SIMON THOMPSON

LEARNING FUNCTIONAL PROGRAMMING

The essence of functional programming

Beginning to learn functional programming

Resources, advice and bottlenecks

THE ESSENCE OF FUNCTIONAL PROGRAMMING

LISP

Elm

Miranda

Erlang

Idris

Haskell

Scala

F#

Elixir

OCaml

higher-order
functions

pattern
matching

data
types

lambdas

recursion

higher-order
functions

pattern
matching

data
types

lambdas

recursion

lenses

dependent types

reactive

higher-order functions

DSLs

lazy

pattern matching

data types

monoids

types,
types,
types,
. . .

lambdas

recursion

effects

monads

immutability

fun

Model the world as data

+

Functions over the data

+

Functions as data

And when I say “function” . . .

I mean in the mathematical sense, taking inputs to outputs, and doing **nothing else**!

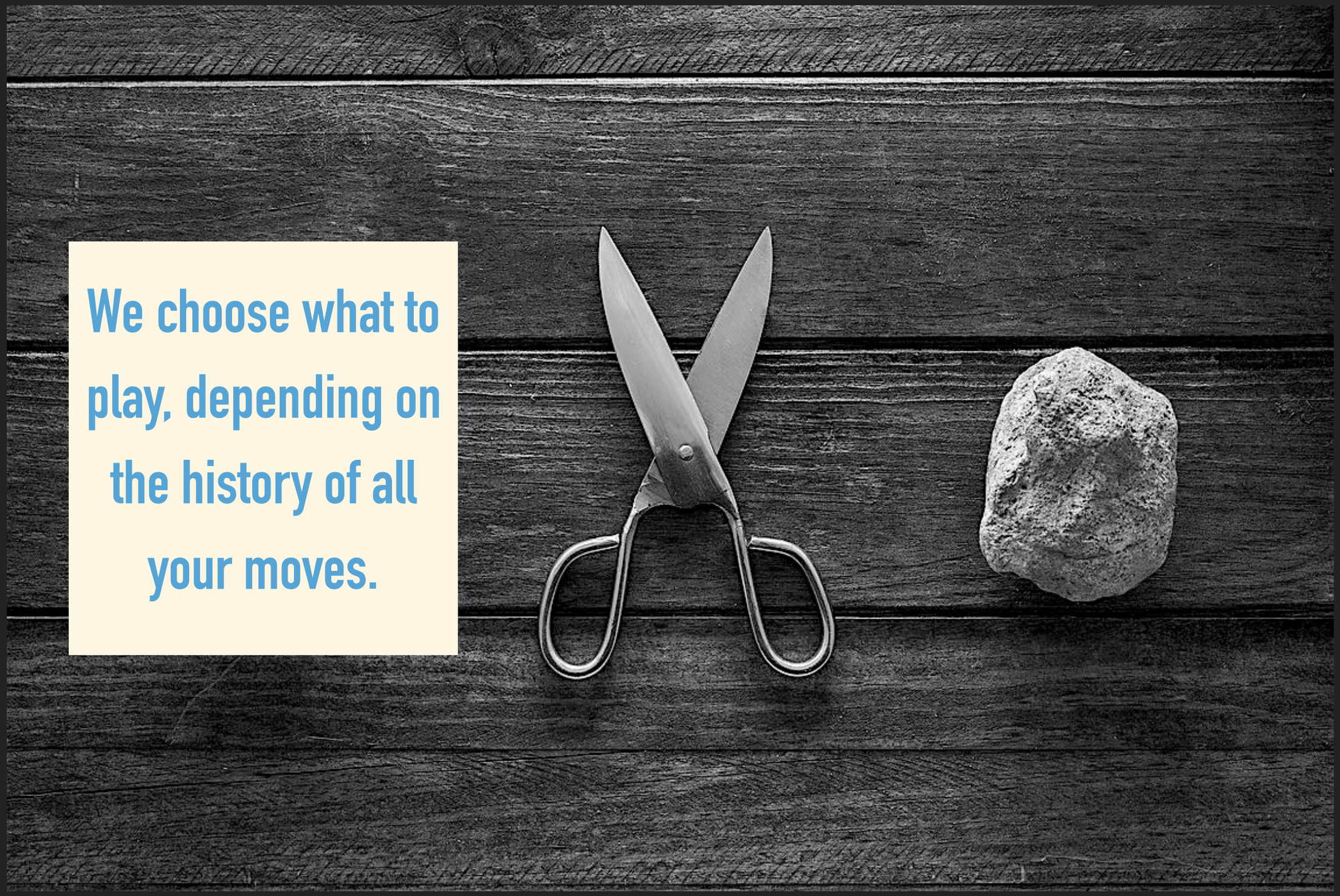
The approach underlies lots of examples,

The approach underlies lots of examples,

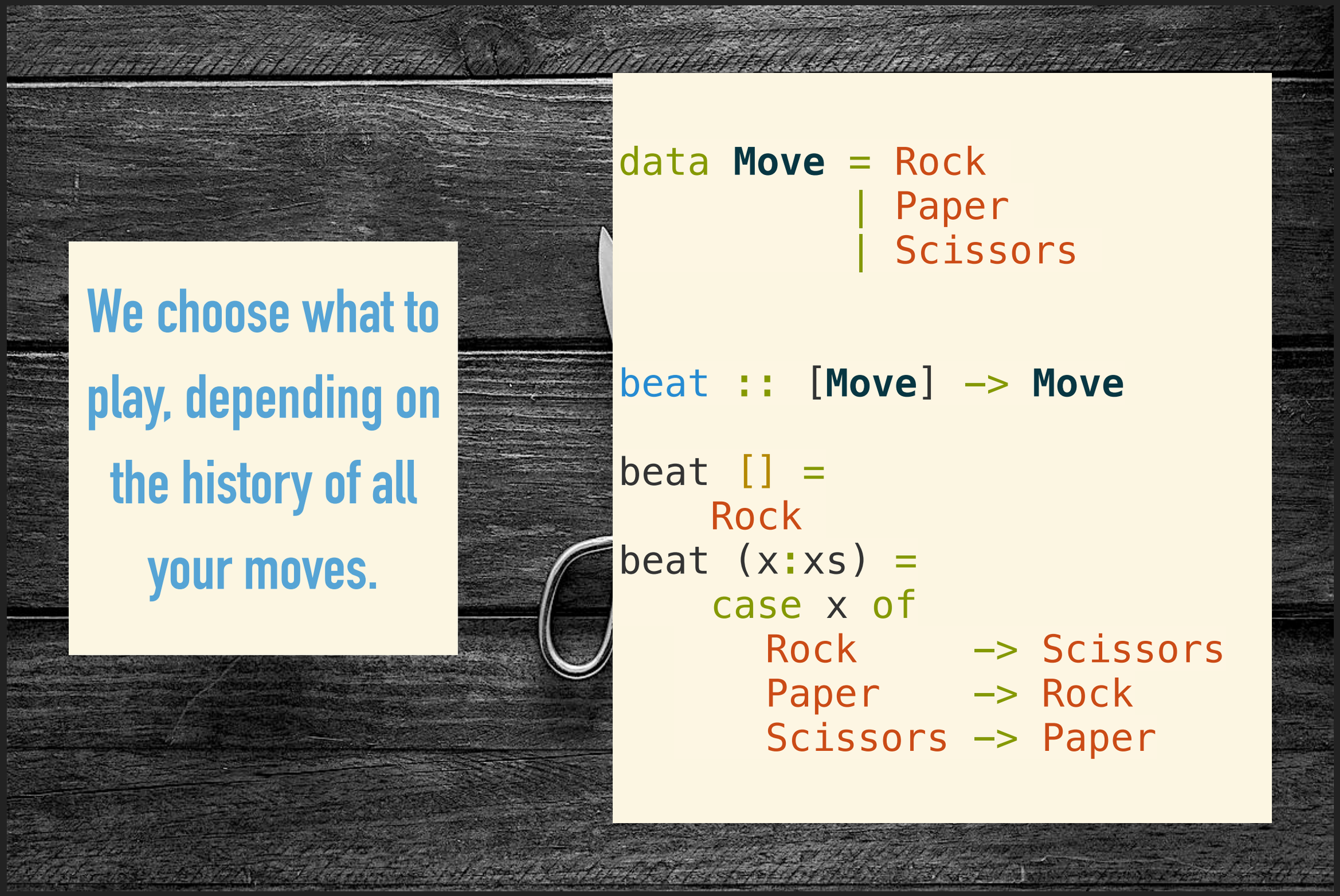
libraries, laziness, monads, lenses . . .

