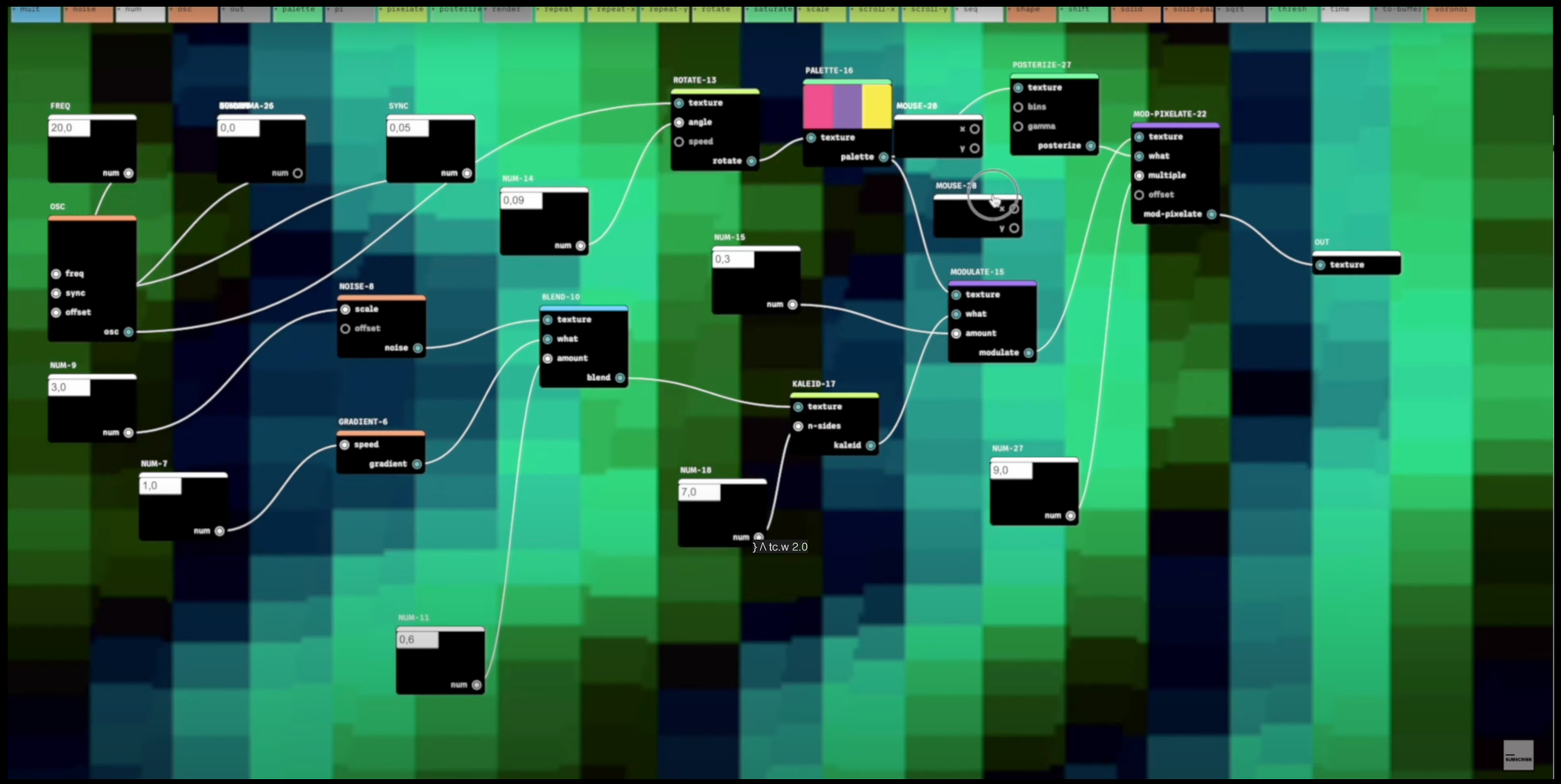
Noodle

https://noodle.labs.jb.gg/



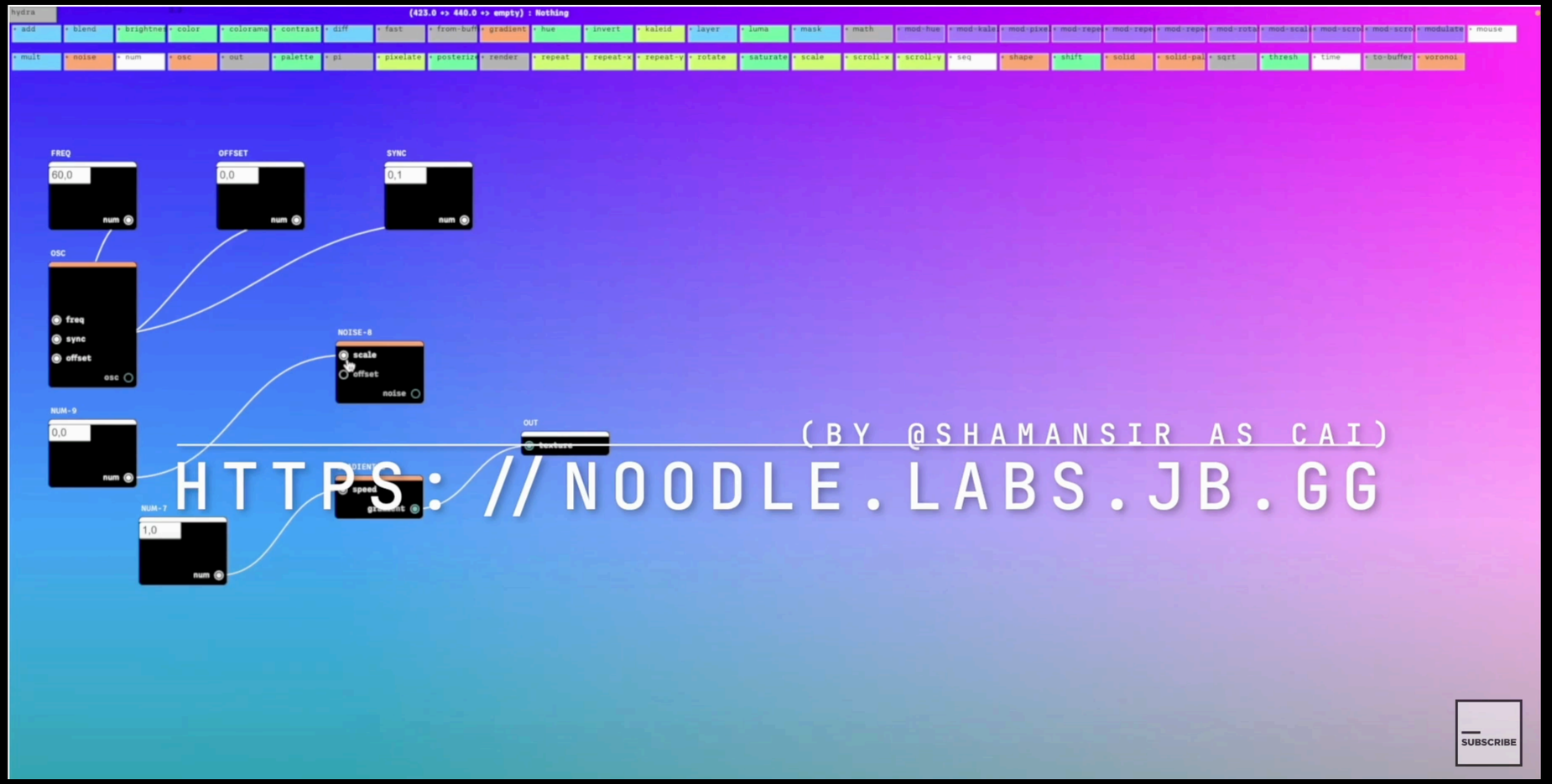
Hydra

https://hydra.ojack.xyz/



Noodle

https://noodle.labs.jb.gg/

