parser combinators

```
{:name "nate young"
 :from "revelytix"
 :date (sep 19 2011)}
```

why use parser combinators

how to use parser combinators

write your own

why use parser combinators

how to use parser combinators

write your own

for those still awake: category theory!

formal grammars have large upfront tax

LL, LR, LALR

formal grammars have large upfront tax

- LL, LR, LALR

project integration

- yacc, ANTLR
- parser generators aren't in-language

formal grammars have large upfront tax

- LL, LR, LALR

project integration

- yacc, ANTLR
- parser generators aren't in-language

the anti-DSL

- DSL == small, specific (new) language
- parsec == small part of (old) language

modular, TDD-friendly

modular, TDD-friendly

user-friendly default error messages

modular, TDD-friendly

user-friendly default error messages

closely resemble normal functions

natural mix of parsing and processing

modular, TDD-friendly

user-friendly default error messages

closely resemble normal functions

natural mix of parsing and processing

incredibly fine-grained

use of closures borders on pathological

Parsec

JParsec

JSParsec

C++

Parsnip

Parsec Erlang

Pysec

Ruby Parsec

C#

NParsec

PCL

Consumed

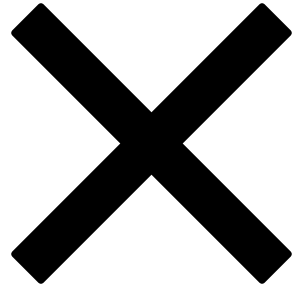
Empty

Ok

Error

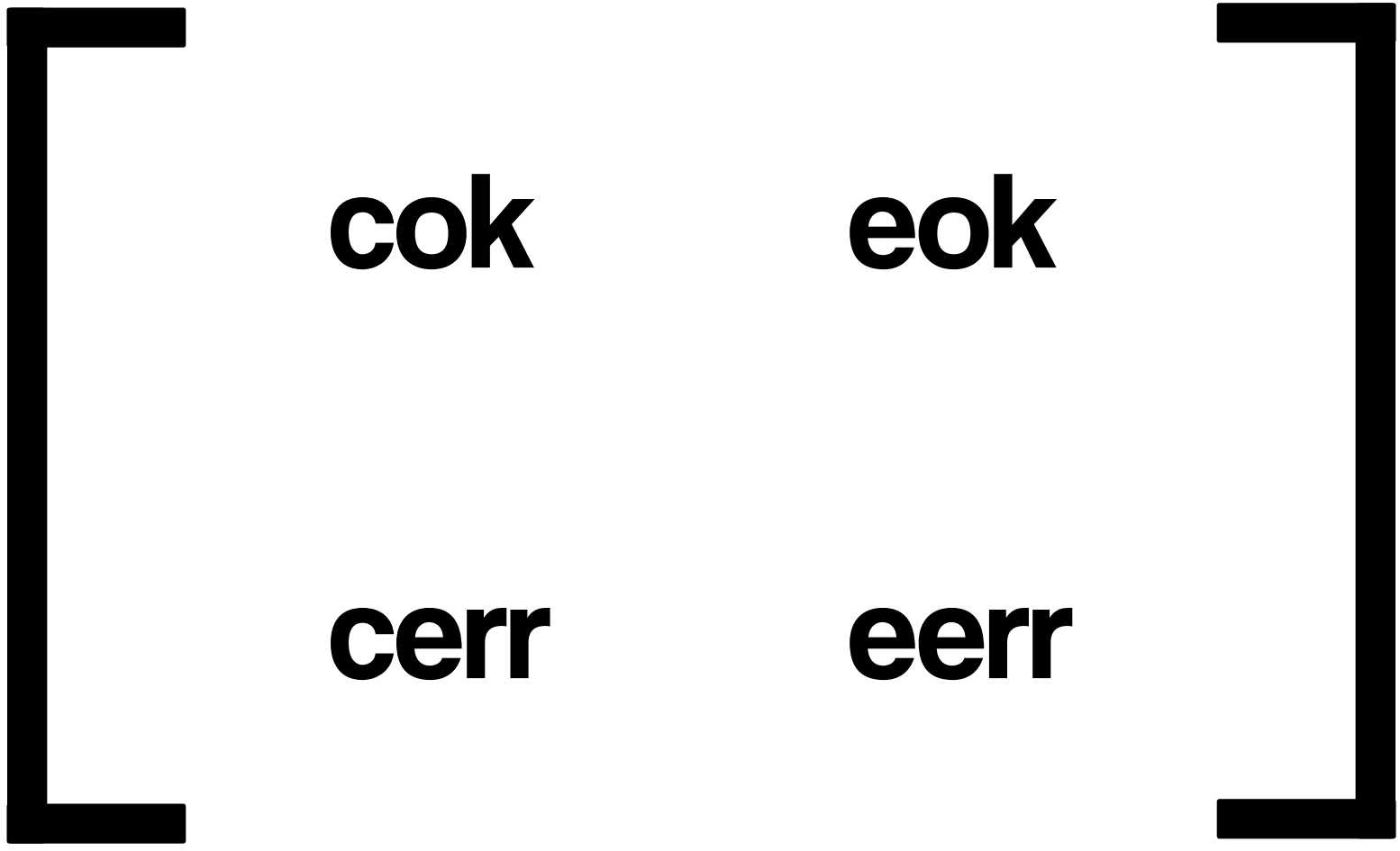
Consumed

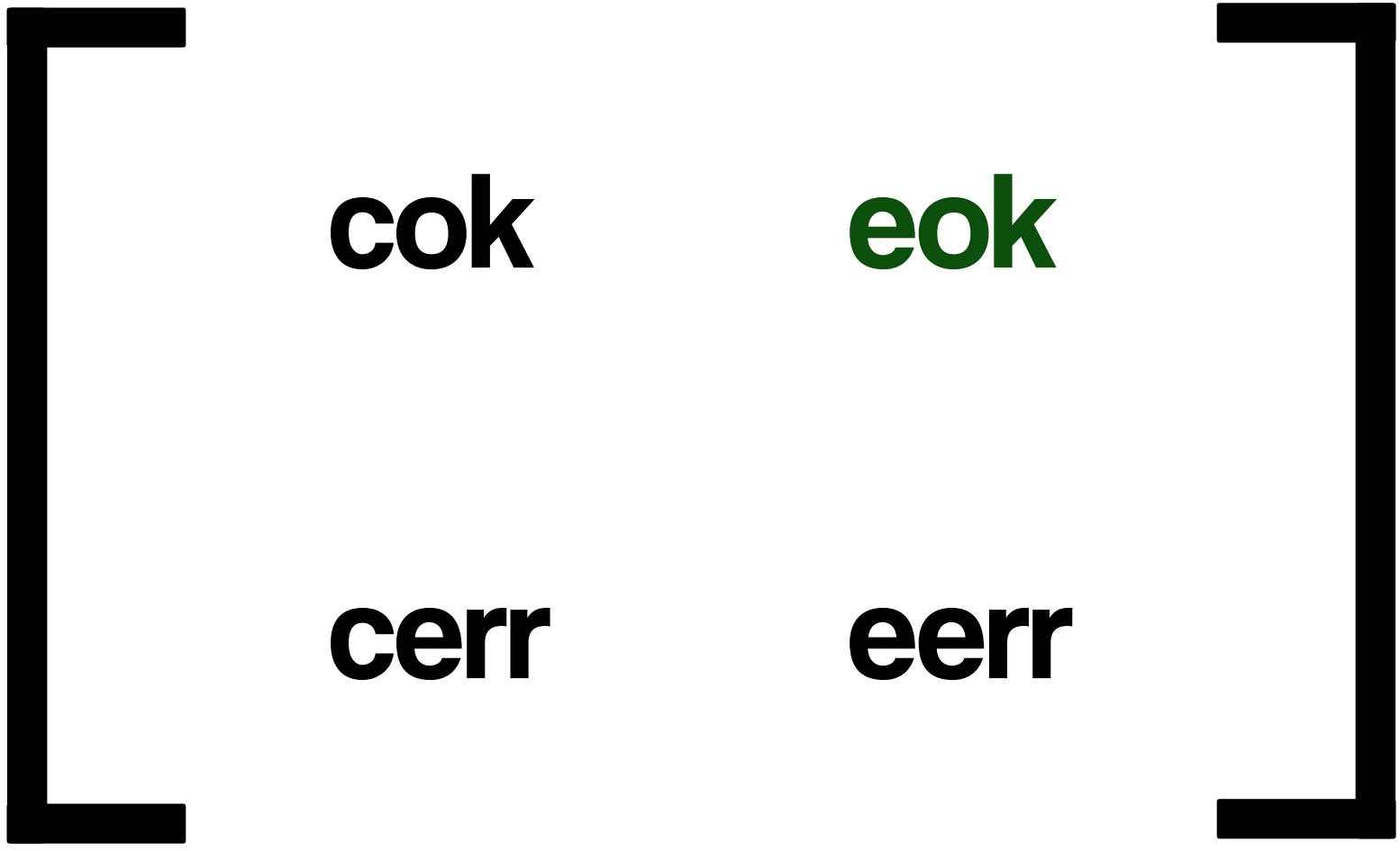
Ok



Empty

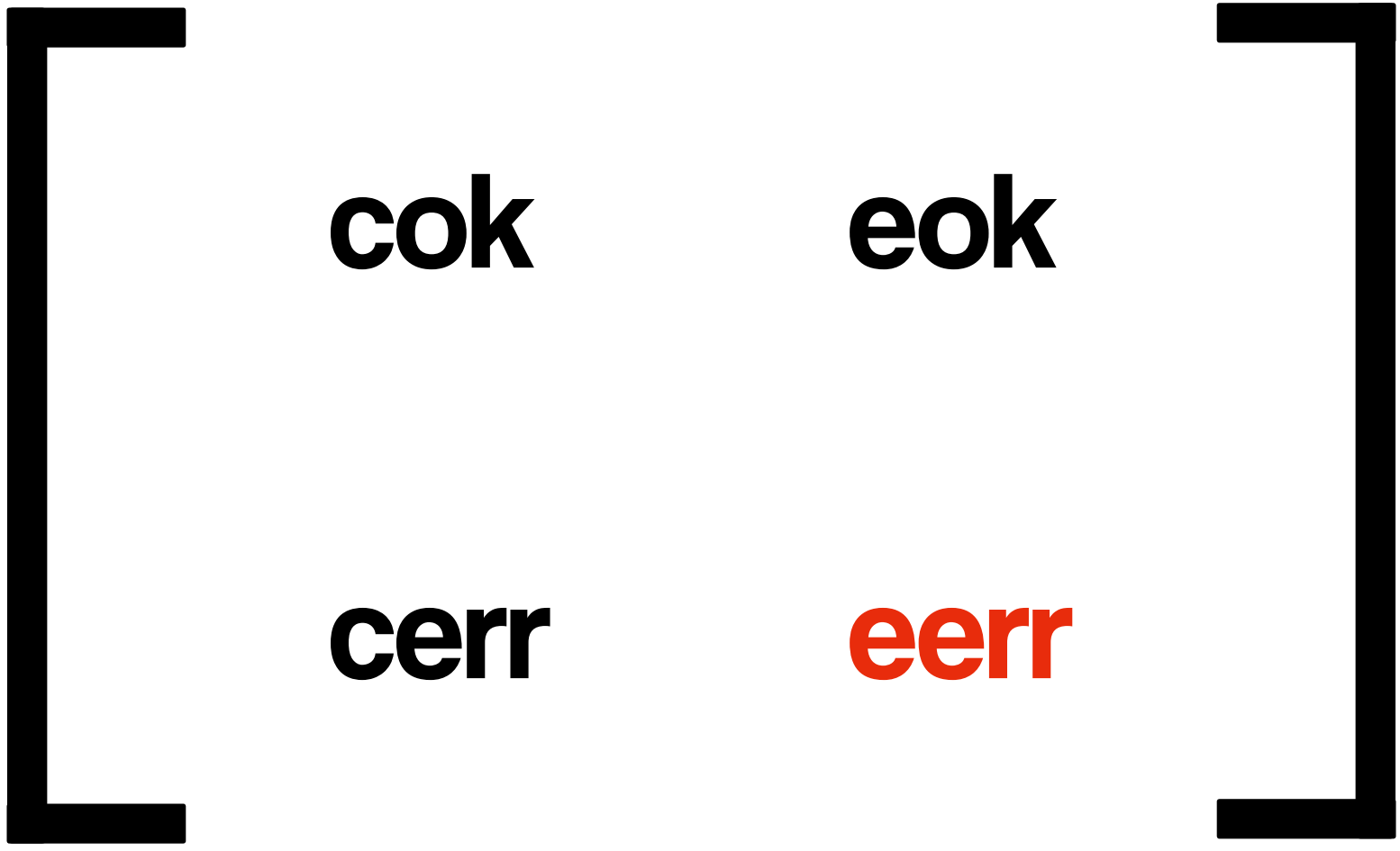
Error


cok
eok
cerr
eerr


cok
eok
cerr
eerr

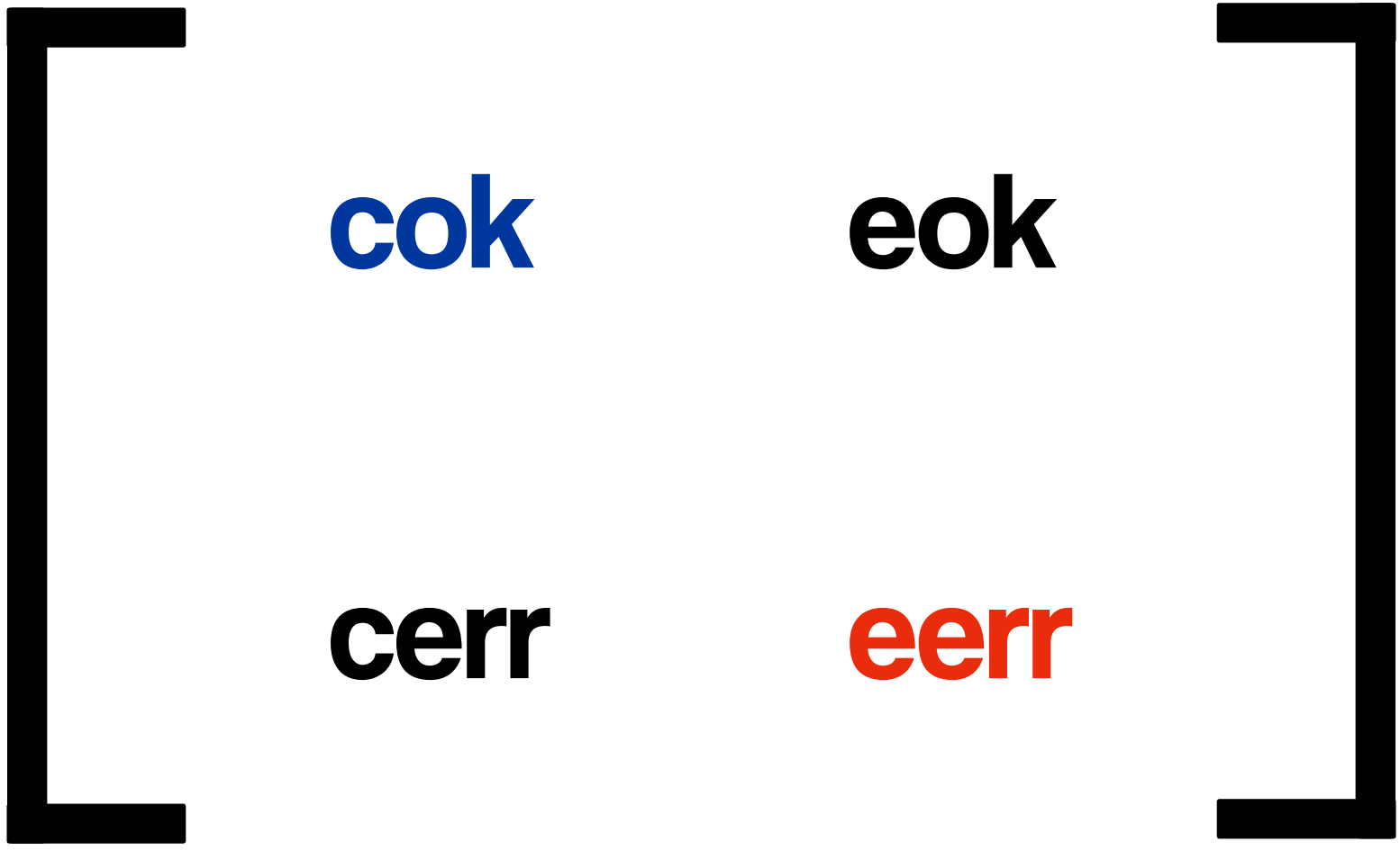
```
(defn always [x]
  (fn [state cok cerr eok eerr]
    (eok x state)))
```

```clojure
(defn always [x]
  (fn [state cok cerr eok eerr]
    (eok x state)))
```


cok
eok
cerr
eerr

```
(defn never []
  (fn [state cok cerr eok eerr]
    (eerr (UnknownError. (:pos state)))))
```

```
(defn never []
  (fn [state cok cerr eok eerr]
    (eerr (UnknownError. (:pos state)))))
```


cok
eok
cerr
eerr

```
(defn token []
  (fn [{:keys (input pos) :as state} cok cerr eok eerr]
    (if-let [tok (first input)]
      (cok tok (InputState. (rest input) (inc pos)))
      (eerr (UnexpectedError. "End of input" pos)))))
```

```clojure
(defn token []
  (fn [{:keys (input pos) :as state} cok cerr eok eerr]
    (if-let [tok (first input)]
      (cok tok (InputState. (rest input) (inc pos)))
      (eerr (UnexpectedError. "End of input" pos)))))
```

$$p \gg q$$

what should p >> q return?

what should p >> q return?

what if q depends on the value of p?