Rock–Paper–Scissors

Original image: https://www.thishopeanchors.com/single-post/2017/04/06/Rock-Paper-Scissors

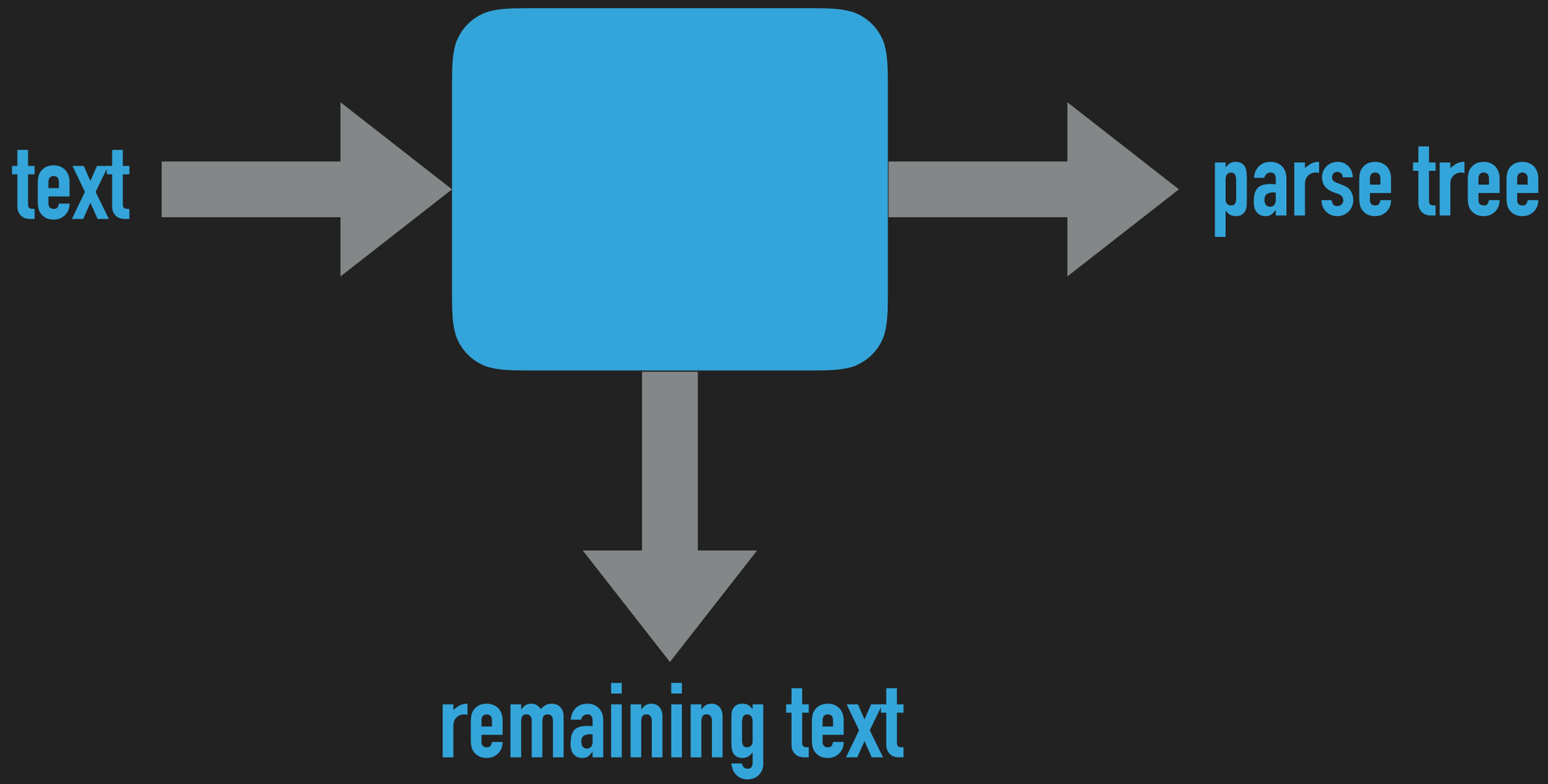


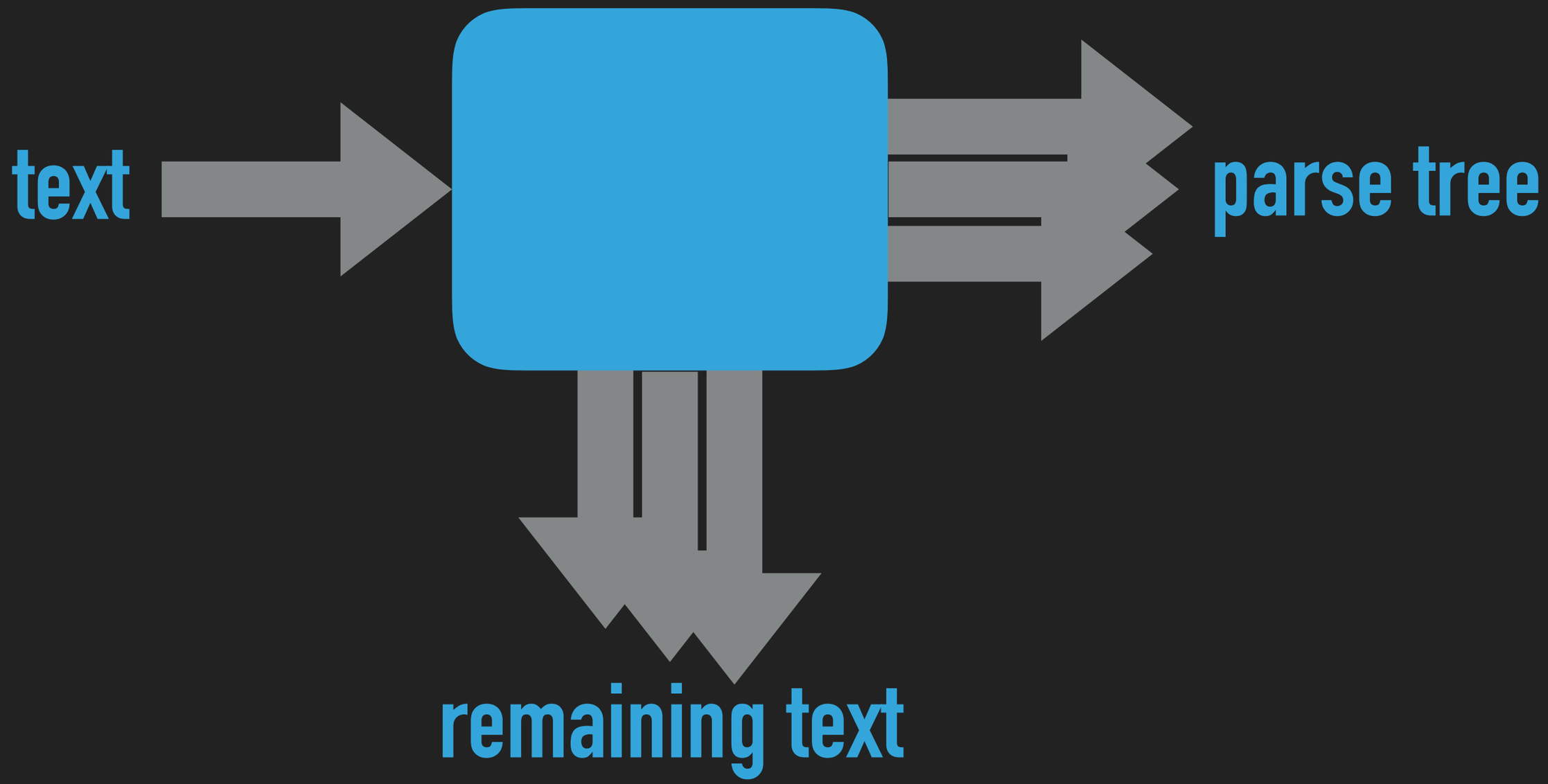
Original image: https://www.thishopeanchors.com/single-post/2017/04/06/Rock-Paper-Scissors