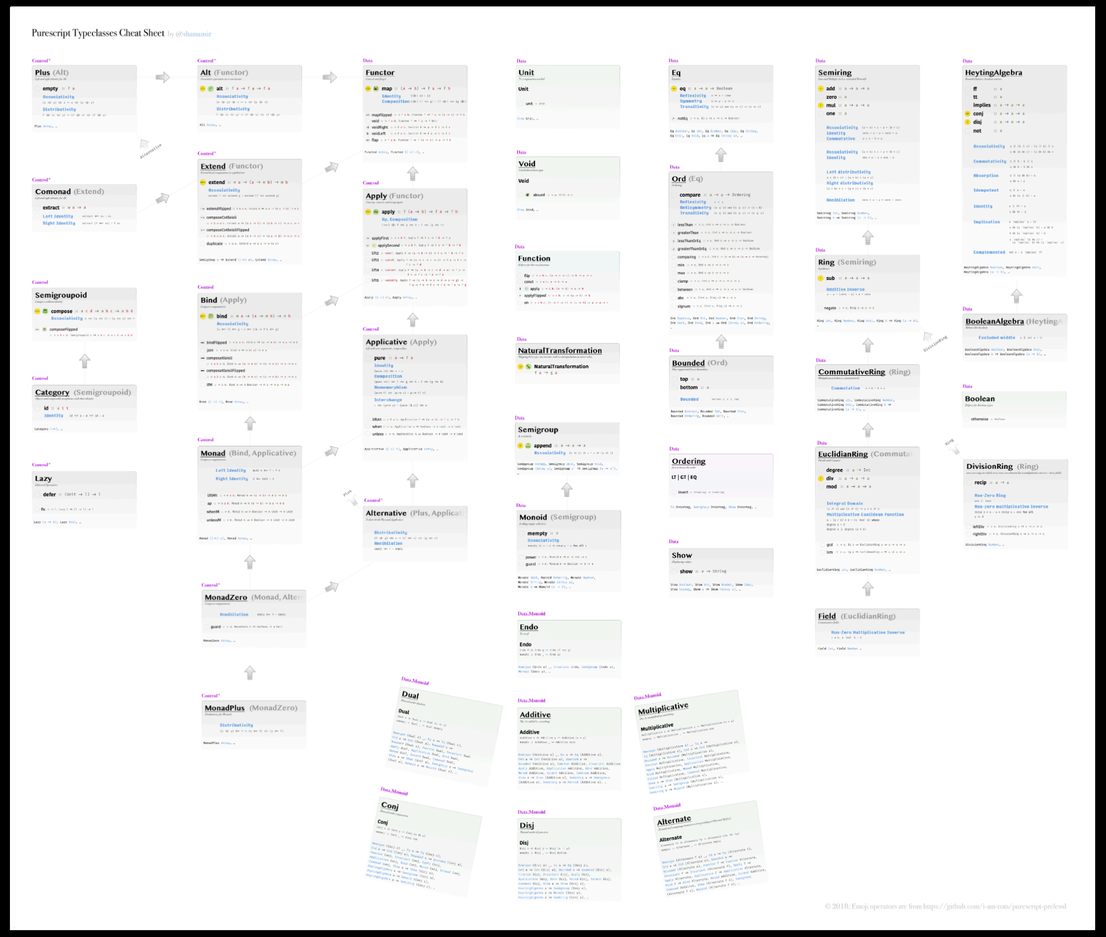


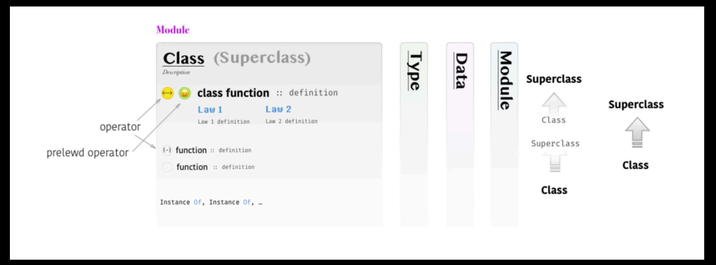
Noodle

https://noodle.labs.jb.gg/

Noodle

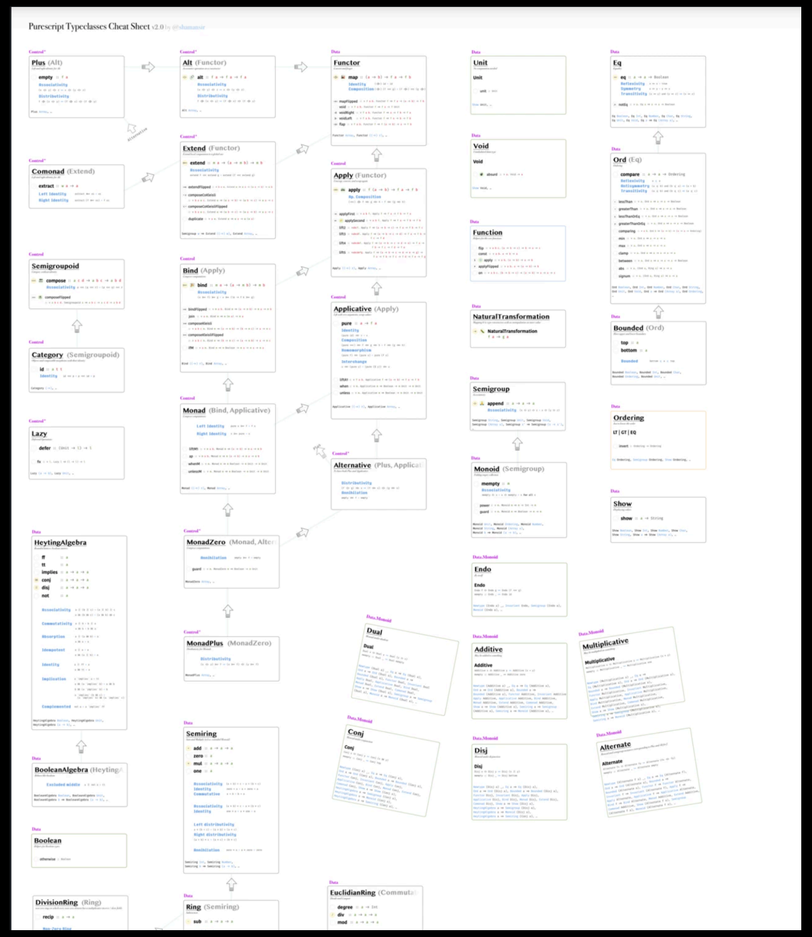