$$p \gg q$$

p

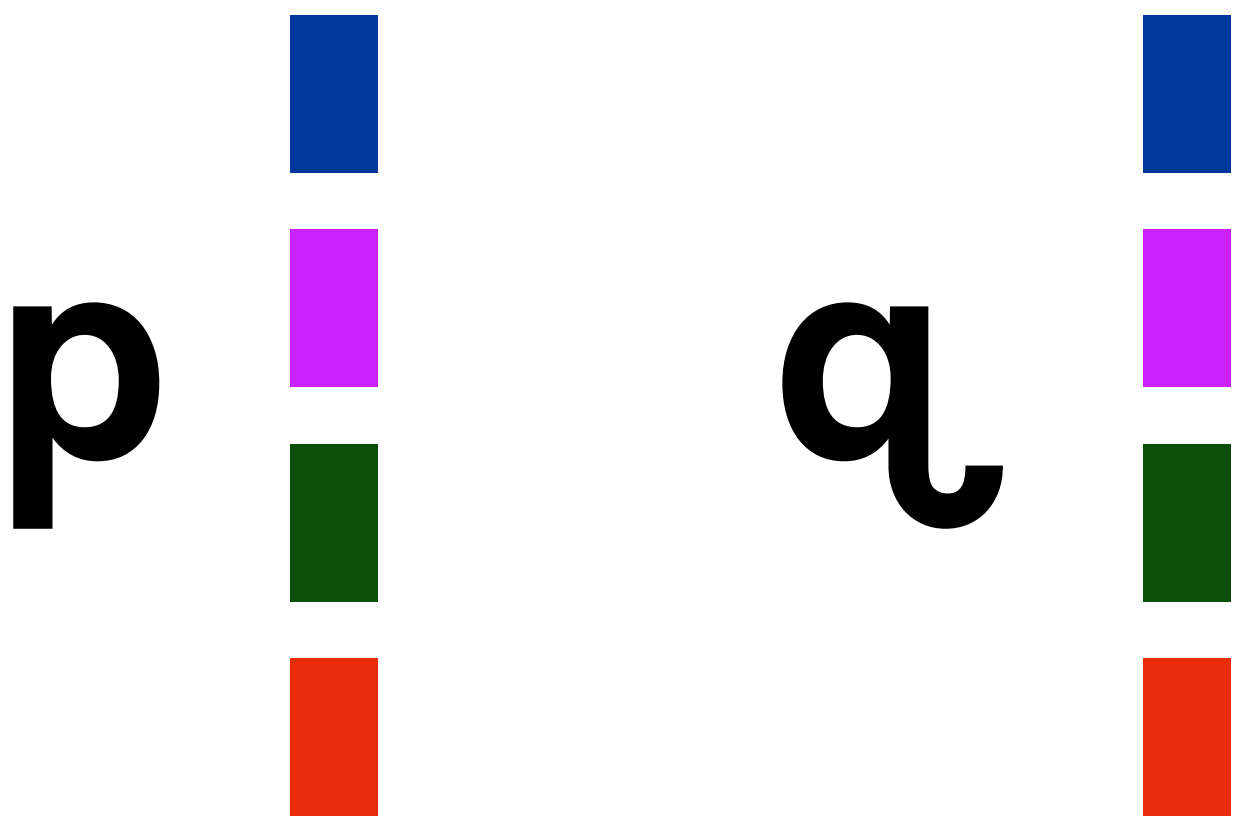
q

cok

cerr

eok

eerr

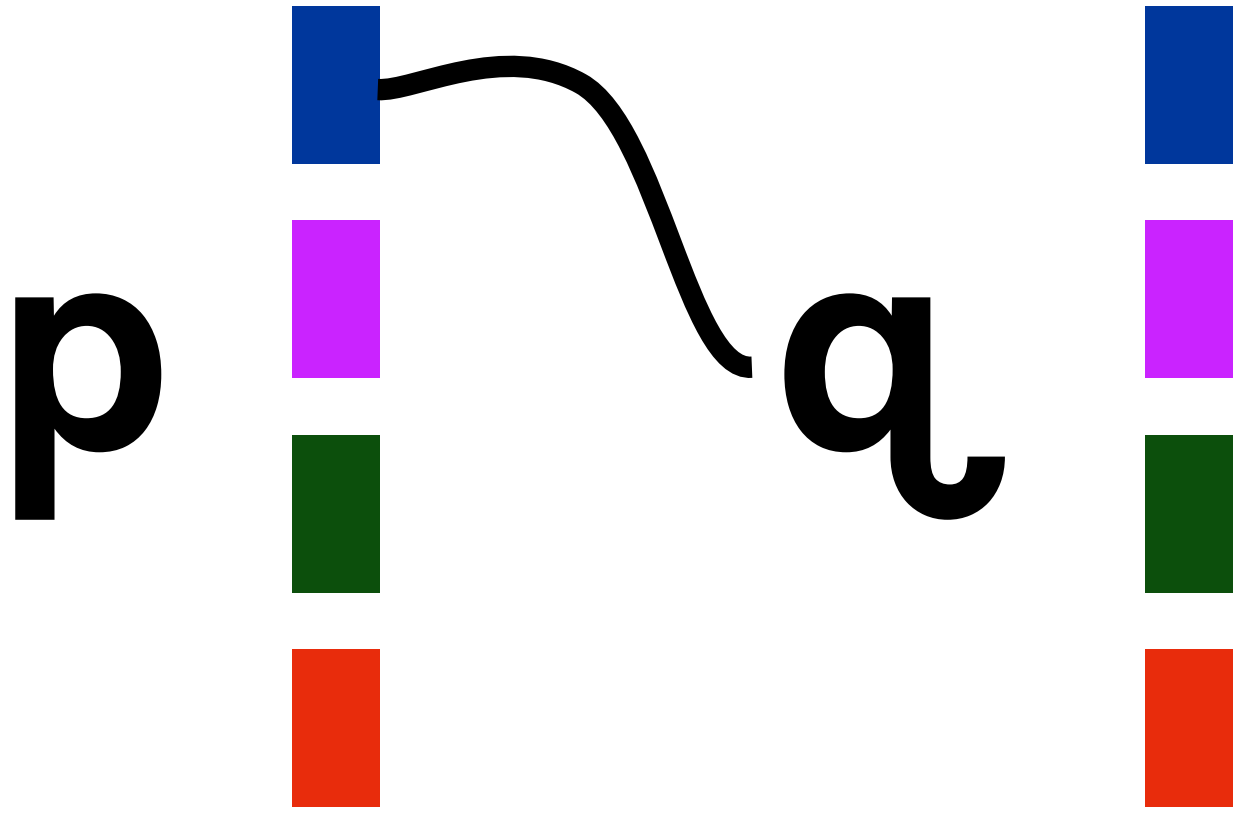


cok

cerr

eok

eerr

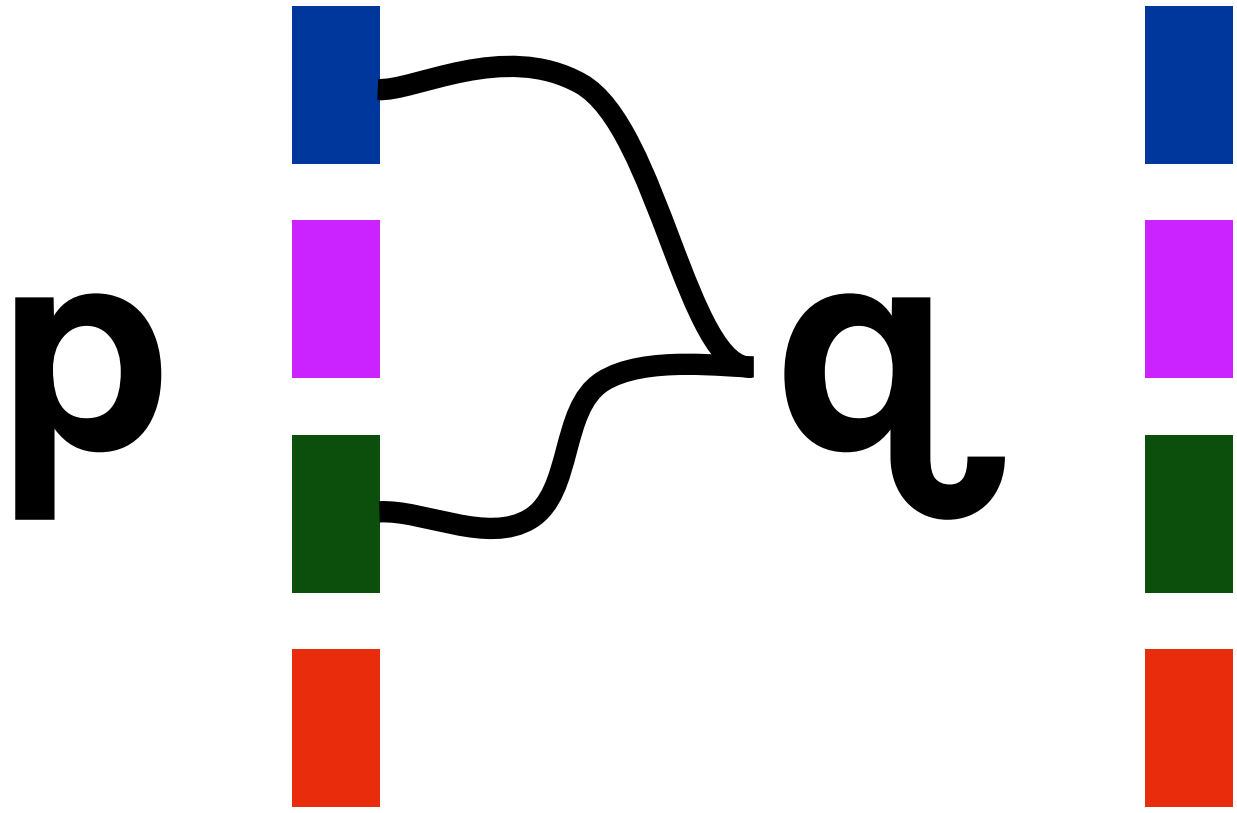


cok

cerr

eok

eerr

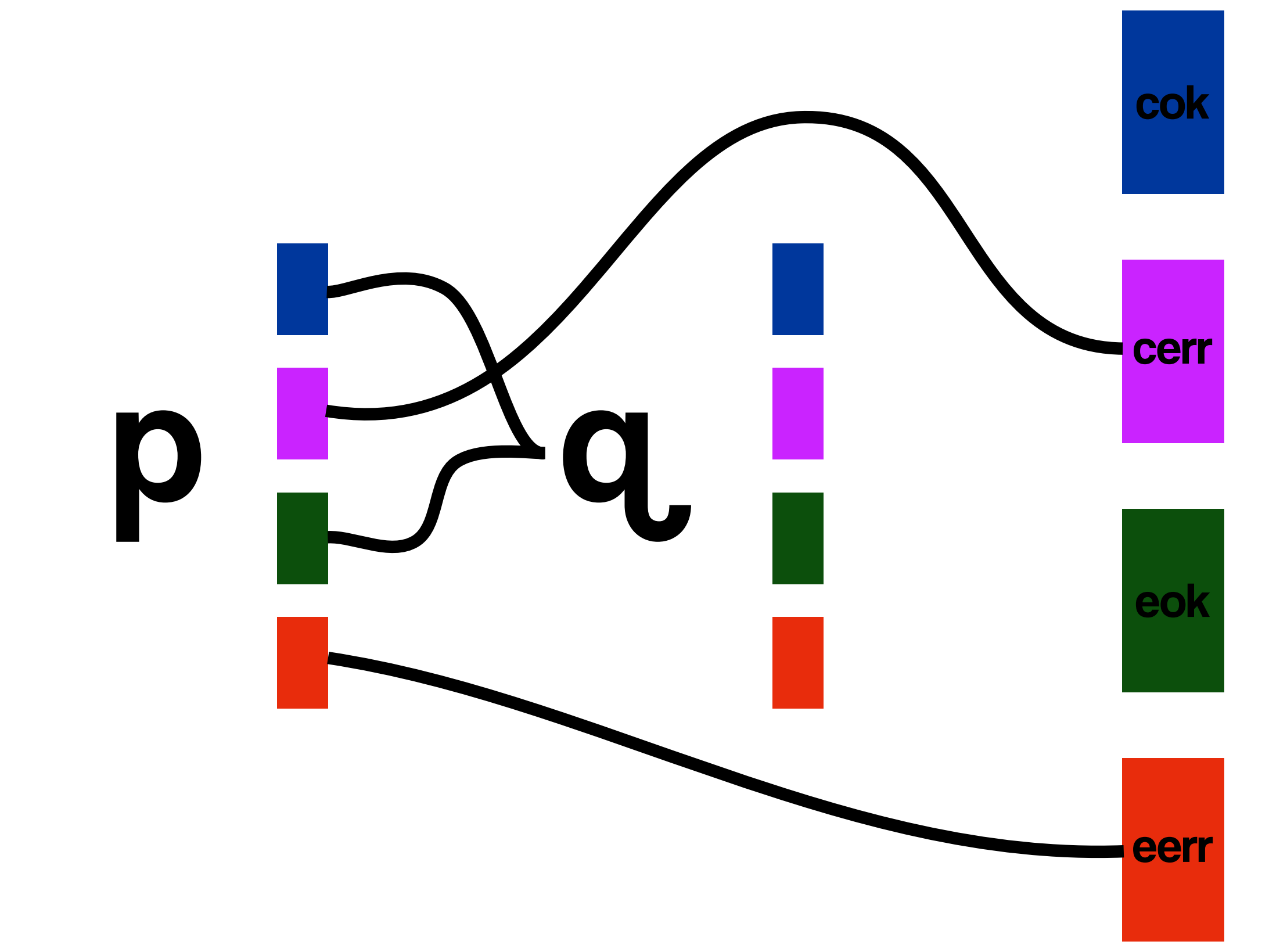


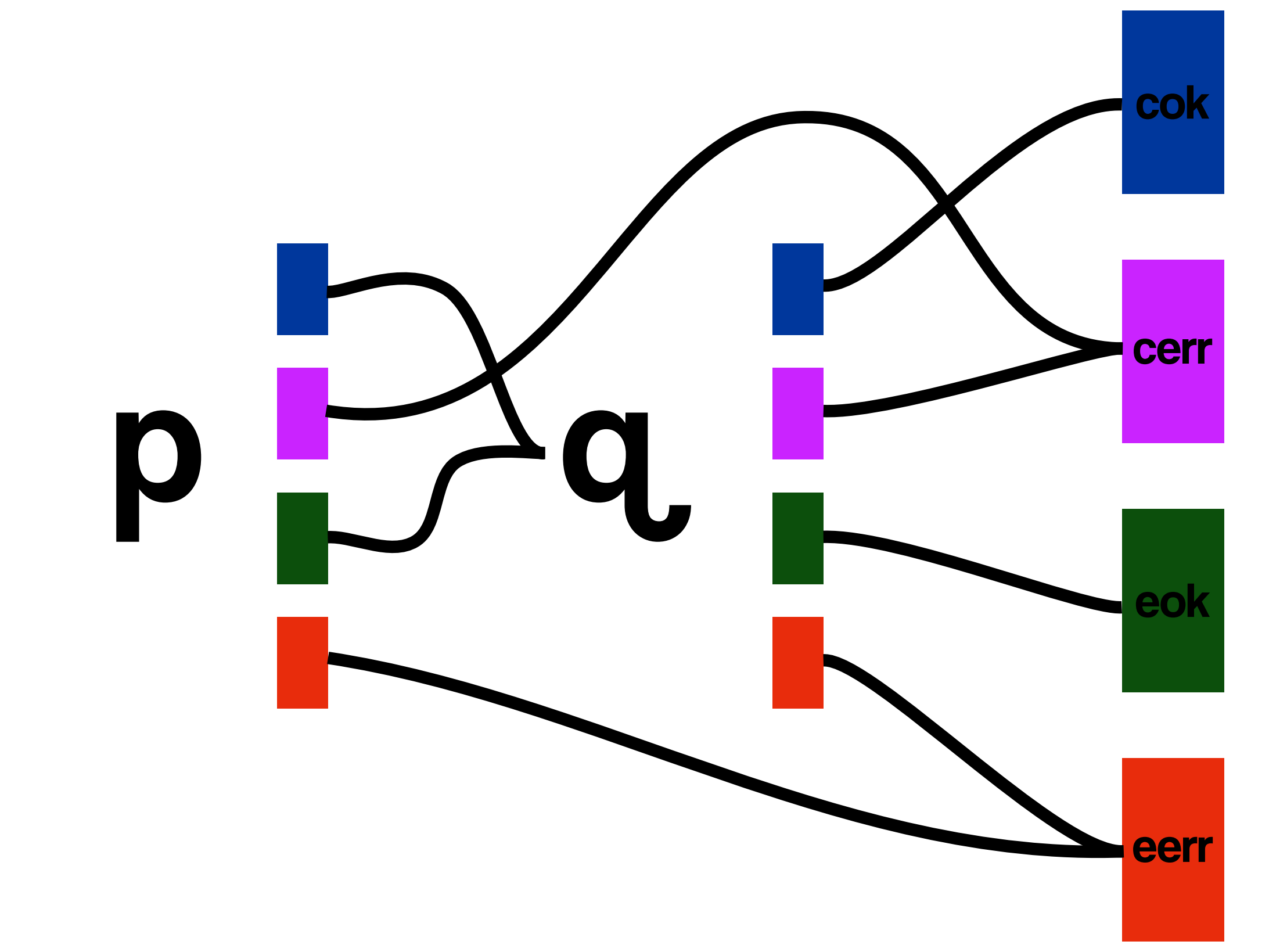
cok

cerr

eok

eerr


p
q
cok
cerr
eok
eerr


p
q
cok
cerr
eok
eerr

```
(defn next [p q]
  (fn [state cok cerr eok eerr]
    (letfn [(pcok [item state]
              (q state cok cerr cok cerr))
            (peok [item state]
              (q state cok cerr eok eerr))]
      (p state pcok cerr peok eerr))))
```

```
(defn next [p q]
  (fn [state cok cerr eok eerr]
    (letfn [(pcok [item state]
              (q state cok cerr cok cerr))
            (peok [item state]
              (q state cok cerr eok eerr))]
      (p state pcok cerr peok eerr))))
```

```
(defn bind [p f]
  (fn [state cok cerr eok eerr]
    (letfn [(pcok [item state]
                  (let [q (f item)]
                    (q state cok cerr cok cerr)))
            (peok [item state]
                  (let [q (f item)]
                    (q state cok cerr eok eerr)))]
      (p state pcok cerr peok eerr))))
```