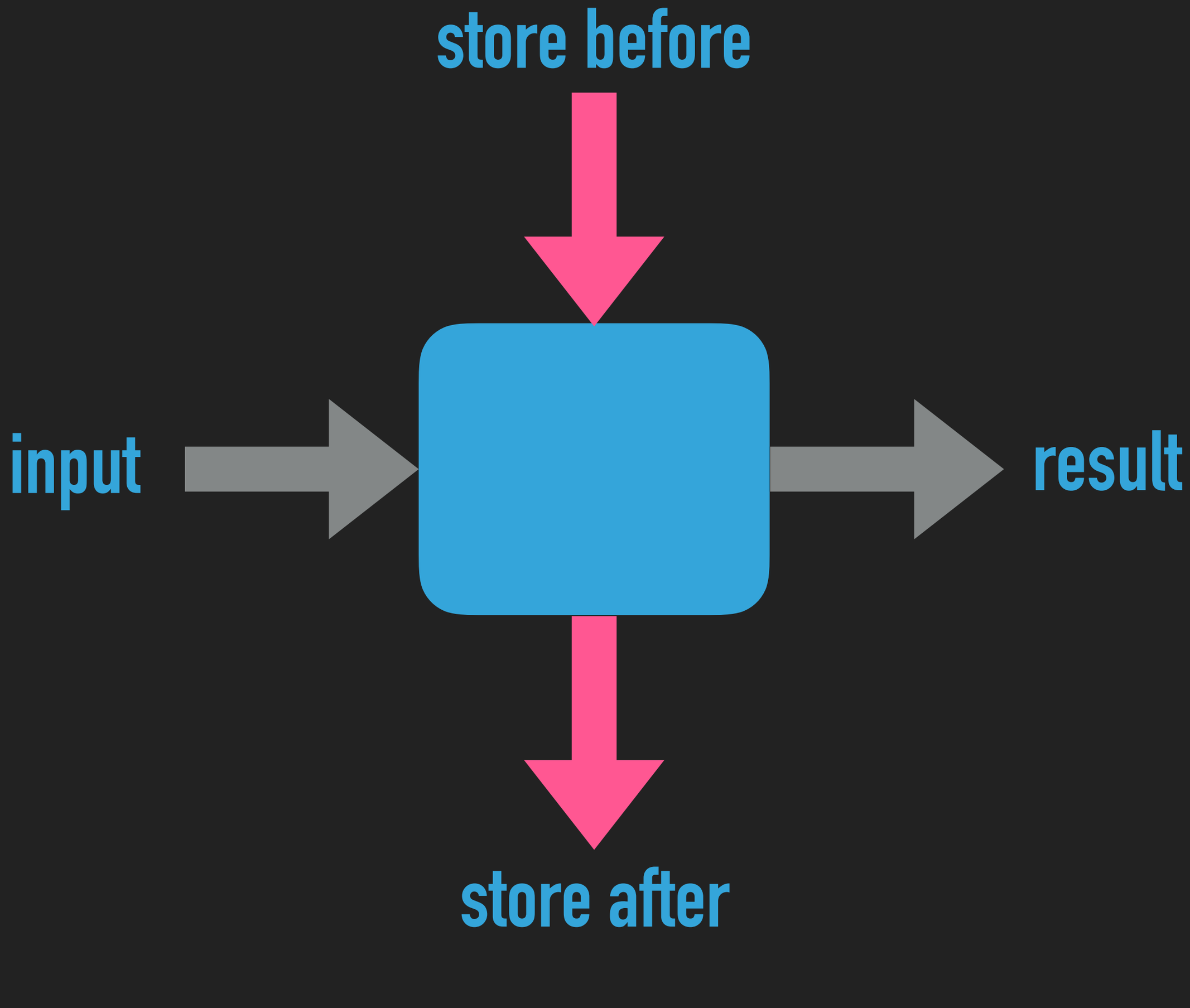
Original image: https://www.thishopeanchors.com/single-post/2017/04/06/Rock-Paper-Scissors