https://noodle.labs.jb.gg/

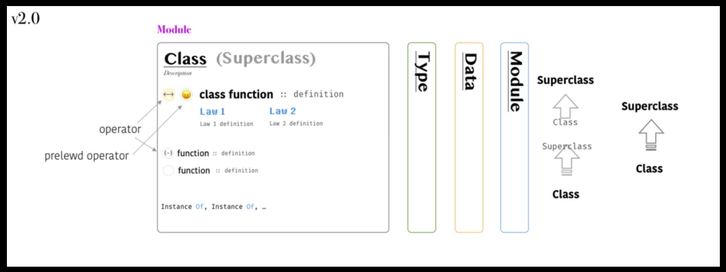




PureScript Typeclasses Cheatsheet

https://imgur.com/a/ESOR7

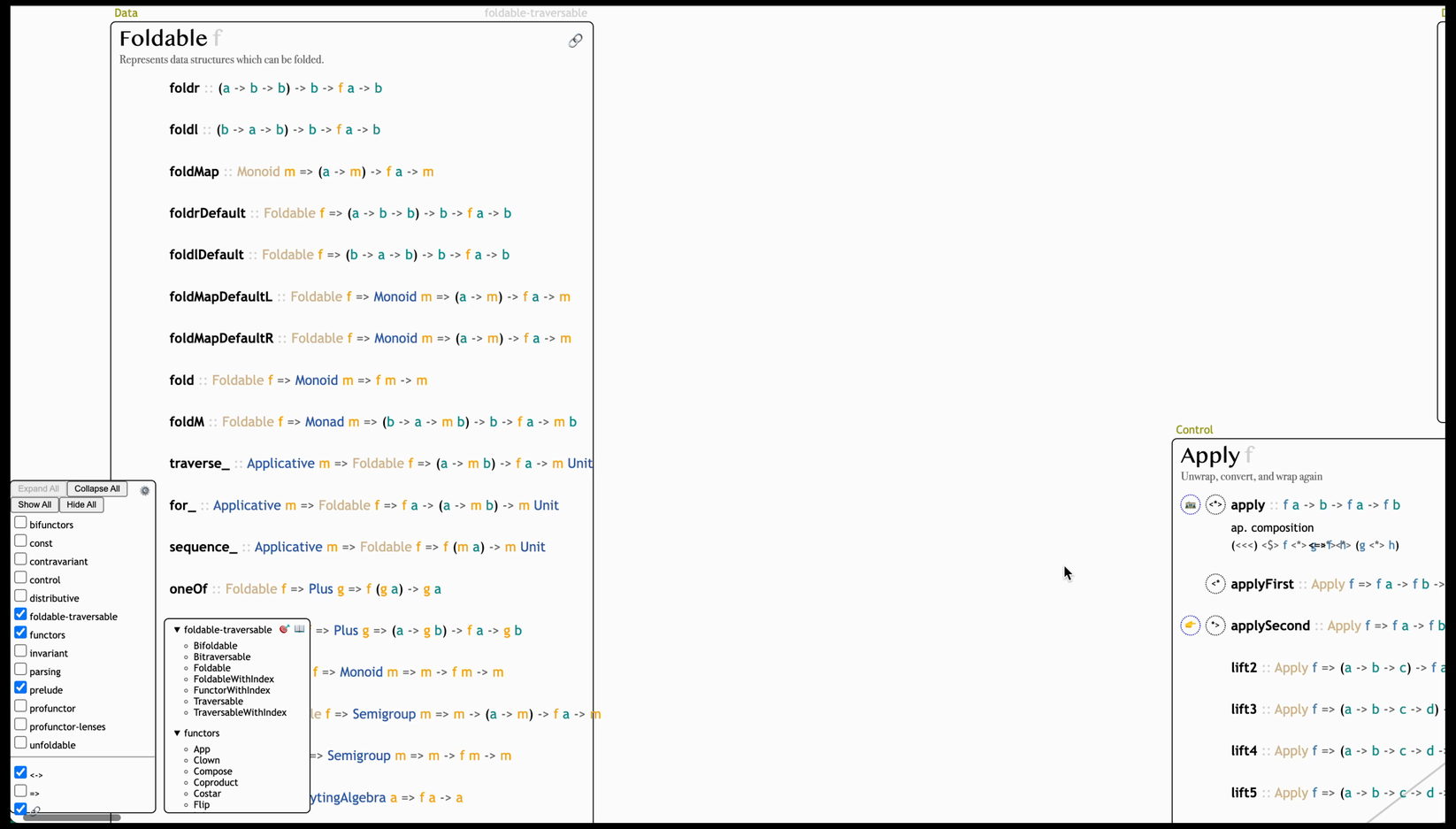