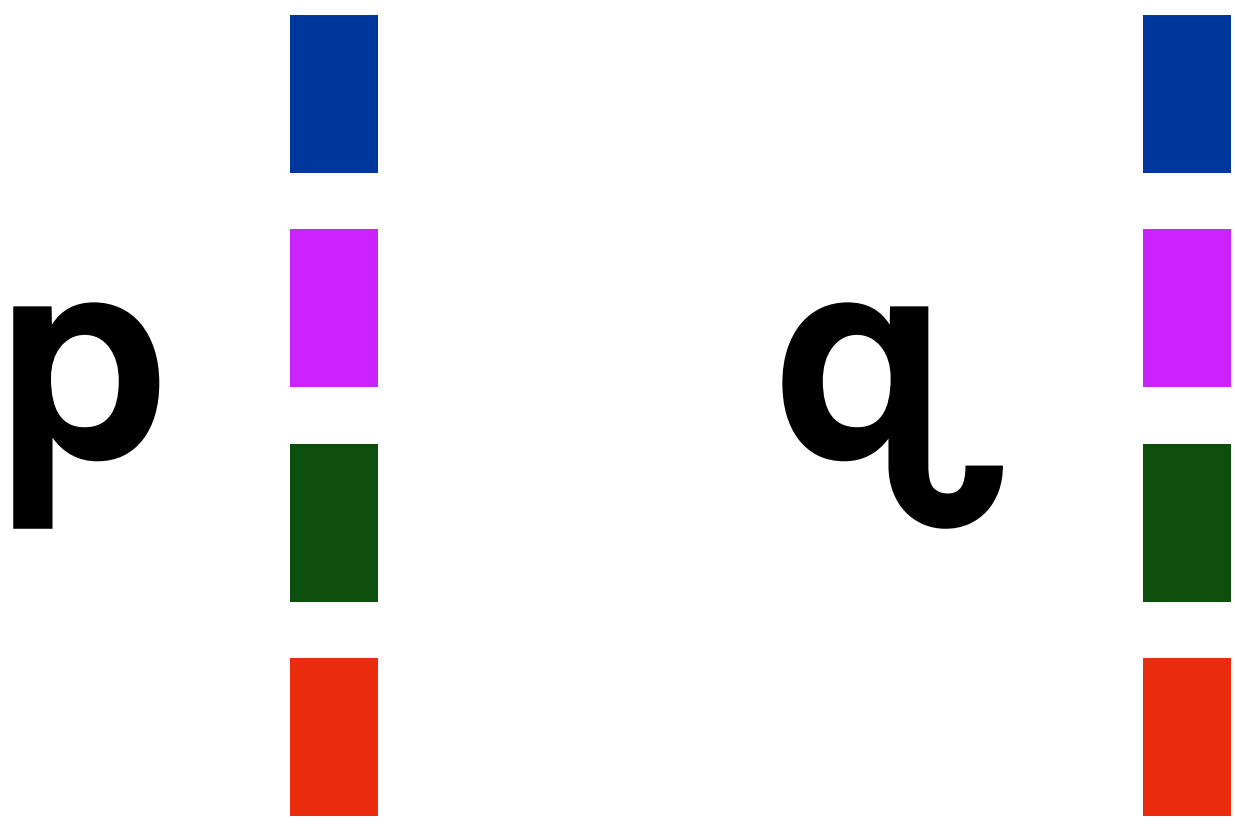
$$p \ <|> q$$

what should p <|> q return?

what should p <|> q return?

error message if both fail?