Parsers


text
parse tree
remaining text

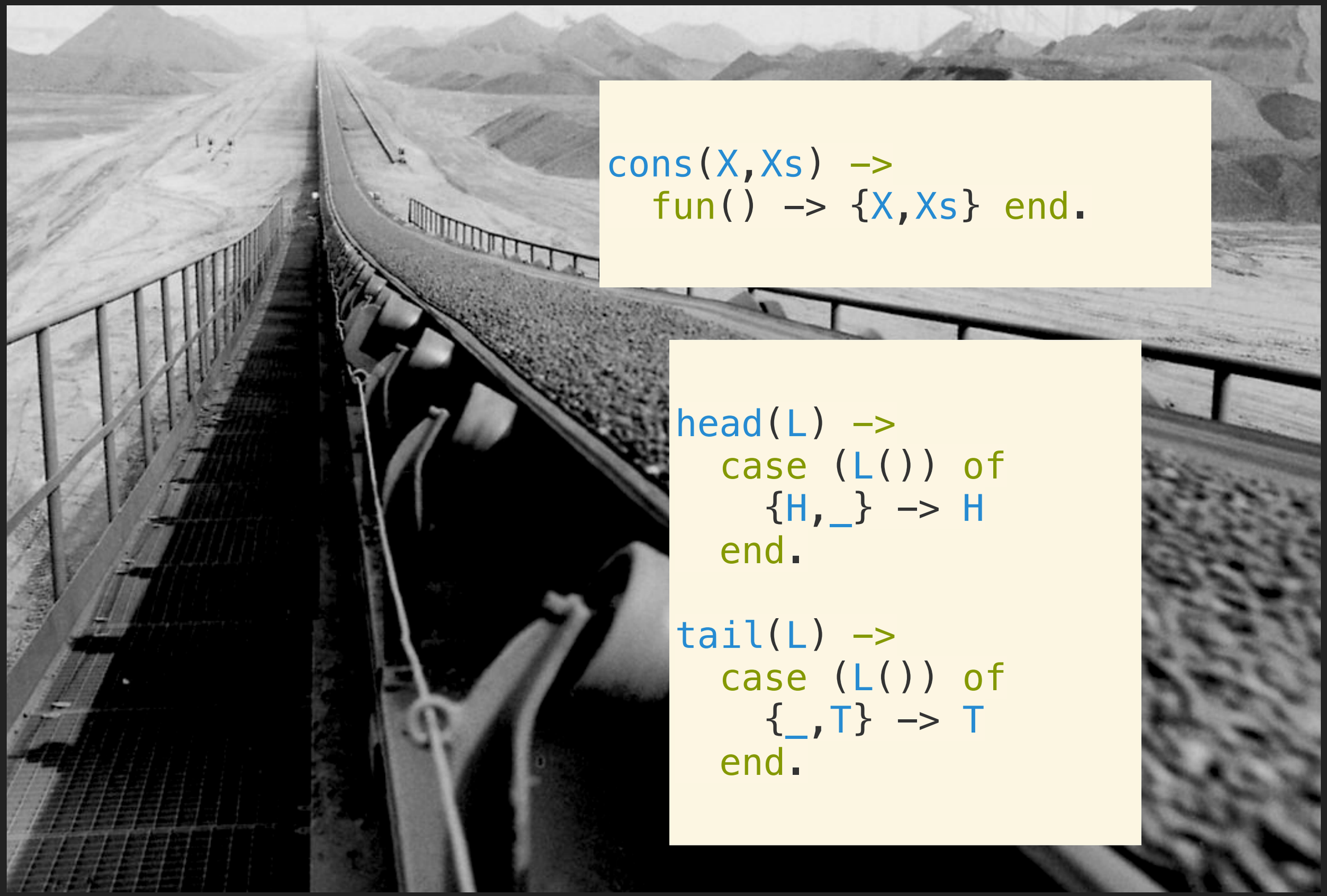

text
parse tree
remaining text

Side-effects


store before
input
result
store after

Delay/force . . . streams

Original image: http://www.metso.com/services/spare-wear-parts-conveyors/conveyor-belts/

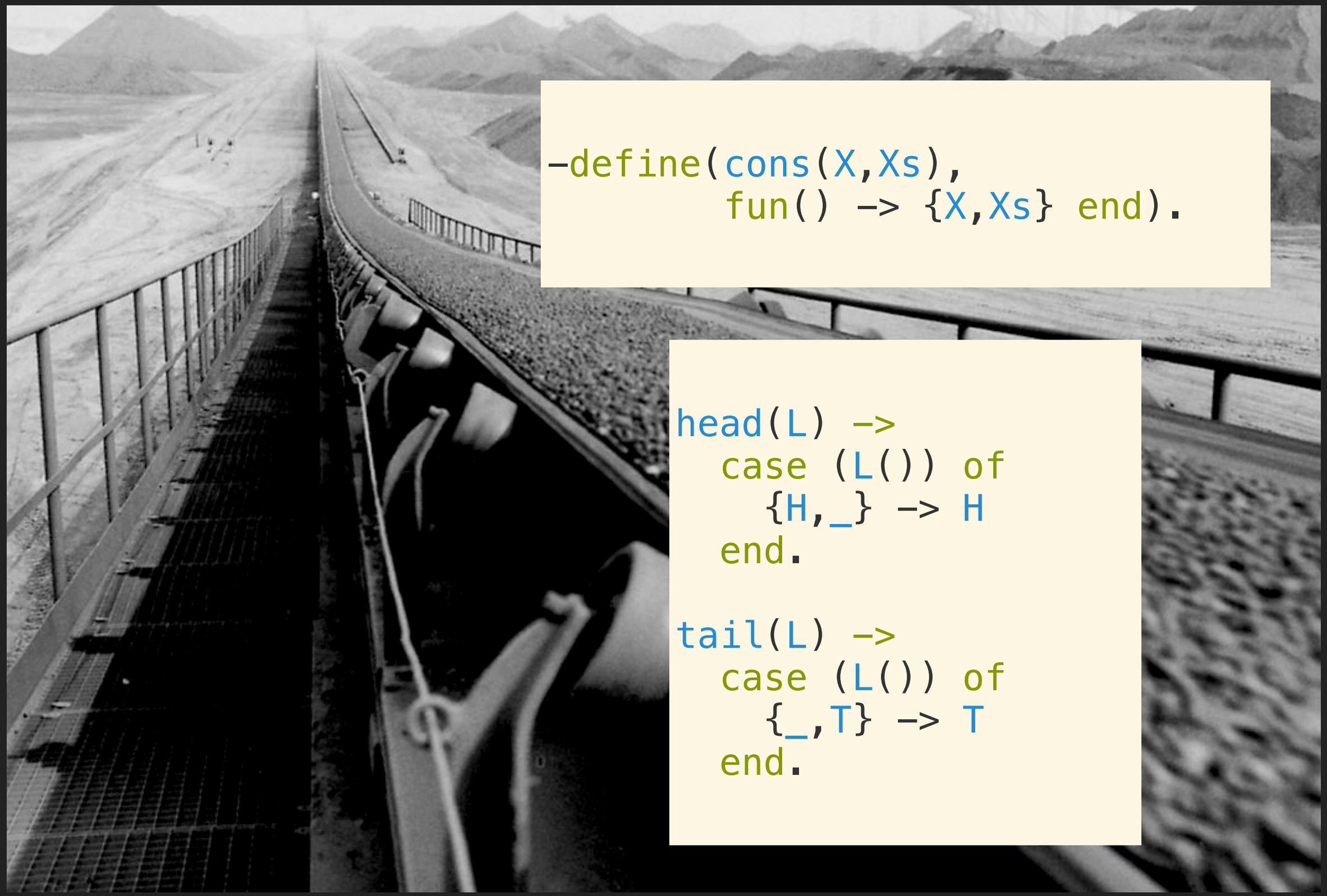


```
cons(X,Xs) ->
  fun() -> {X,Xs} end.
```

```
head(L) ->
  case (L()) of
    {H,_} -> H
  end.

tail(L) ->
  case (L()) of
    {_,T} -> T
  end.
```

Original image: http://www.metso.com/services/spare-wear-parts-conveyors/conveyor-belts/



```
-define(cons(X,Xs),
        fun() -> {X,Xs} end).
```

```
head(L) ->
  case (L()) of
    {H,_} -> H
  end.

tail(L) ->
  case (L()) of
    {_,T} -> T
  end.
```

Original image: http://www.metso.com/services/spare-wear-parts-conveyors/conveyor-belts/

github.com/simonjohnthompson/streams

github.com/simonjohnthompson/Interaction

type

Functions give us expressivity

+

Types help to constrain that

+

Give a language for modelling

BEGINNING FUNCTIONAL PROGRAMMING

higher-order
functions

data

types

lambdas

higher-order
functions

pattern matching

data types

lambdas

recursion

Pick any language

Start with the concrete before going to complex abstractions.