PureScript Typeclasses Cheatsheet

https://imgur.com/a/ESOR7

V2.0



PureScript Typeclasses Cheatsheet

Interactive

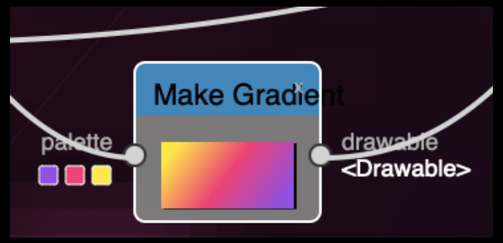
https://github.com/shamansir/purs-typeclasses

Why Free Monads?

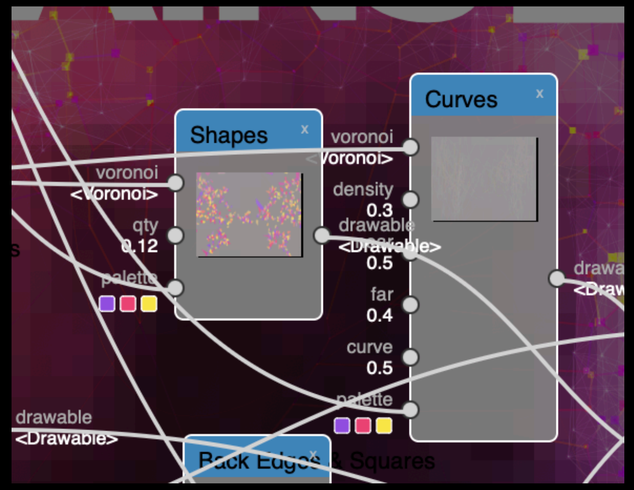
Free Monads
Versus
Monad Transformers

When life gives you le monads,
make them le free!