$$p \ <|> q$$

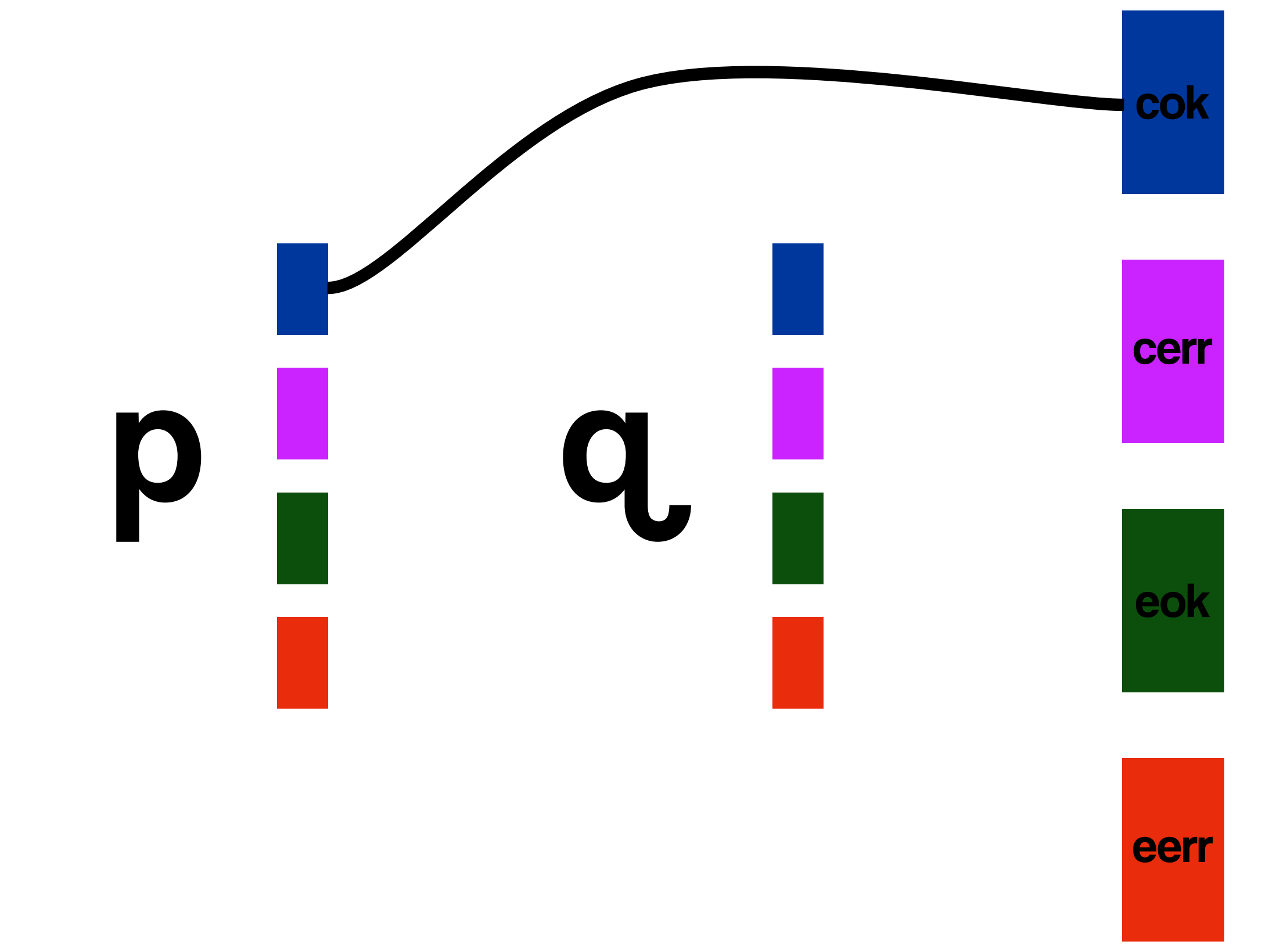


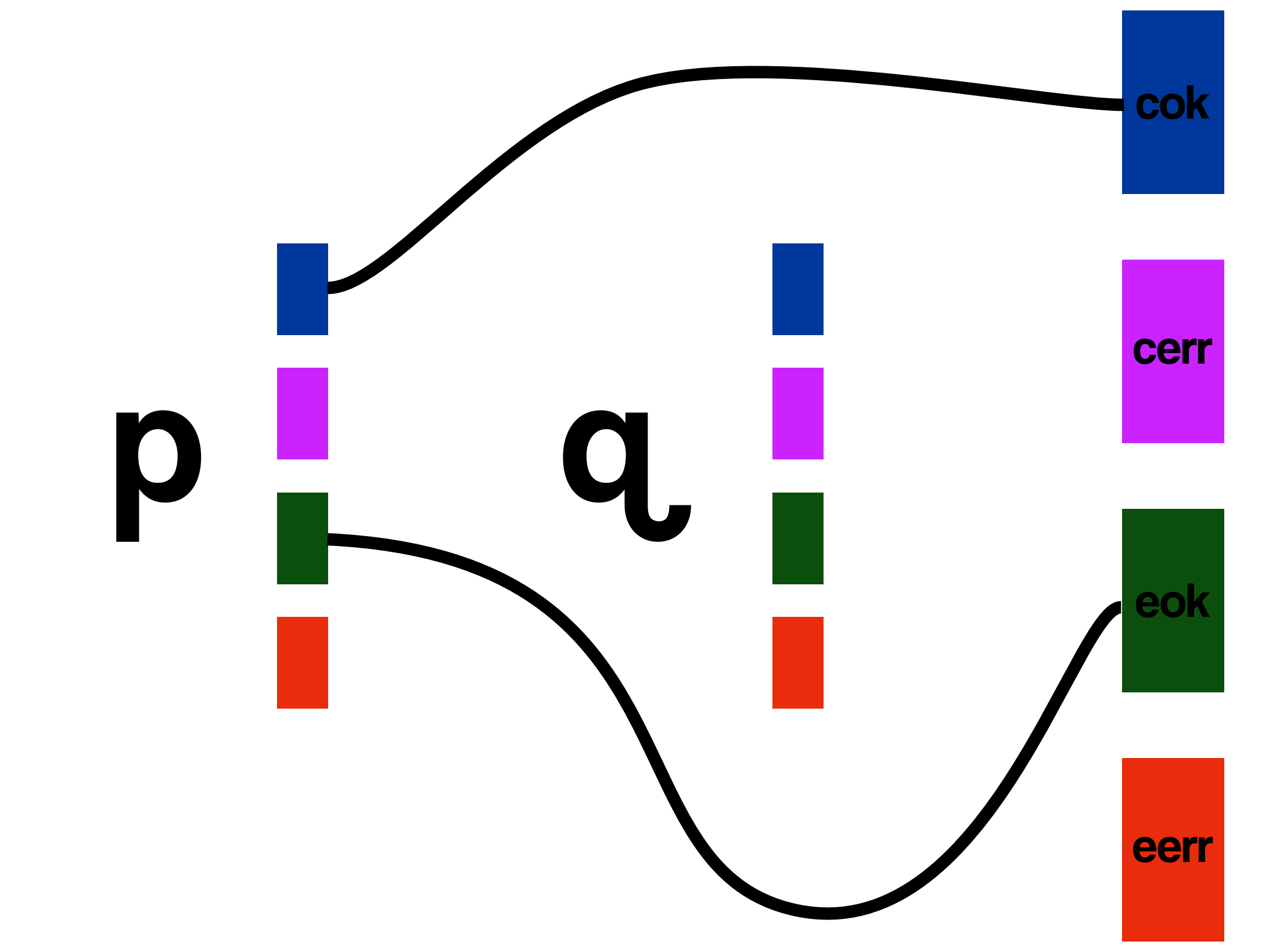
cok

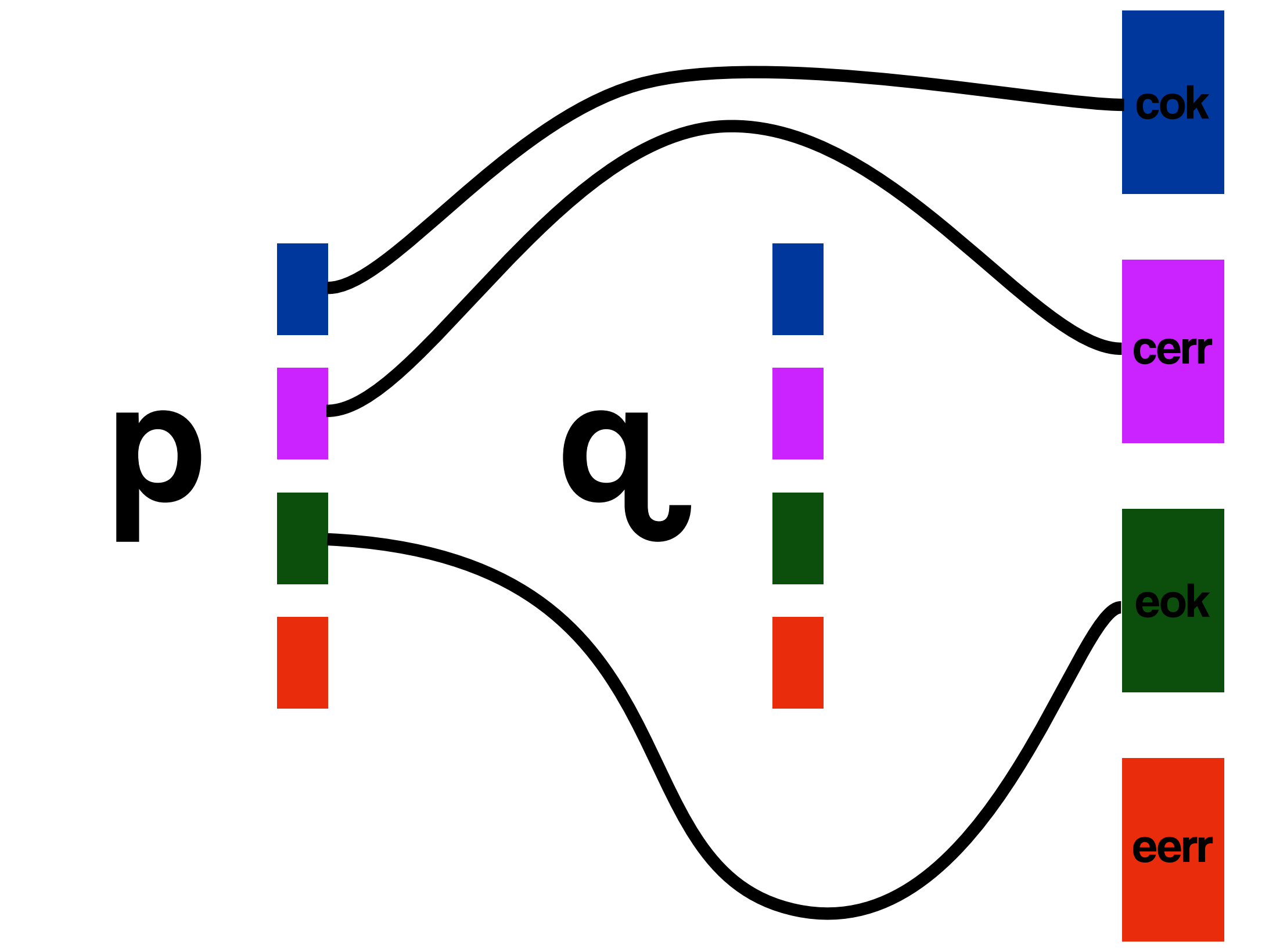
cerr

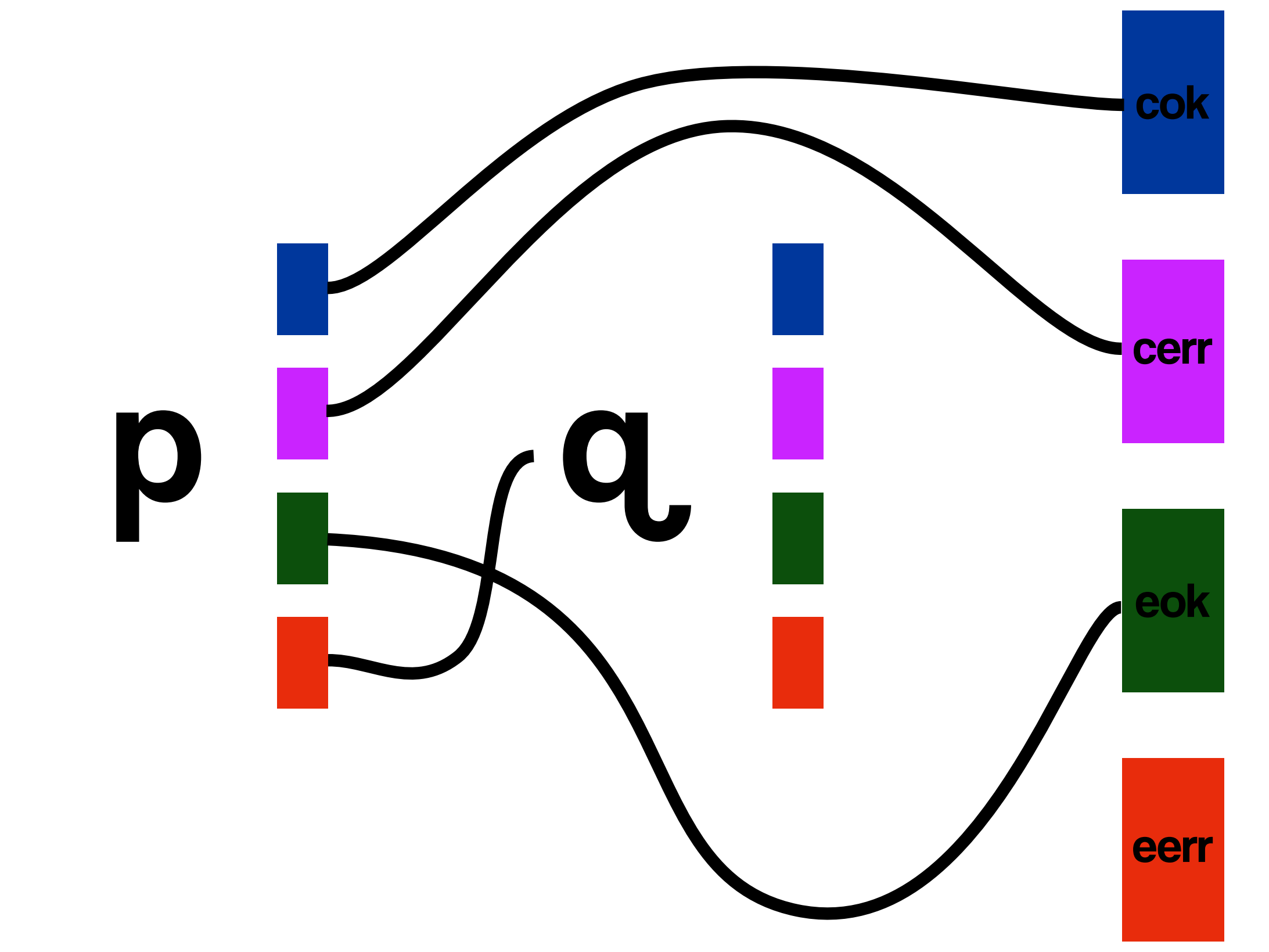
eok

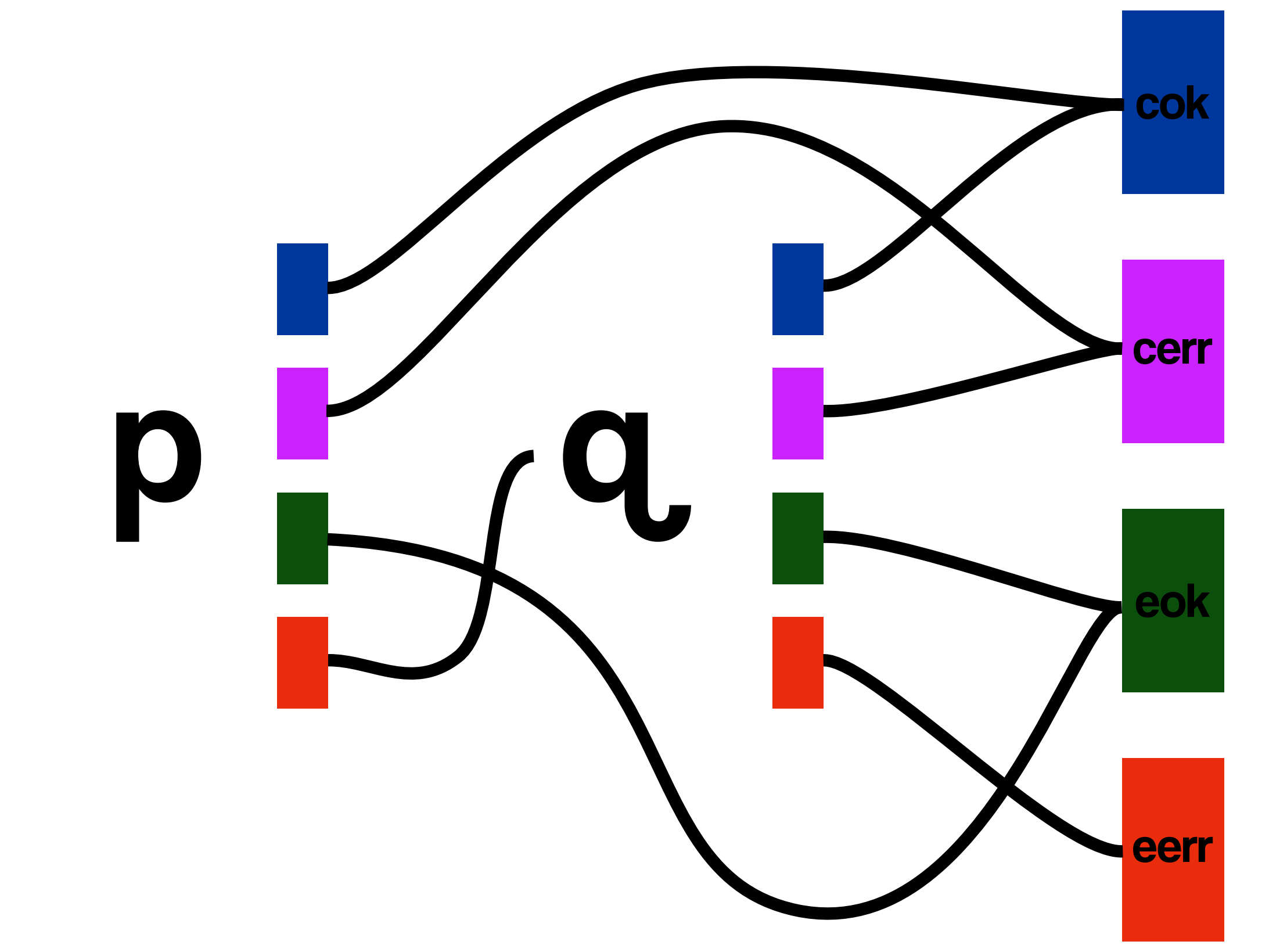
eerr


cok
cerr
eok
eerr
p
q


cok
cerr
p
q
eok
eerr


p
q
cok
cerr
eok
eerr

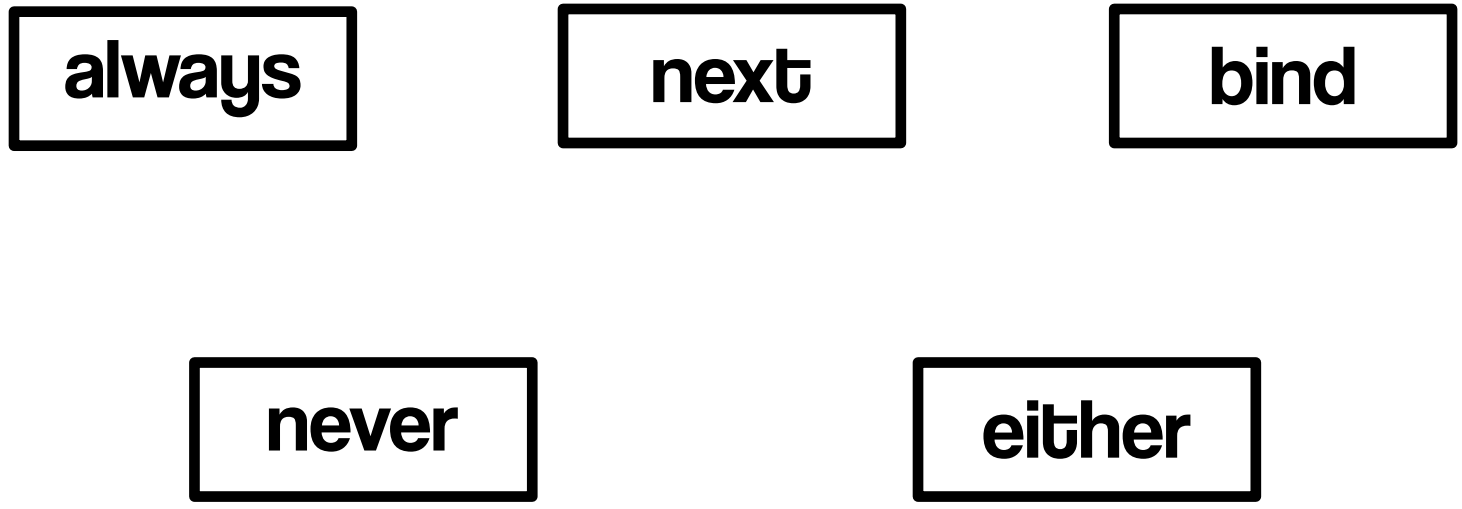

p
q
cok
cerr
eok
eerr

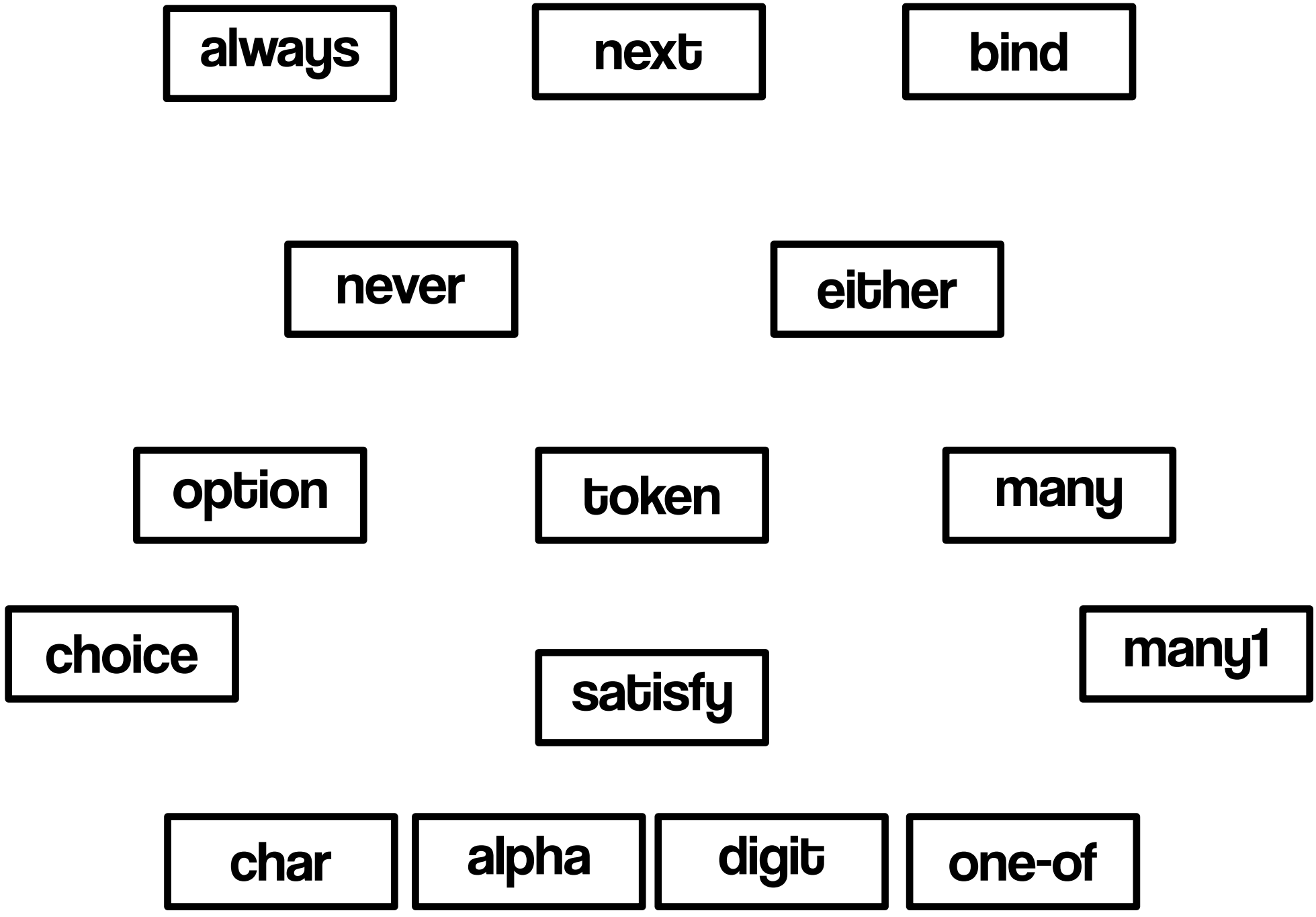

p
q
cok
cerr
eok
eerr

```
(defn either []
  (fn [state cok cerr eok eerr]
    (letfn [(peerr [from-p]
              (letfn [(qeerr [from-q]
                        (eerr (merge from-p
                                     from-q)))]
                (q state cok cerr eok qeerr)))]
      (p state cok cerr eok peerr))))
```

```
(defn either []
  (fn [state cok cerr eok eerr]
    (letfn [(peerr [from-p]
              (letfn [(qeerr [from-q]
                        (eerr (merge from-p
                                     from-q)))]
                (q state cok cerr eok qeerr)))]
      (p state cok cerr eok peerr))))
```


always
next
bind


always
next
bind
never
either


always
next
bind
never
either
option
token
many
choice
satisfy
many1
char
alpha
digit
one-of

hofs (um, hops?)

hofs (um, hops?)

many

hofs (um, hops?)

many *

hofs (um, hops?)

many *

option

hofs (um, hops?)

many	*
option	|

hofs (um, hops?)

many	*
option	|
many1	

hofs (um, hops?)

many	*
option	|
many1	+

hofs (um, hops?)

many	*
option	|
many1	+
try	

hofs (um, hops?)