PATTERN MATCHING

```haskell
type Point = (Float,Float)

data Shape = Circle Point Float
           | Rectangle Point Float Float

area :: Shape -> Float

area (Circle _ r)       = pi*r*r
area (Rectangle _ h w) = h*w
```

```haskell
type Point = (Float,Float)

data Shape = Circle Point Float
           | Rectangle Point Float Float

area :: Shape -> Float

area (Circle _ r)       = pi*r*r
area (Rectangle _ h w) = h*w
```

```erlang
area({circle,_,R})     -> math:pi()*R*R;
area({rectangle,H,W}) -> H*W.
```

```haskell
type Point = (Float,Float)

data Shape = Circle Point Float
           | Rectangle Point Float Float

area :: Shape -> Float

area (Circle _ r)       = pi*r*r
area (Rectangle _ h w) = h*w
```

```erlang
-type point() :: {float(),float()}.

-type shape() :: {circle,point(),float()} |
                 {rectangle,point(),float(),float()}.

-spec area(shape()) -> float().

area({circle,_,R})    -> math:pi()*R*R;
area({rectangle,H,W}) -> H*W.
```

PATTERN MATCHING!

```haskell
area :: Shape -> Float

area (Circle _ r)       = pi*r*r
area (Rectangle _ h w) = h*w
```

```haskell
area :: Shape -> Float

area (Circle _ r)       = pi*r*r
area (Rectangle _ h w) = h*w
```

Link to something more familiar

```haskell
area :: Shape -> Float

area (Circle _ r)       = pi*r*r
area (Rectangle _ h w) = h*w
```

```haskell
area shape =
    if is_circle shape
        then pi*(radius shape)*(radius shape)
        else height shape * width shape
```

```haskell
area :: Shape -> Float

area (Circle _ r)       = pi*r*r
area (Rectangle _ h w) = h*w
```

```haskell
area shape =
    if is_circle shape
        then pi*(radius shape)*(radius shape)
        else height shape * width shape

is_circle :: Shape -> Bool

is_circle (Circle _ _) = True
is_circle _            = False

radius, height, width :: Shape -> Float

radius (Circle _ r)       = r
height (Rectangle _ h _) = h
width  (Rectangle _ _ w) = w
```

RECURSION

The nub of a list is the list with all duplicates removed.

```haskell
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
        then nub xs
        else x : nub xs
```

The nub of a list is the list with all duplicates removed.

```
nub([]) -> [];

nub([X|Xs]) ->
    [X|nub(remove(X,Xs))].
```

The nub of a list is the list with all duplicates removed.

```
nub([]) -> [];

nub([X|Xs]) ->
    [X|nub(remove(X,Xs))].

remove(_,[]) -> [];

remove(X,[X|Xs]) ->
    remove(X,Xs);

remove(X,[Y|Xs]) ->
    [Y|remove(X,Xs)].
```

RECURSION!

The nub of a list is the list with all duplicates removed.

```haskell
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
        then nub xs
        else x : nub xs
```

Generate all the answers?

```haskell
nub []  = []
```

The nub of a list is the list with all duplicates removed.

```haskell
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
        then nub xs
        else x : nub xs
```

Generate all the answers?

```haskell
nub []  = []

nub [1] = nub (1:[]) = 1 : nub [] = [1]
```

The nub of a list is the list with all duplicates removed.

```haskell
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
        then nub xs
        else x : nub xs
```

Generate all the answers?

```
nub []  = []

nub [1] = nub (1:[]) = 1 : nub [] = [1]

nub [2,1] = nub (2:[1]) = 2 : nub [1] = [2,1]
```

The nub of a list is the list with all duplicates removed.

```haskell
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
        then nub xs
        else x : nub xs
```

Generate all the answers?

```
nub []  = []

nub [1] = nub (1:[]) = 1 : nub [] = [1]

nub [2,1] = nub (2:[1]) = 2 : nub [1] = [2,1]

nub [1,2,1] = nub (1:[2,1]) = nub [2,1] = [2,1]
```

The nub of a list is the list with all duplicates removed.

```haskell
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
        then nub xs
        else x : nub xs
```

Rewrite . . . work “top down”

```haskell
nub [1,2,1]
```

The nub of a list is the list with all duplicates removed.

```haskell
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
        then nub xs
        else x : nub xs
```

Rewrite . . . work “top down”

```haskell
nub [1,2,1]
= nub (1:[2,1])
```

The nub of a list is the list with all duplicates removed.

```haskell
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
        then nub xs
        else x : nub xs
```

Rewrite . . . work “top down”

```
nub [1,2,1]
= nub (1:[2,1])
= nub [2,1]
```

The nub of a list is the list with all duplicates removed.

```haskell
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
        then nub xs
        else x : nub xs
```

Rewrite . . . work “top down”

```
nub [1,2,1]
= nub (1:[2,1])
= nub [2,1]
= nub (2:[1])
```

The nub of a list is the list with all duplicates removed.

```haskell
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
        then nub xs
        else x : nub xs
```

Rewrite . . . work “top down”

```
nub [1,2,1]
= nub (1:[2,1])
= nub [2,1]
= nub (2:[1])
= 2 : nub [1]
```

The nub of a list is the list with all duplicates removed.

```haskell
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
        then nub xs
        else x : nub xs
```

Rewrite . . . work “top down”

```
nub [1,2,1]
= nub (1:[2,1])
= nub [2,1]
= nub (2:[1])
= 2 : nub [1]
= 2 : nub (1:[])
```

The nub of a list is the list with all duplicates removed.

```haskell
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
        then nub xs
        else x : nub xs
```

Rewrite . . . work “top down”

```
nub [1,2,1]
= nub (1:[2,1])
= nub [2,1]
= nub (2:[1])
= 2 : nub [1]
= 2 : nub (1:[])
= 2 : 1 : nub []
```

The nub of a list is the list with all duplicates removed.

```haskell
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
        then nub xs
        else x : nub xs
```

Rewrite . . . work “top down”

```
nub [1,2,1]
= nub (1:[2,1])
= nub [2,1]
= nub (2:[1])
= 2 : nub [1]
= 2 : nub (1:[])
= 2 : 1 : nub []
= 2 : 1 : []
```

The nub of a list is the list with all duplicates removed.

```haskell
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
        then nub xs
        else x : nub xs
```

Rewrite . . . work “top down”

```
nub [1,2,1]
= nub (1:[2,1])
= nub [2,1]
= nub (2:[1])
= 2 : nub [1]
= 2 : nub (1:[])
= 2 : 1 : nub []
= 2 : 1 : []
= 2 :[1]
```

The nub of a list is the list with all duplicates removed.

```haskell
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
        then nub xs
        else x : nub xs
```

Rewrite . . . work “top down”

```
nub [1,2,1]
= nub (1:[2,1])
= nub [2,1]
= nub (2:[1])
= 2 : nub [1]
= 2 : nub (1:[])
= 2 : 1 : nub []
= 2 : 1 : []
= 2 :[1]
= [2,1]
```

The nub of a list is the list with all duplicates removed.

```haskell
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
        then nub xs
        else x : nub xs
```

Accept the template: the lists get shorter . . .

```haskell
foo [] = …
foo (x:xs) = … x … foo xs …
```

The nub of a list is the list with all duplicates removed.

```haskell
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
        then nub xs
        else x : nub xs
```

Accept the template: the lists get shorter . . .

```haskell
foo [] = …
foo (x:xs) = … x … foo xs …
```

. . . and look at some examples

```haskell
nub [1,2] = [1,2]
nub [2,1,2] = [1,2]
```