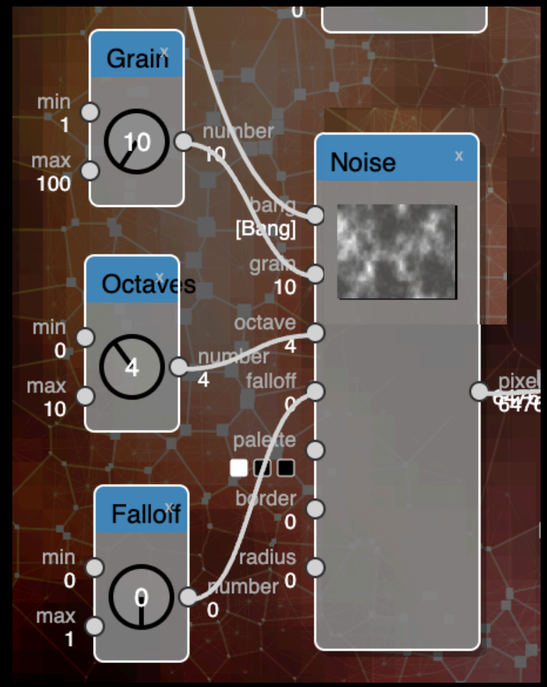
```
noise :: Gen.Fn
noise =
    Gen.fn2v
        "noise"
        ( "scale" ∧ Num 10.0
        +> "offset" ∧ Num 0.1
        +> Vec.empty
        )
        ( Vec.singleton "offset" )
        (\scale offset → Noise { scale, offset })
```



State



Effects



Halogen

https://purescript-halogen.github.io/purescript-halogen/

```purescript
type State = Maybe Number

data Action = Regenerate

component :: forall query input output m. MonadEffect m => H.Component query input output
component =
  H.mkComponent
    { initialState
    , render
    , eval: H.mkEval $ H.defaultEval { handleAction = handleAction }
    }

initialState :: forall input. input -> State
initialState _ = Nothing

render :: forall m. State -> H.ComponentHTML Action () m
render state = do
  let value = maybe "No number generated yet" show state
  HH.div_
    [ HH.h1_
        [ HH.text "Random number" ]
    , HH.p_
        [ HH.text ("Current value: " <> value) ]
    , HH.button
        [ HE.onClick \_ -> Regenerate ]
        [ HH.text "Generate new number" ]
    ]

handleAction :: forall output m. MonadEffect m => Action -> H.HalogenM State Action () ou
handleAction = case _ of
  Regenerate -> do
    newNumber <- H.liftEffect random
    H.modify_ \_ -> Just newNumber
```

Halogen

https://purescript-halogen.github.io/purescript-halogen/

```purescript
main :: Effect Unit
main = runHalogenAff do
  body <- awaitBody
  runUI component unit body

type State =
  { loading :: Boolean
  , username :: String
  , result :: Maybe String
  }

data Action
  = SetUsername String
  | MakeRequest Event

component :: forall query input output m. MonadAff m => H.Component query input output m
component =
  H.mkComponent
    { initialState
    , render
    , eval: H.mkEval $ H.defaultEval { handleAction = handleAction }
    }

initialState :: forall input. input -> State
initialState _ = { loading: false, username: "", result: Nothing }

render :: forall m. State -> H.ComponentHTML Action () m
render st =
  HH.form
    [ HE.onSubmit \ev -> MakeRequest ev ]
    [ HH.h1_ [ HH.text "Look up GitHub user" ]
```

Halogen

https://purescript-halogen.github.io/purescript-halogen/

```
handleAction :: forall output m. MonadAff m => Action -> H.HalogenM State Action () outpu
handleAction = case _ of
  SetUsername username -> do
    H.modify_ _ { username = username, result = Nothing }

  MakeRequest event -> do
    H.liftEffect $ Event.preventDefault event
    username <- H.gets _.username
    H.modify_ _ { loading = true }
    response <- H.liftAff $ AX.get AXRF.string ("https://api.github.com/users/" <> usern
    H.modify_ _ { loading = false, result = map _.body (hush response) }
```