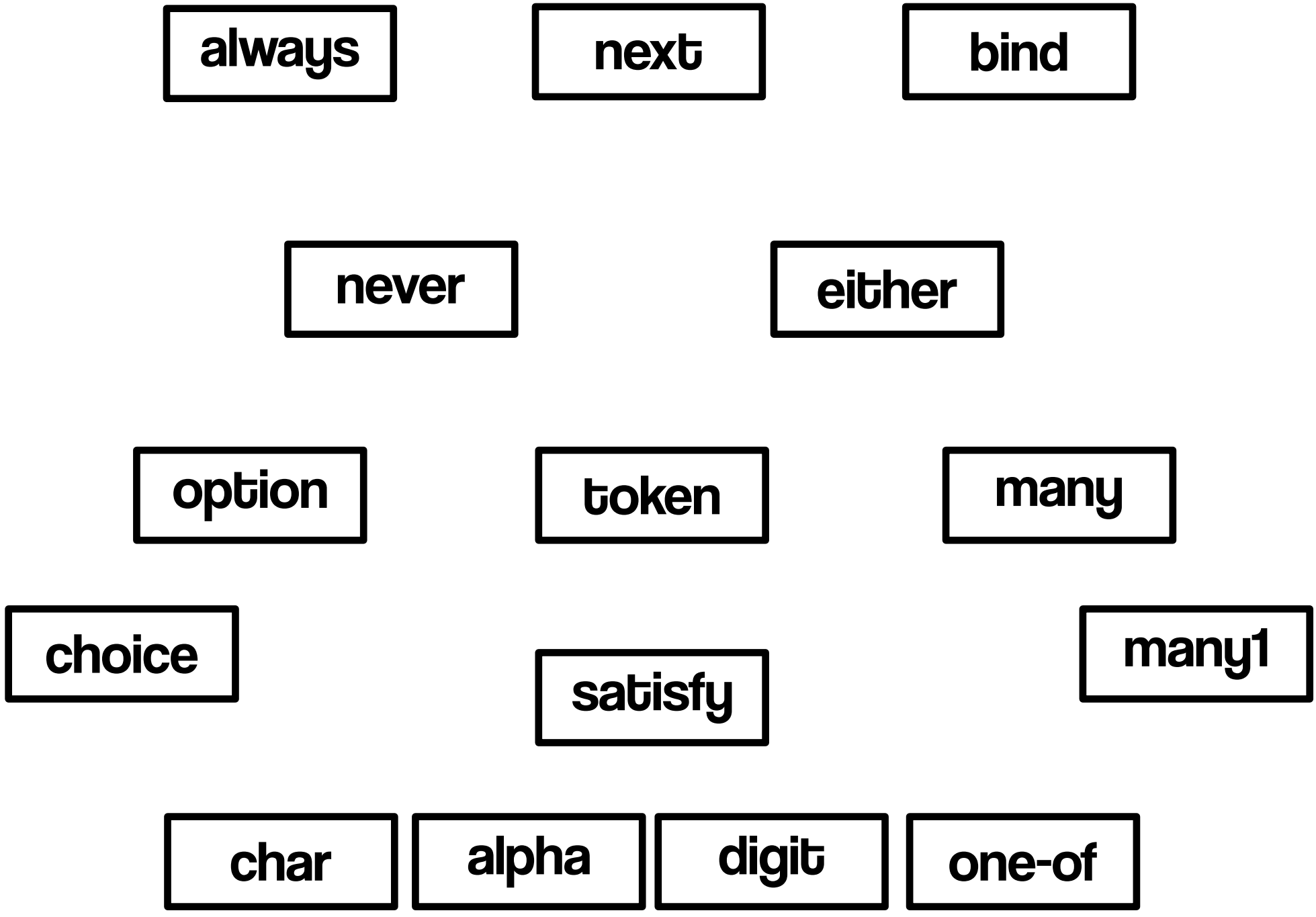
many	*
option	|
many1	+
try	fail without consuming

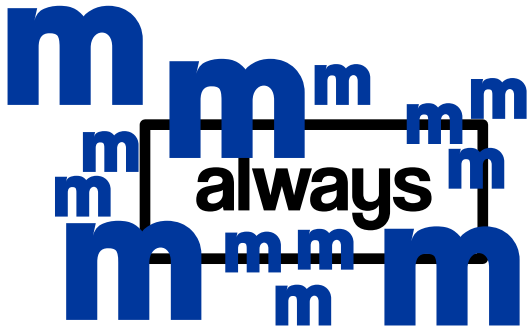
hofs (um, hops?)

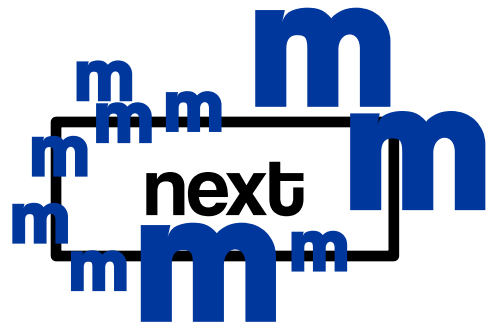
many	*
option	|
many1	+
try	fail without consuming
lookahead	

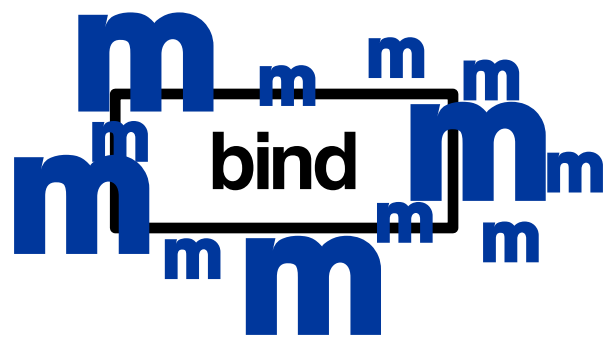
hofs (um, hops?)

many	*
option	|
many1	+
try	fail without consuming
lookahead	succeed without consuming


always
next
bind
never
either
option
token
many
choice
satisfy
many1
char
alpha
digit
one-of







never

either

option

token

many

choice

satisfy

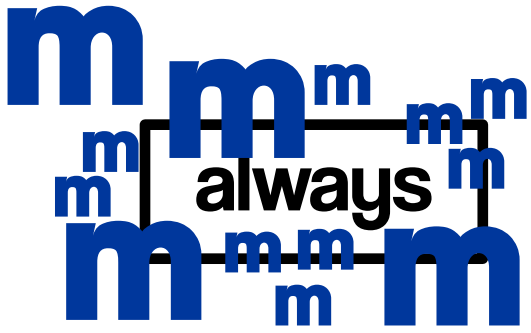
many1

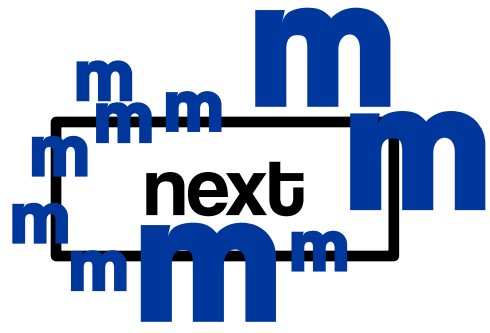
char

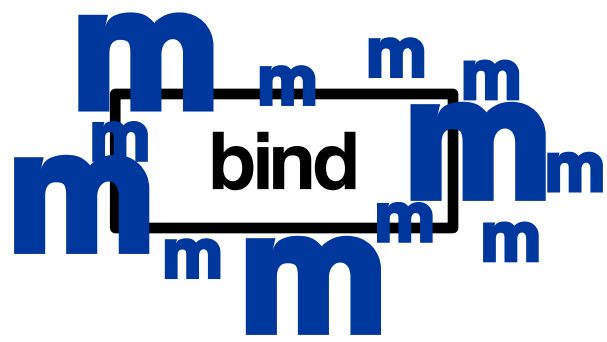
alpha

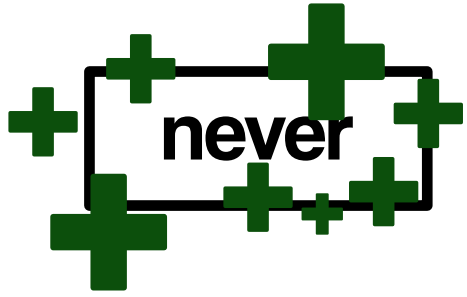
digit

one-of









option

token

many

choice

satisfy

many1

char

alpha

digit

one-of