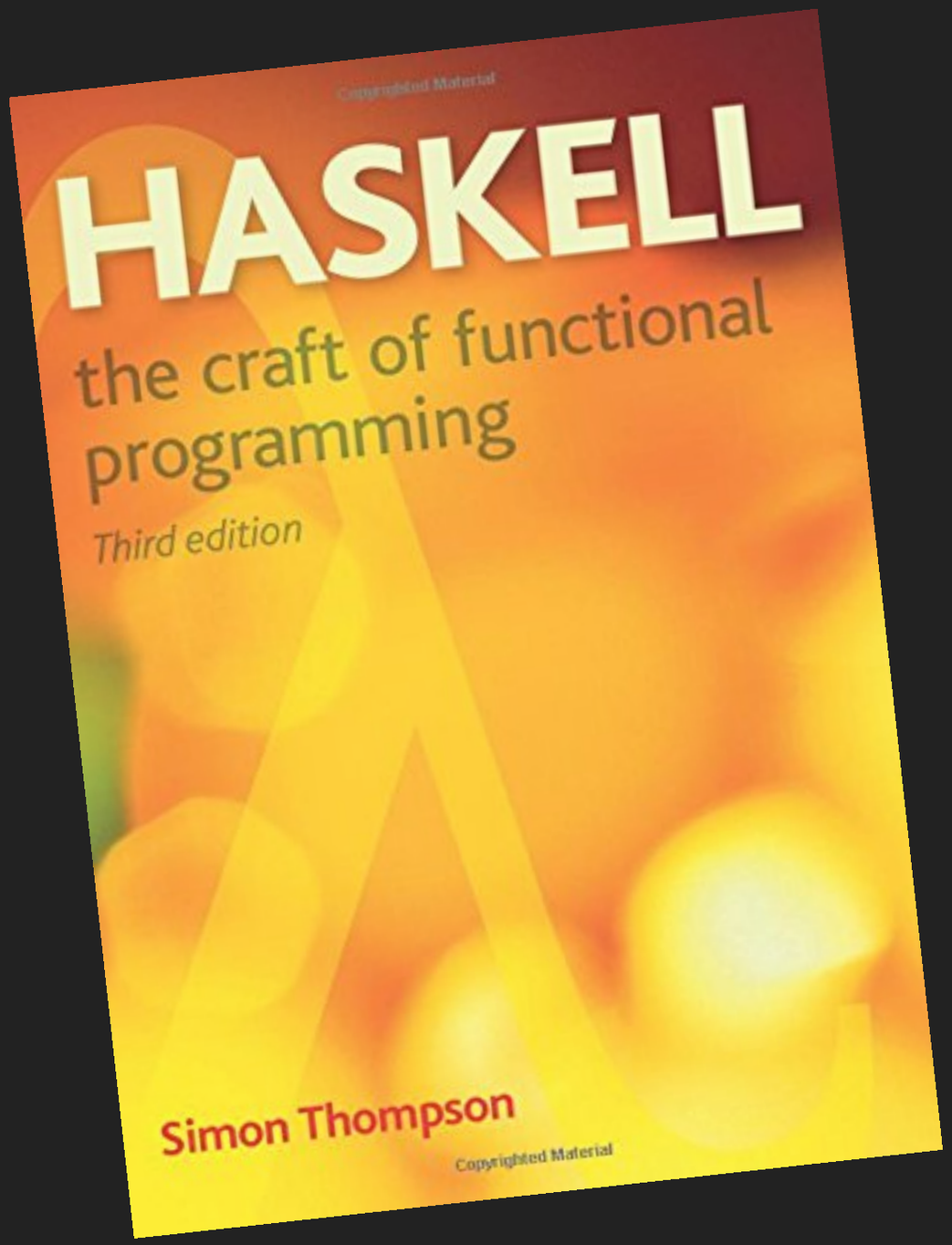
How to get started (with recursion)?

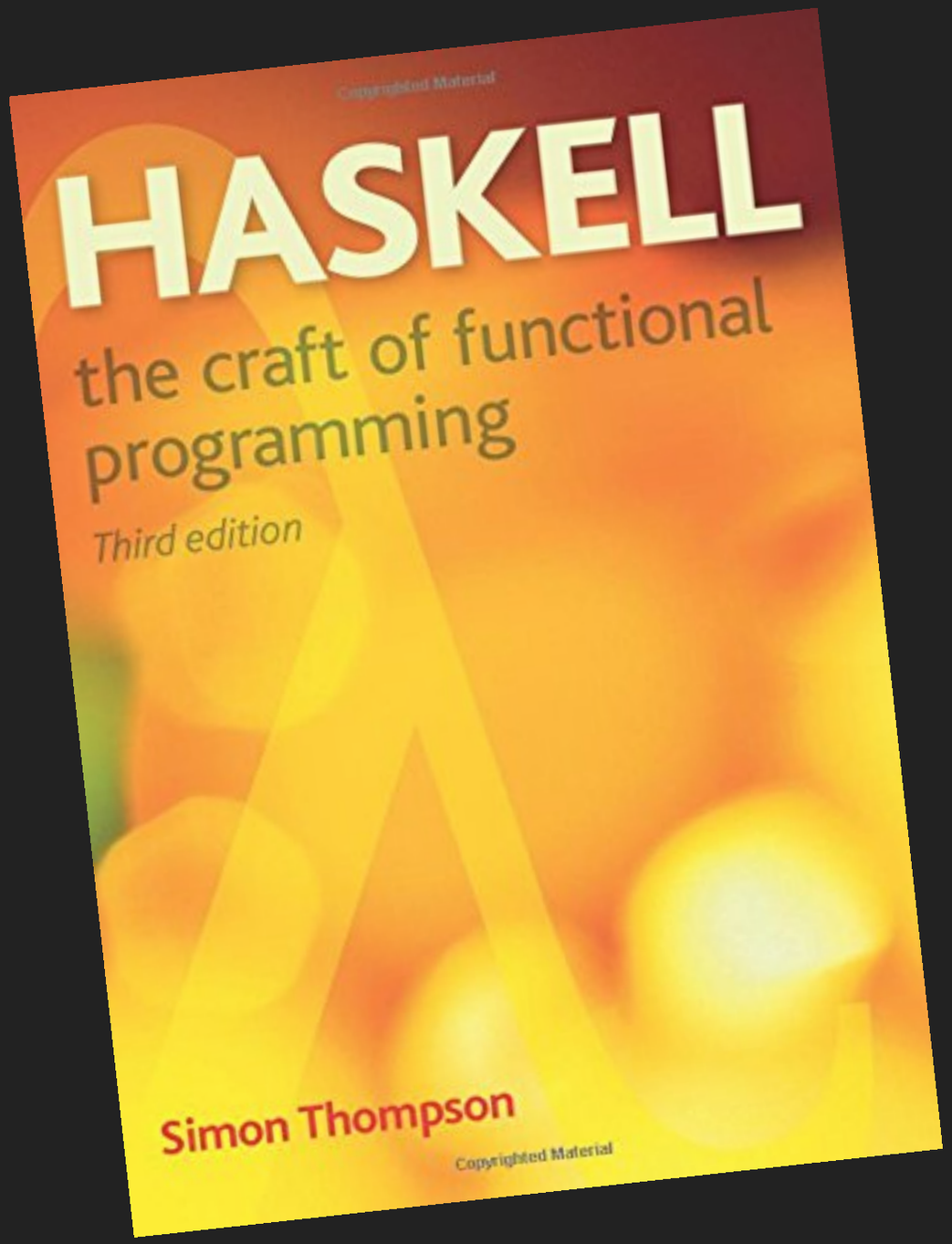
Examples, examples, examples.