Halogen

https://purescript-halogen.github.io/purescript-halogen/

```purescript
describe "foo" $ do

  it "summing works" $ do
    p ← liftEffect $ Protocol.mkDefault [ "a" ∧ 5, "b" ∧ 3 ]
    let
      fn :: forall m. MonadEffect m ⇒ Fn' String String Unit m Int
      fn =
        Fn.make' "foo" [ "a", "b" ] [ "sum" ] $ do
          a ← Fn.receive "a"
          b ← Fn.receive "b"
          Fn.send "sum" $ a + b
    Fn.run 0 unit p.protocol fn
    p.outputs # shouldContain "sum" 8

  it "summing works with sendIn" $ do
    p ← liftEffect $ Protocol.mkDefault [ "a" ∧ 0, "b" ∧ 0 ]
    let
      fn :: forall m. MonadEffect m ⇒ Fn' String String Unit m Int
      fn =
        Fn.make' "foo" [ "a", "b" ] [ "sum" ] $ do
          Fn.sendIn "a" 6
          Fn.sendIn "b" 7
          a ← Fn.receive "a"
          b ← Fn.receive "b"
          Fn.send "sum" $ a + b
    Fn.run 0 unit p.protocol fn
    p.outputs # shouldContain "sum" 13
    p.inputs # shouldContain "a" 6
    p.inputs # shouldContain "b" 7
```

```haskell
data ProcessF i o state d m a
  = State (state -> a ^ state)
  | Lift (m a)
  | Send' o d a
  | SendIn i d a
  | Receive' i (d -> a)


instance functorProcessF :: Functor m => Functor (ProcessF i o state d m) where
  map f = case _ of
    State k -> State (lmap f <<< k)
    Lift m -> Lift (map f m)
    Receive' iid k -> Receive' iid $ map f k
    Send' oid d next -> Send' oid d $ f next
    SendIn iid d next -> SendIn iid d $ f next


newtype ProcessM i o state d m a = ProcessM (Free (ProcessF i o state d m) a)


derive newtype instance functorProcessM :: Functor (ProcessM i o state d m)
derive newtype instance applyProcessM :: Apply (ProcessM i o state d m)
derive newtype instance applicativeProcessM :: Applicative (ProcessM i o state d m)
derive newtype instance bindProcessM :: Bind (ProcessM i o state d m)
derive newtype instance monadProcessM :: Monad (ProcessM i o state d m)
derive newtype instance semigroupProcessM :: Semigroup a => Semigroup (ProcessM i o state d m a)
derive newtype instance monoidProcessM :: Monoid a => Monoid (ProcessM i o state d m a)
--derive newtype instance bifunctorProcessM :: Bifunctor (ProcessM i o state d m a)
```