From simple “five finger” exercises,

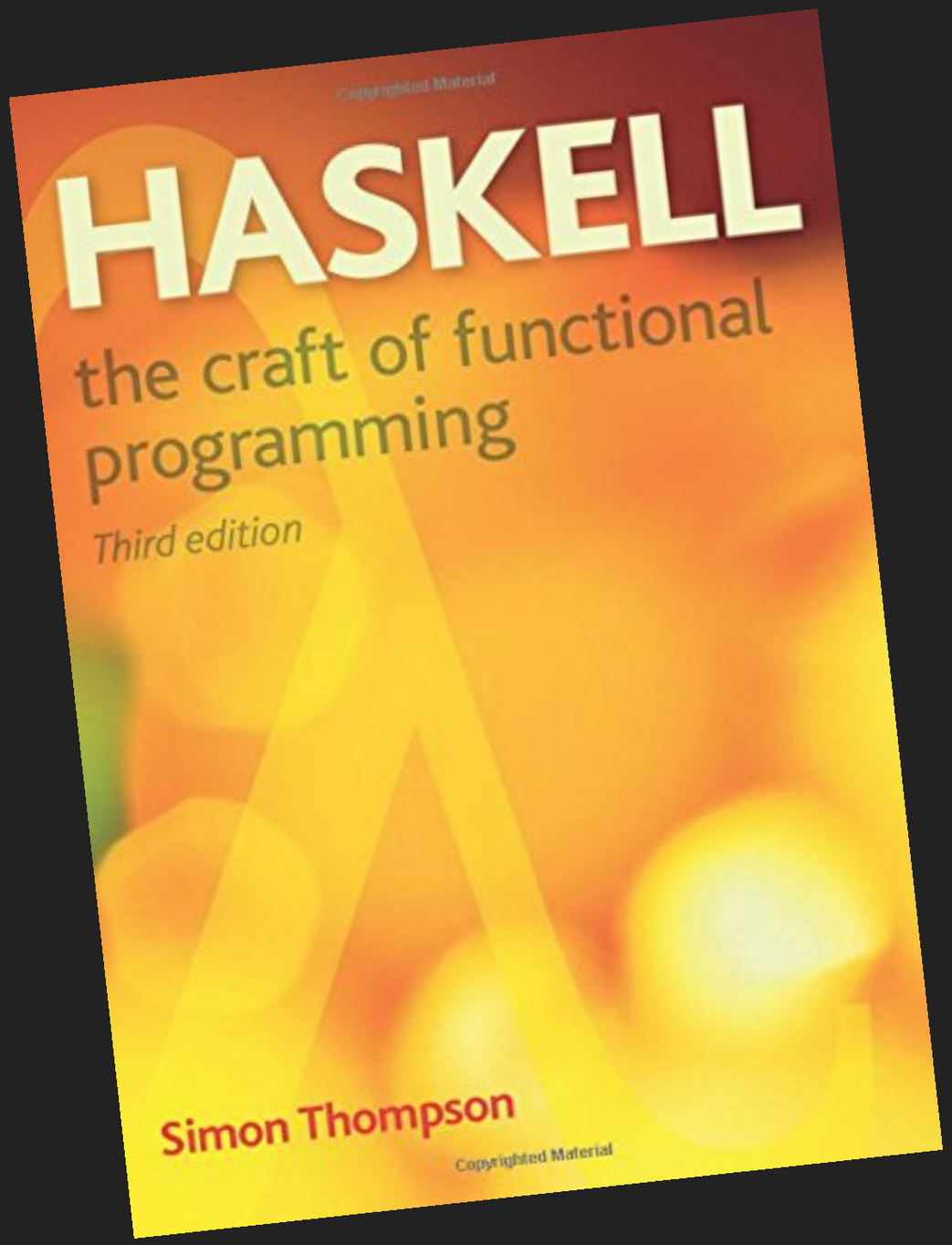
to a favourite small library.

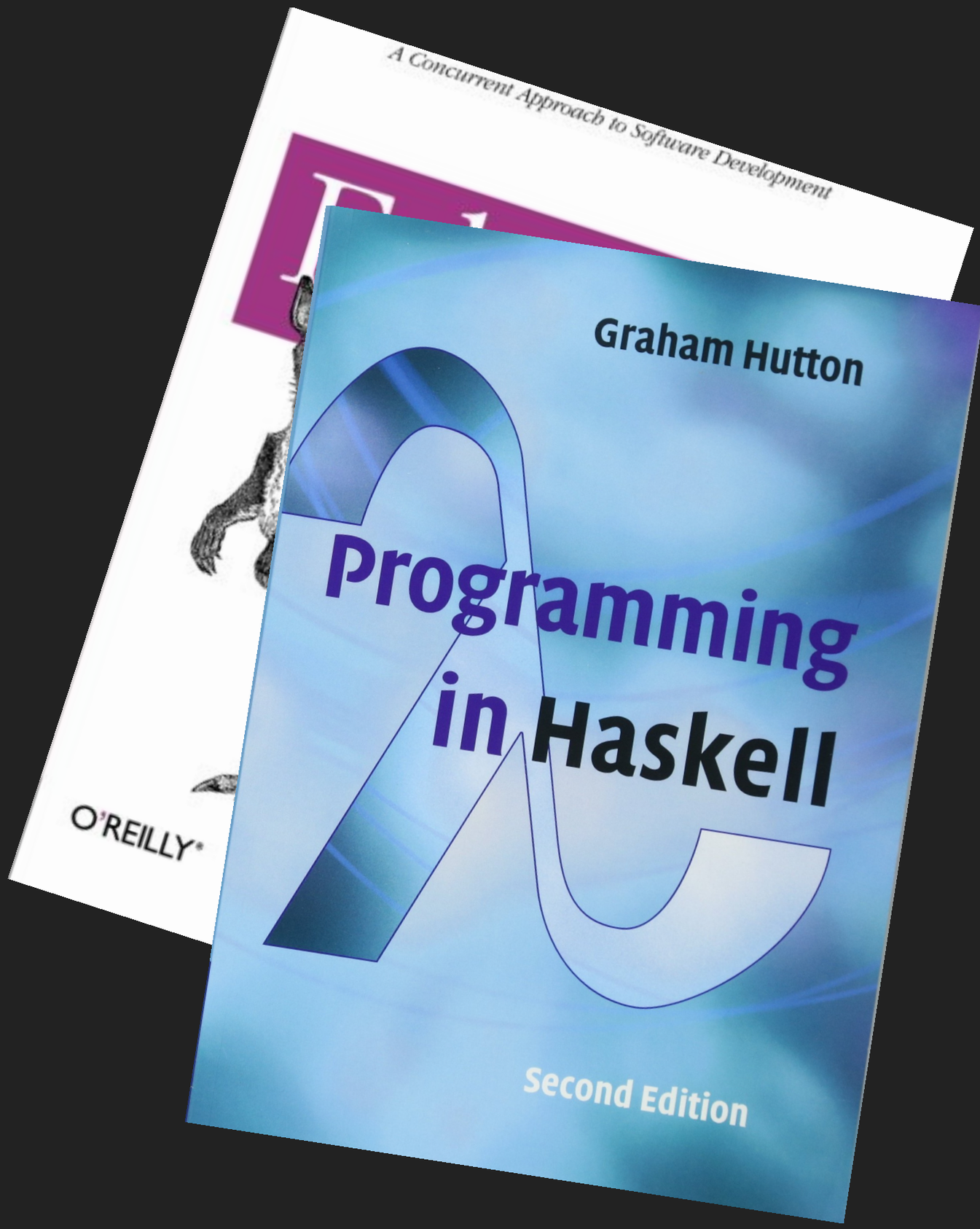
RESOURCES


Copyrighted Material
HASKELL
the craft of functional programming
Third edition
Simon Thompson
Copyrighted Material


Copyrighted Material
HASKELL
the craft of functional programming
Third edition
Simon Thompson
Copyrighted Material


A Concurrent Approach to Software Development
Erlang
Programming
O'REILLY
Francesco Cesarini
& Simon Thompson


Copyrighted Material
HASKELL
the craft of functional programming
Third edition
Simon Thompson
Copyrighted Material


A Concurrent Approach to Software Development
O'REILLY®
Graham Hutton
Programming in Haskell
Second Edition

The Pragmatic Programmers

Programming Erlang

Software for a Concurrent World

Second Edition

Joe Armstrong

Edited by Susannah Davidson Pfalzer

A Concurrent Approach to Software Development

Graham Hutton

Programming in Haskell

Second Edition

Code You Can Believe In

Real World

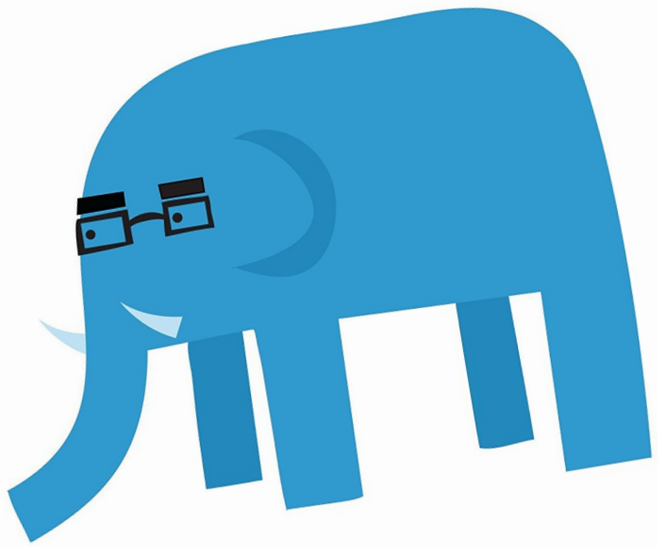
Haskell

O'REILLY®

Bryan O'Sullivan,
John Goerzen & Don Stewart

Learn You a Haskell for Great Good!

A Beginner's Guide



no starch press

Miran Lipovača

Code You Can Believe In

Real World

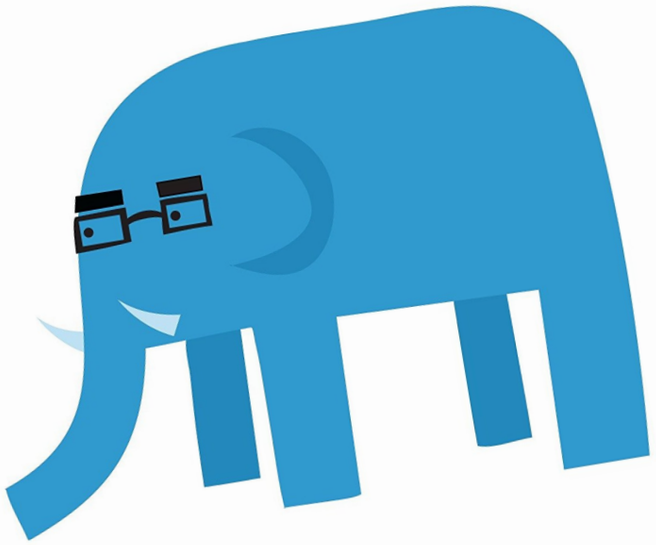
Haskell

O'REILLY®

Bryan O'Sullivan,
John Goerzen & Don Stewart

Learn You a Haskell for Great Good!

A Beginner's Guide



Miran Lipovača

no starch press

Code You Can Believe In

Learn You Some Erlang for Great Good!

A Beginner's Guide

Fred Hébert

Foreword by Joe Armstrong

no starch press

Bryan O'Sullivan,
erzen & Don Stewart

Online

Wiki-books, MOOCs, video channels, try-XXX, tutorials . . .


PROPERTY OF

Working together

XXX-bridge, code clubs,
meet-ups, reading groups...

AND THEN . . .

Pick any language

Start with the concrete before going to complex abstractions.

Choosing a language

LISP

Elm

Miranda

Erlang

Idris

Haskell

Scala

F#

Elixir

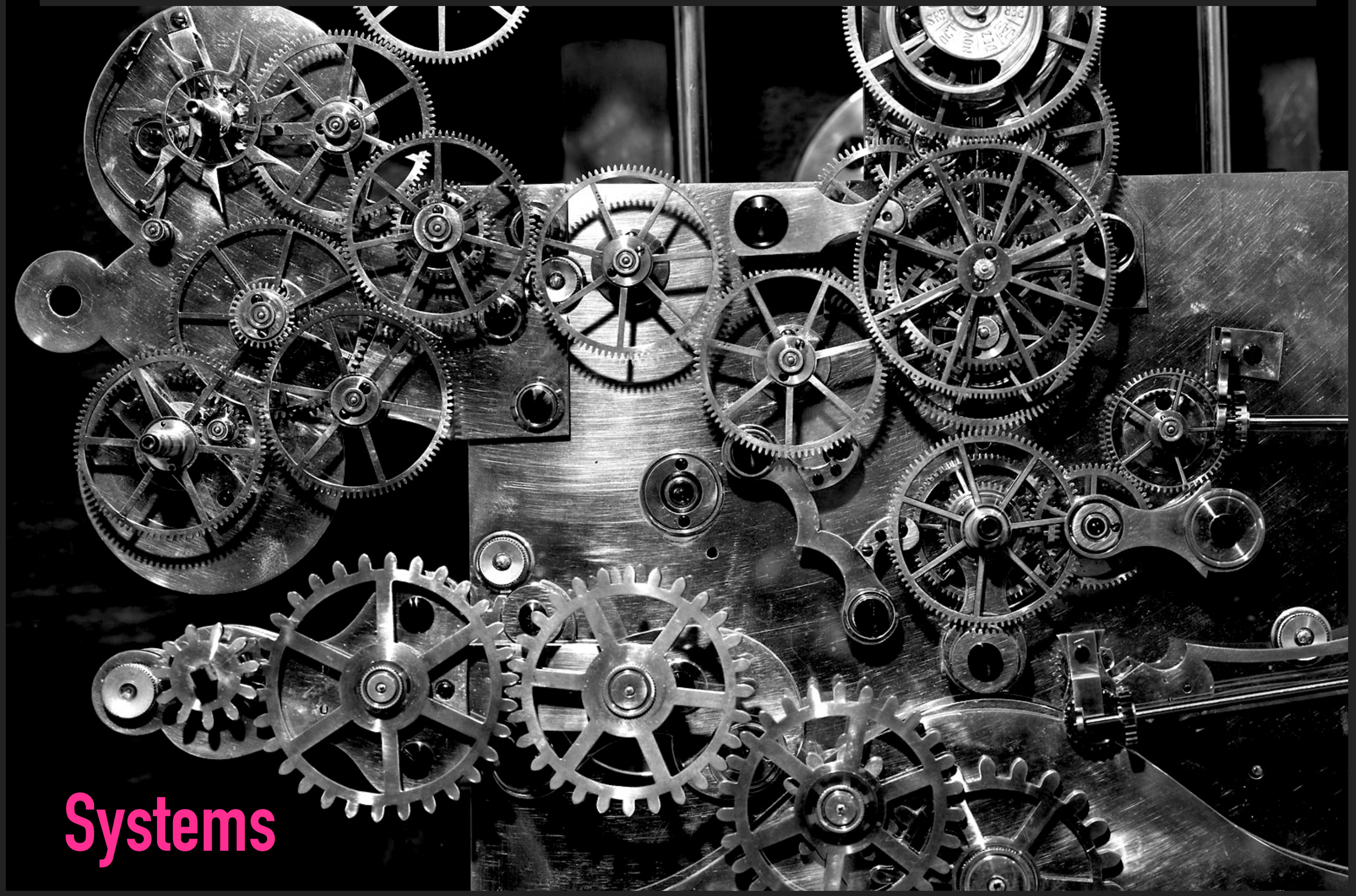
OCaml

Find a project

Reimplement something

Try something new

Join an Open Source project


Systems

Abstraction

This board belongs to Newcastle University PhD student Tom Fisher, who is doing research in homological algebra.
Thanks to Christian Perfect for the photo.
whatsonmyblackboard.wordpress.com

Enjoy!

Type-driven development

Functional Concurrency

Systems programming in ML

SIMON THOMPSON

LEARNING FUNCTIONAL PROGRAMMING