Homogenous

Heterogeneous

Functor, Foldable, Traversable…

Array a, List a, Map k v, …

Homogenous

Heterogeneous

Variant (foo :: Int | v), Record (foo :: Int | r), …

HMap, HFlodl, HTraverse…

```
it "function is performed properly" $ do

    node ←
        Node.make _sum unit { a : 2, b : 3 } { sum : 0 }
            $ do
                a ← P.receive (Fn.Input :: Fn.Input "a")
                b ← P.receive (Fn.Input :: Fn.Input "b")
                P.send (Fn.Output :: Fn.Output "sum") $ a + b

    atSum ← node `Node.at_` _.sum
    atSum `shouldEqual` 0

    _ ← Node.run node

    atSumAfter ← node `Node.at_` _.sum
    atSumAfter `shouldEqual` 5

    pure unit
```

```purescript
type Families2 m =
  ( foo :: Family.Def Unit ( foo :: String, bar :: String, c :: Int ) ( out :: Boolean ) m
  , bar :: Family.Def Unit ( a :: String, b :: String, c :: String ) ( x :: Int ) m
  , sum :: Family.Def Unit ( a :: Int, b :: Int ) ( sum :: Int ) m
  )

type Toolkit2 m
  = Toolkit Unit (Families2 m)
```

```purescript
(toolkit2 :: Toolkit2 Aff) =
  Toolkit.from "test2"
    { foo :
        Family.def
          unit
          { foo : "aaa", bar : "bbb", c : 32 }
          { out : false }
          $ Fn.make "foo" $ pure unit
    , bar :
        Family.def
          unit
          { a : "aaa", b : "bbb", c : "ccc" }
          { x : 12 }
          $ Fn.make "bar" $ pure unit
    , sum :
        Family.def
          unit
          { a : 40, b : 2 }
          { sum : 42 }
          $ Fn.make "sumFn" $ pure unit
    }
```

```
, random :
    Family.def
        unit
        { a : 2, b : 40 }
        { random : 30 }
        $ Fn.make "randomFn"
        $ do
            a ← Fn.receive (Fn.Input :: _ "a")
            b ← Fn.receive (Fn.Input :: _ "b")
            rnd ← liftEffect $ randomInt a b
            Fn.send (Fn.Output :: _ "random") rnd
}
```

```
data ProcessF :: forall is' os'. Type → Row is' → Row os' → (Type → Type) → Type → Type
data ProcessF state is os m a
  = State (state → a ∧ state)
  | Lift (m a)
  | Send' OutputR (forall dout. dout) a
  | SendIn InputR (forall din. din) a
  | Receive' InputR (forall din. din → a)


instance functorProcessF :: Functor m ⇒ Functor (ProcessF state is os m) where
  map f = case _ of
    State k → State (lmap f <<< k)
    Lift m → Lift (map f m)
    Receive' iid k → Receive' iid (map f k)
    Send' oid d next → Send' oid d $ f next
    SendIn iid d next → SendIn iid d $ f next


newtype ProcessM :: forall is' os'. Type → Row is' → Row os' → (Type → Type) → Type → Type
newtype ProcessM state is os m a = ProcessM (Free (ProcessF state is os m) a)


derive newtype instance functorProcessM :: Functor (ProcessM state is os m)
derive newtype instance applyProcessM :: Apply (ProcessM state is os m)
derive newtype instance applicativeProcessM :: Applicative (ProcessM state is os m)
derive newtype instance bindProcessM :: Bind (ProcessM state is os m)
derive newtype instance monadProcessM :: Monad (ProcessM state is os m)
derive newtype instance semigroupProcessM :: Semigroup a ⇒ Semigroup (ProcessM state is os m a)
derive newtype instance monoidProcessM :: Monoid a ⇒ Monoid (ProcessM state is os m a)
--derive newtype instance bifunctorProcessM :: Bifunctor (ProcessM state is os m a)
```

```
receive :: forall i state is is' os din m. IsSymbol i ⇒ Cons i din is' is ⇒ Input i → ProcessM state is os m din
receive iid = ProcessM $ Free.liftF $ Receive' (inputR iid) (unsafeCoerce {-identity-})


send :: forall o state is os os' dout m. IsSymbol o ⇒ Cons o dout os' os ⇒ Output o → dout → ProcessM state is os m Unit
send oid d =
  ProcessM $ Free.liftF $ Send' (outputR oid) (unsafeCoerce d) unit


sendIn :: ∀ i din state is is' os m. IsSymbol i ⇒ Cons i din is' is ⇒ Input i → din → ProcessM state is os m Unit
sendIn iid d =
  ProcessM $ Free.liftF $ SendIn (inputR iid) (unsafeCoerce d) unit


lift :: forall state is os m. m Unit → ProcessM state is os m Unit
lift m = ProcessM $ Free.liftF $ Lift m


inputsOf :: forall rl is. RL.RowToList is rl ⇒ Keys rl ⇒ Record is → List String
inputsOf = keys


outputsOf :: forall rl os. RL.RowToList os rl ⇒ Keys rl ⇒ Record os → List String
outputsOf = keys
```

```
runM :: forall i o state d m. MonadEffect m ⇒ MonadRec m ⇒ Ord i ⇒ Protocol i o d → d → Ref state → ProcessM i o state d m ↝ m
runM protocol default stateRef (ProcessM processFree) =
  runFreeM protocol default stateRef processFree


runFreeM :: forall i o state d m. MonadEffect m ⇒ MonadRec m ⇒ Ord i ⇒ Protocol i o d → d → Ref state → Free (ProcessF i o state d m) ↝ m
runFreeM protocol default stateRef fn =
  Free.runFreeM go fn
  where
    go (State f) = do
      state ← getUserState
      case f state of
        next /\ nextState → do
          writeUserState nextState
          pure next
    go (Lift m) = m
    go (Receive' iid getV) = do
      maybeVal ← liftEffect $ protocol.receive iid
      pure
        $ getV
        $ Maybe.fromMaybe default
        $ maybeVal
    go (Send' output v next) = do
      liftEffect $ protocol.send output v
      pure next
    go (SendIn input v next) = do
      liftEffect $ protocol.sendIn input v
      pure next

    getUserState = liftEffect $ Ref.read stateRef
    writeUserState nextState = liftEffect $ Ref.write nextState stateRef
```

Noodle

https://github.com/shamansir/noodle

The End

Thank you!

The Next Meetup

16 of March