# Polarized Data Parallel Data Flow

Ben Lippmeier, Fil Mackay, Amos Robinson

FHPC 2016/09/17

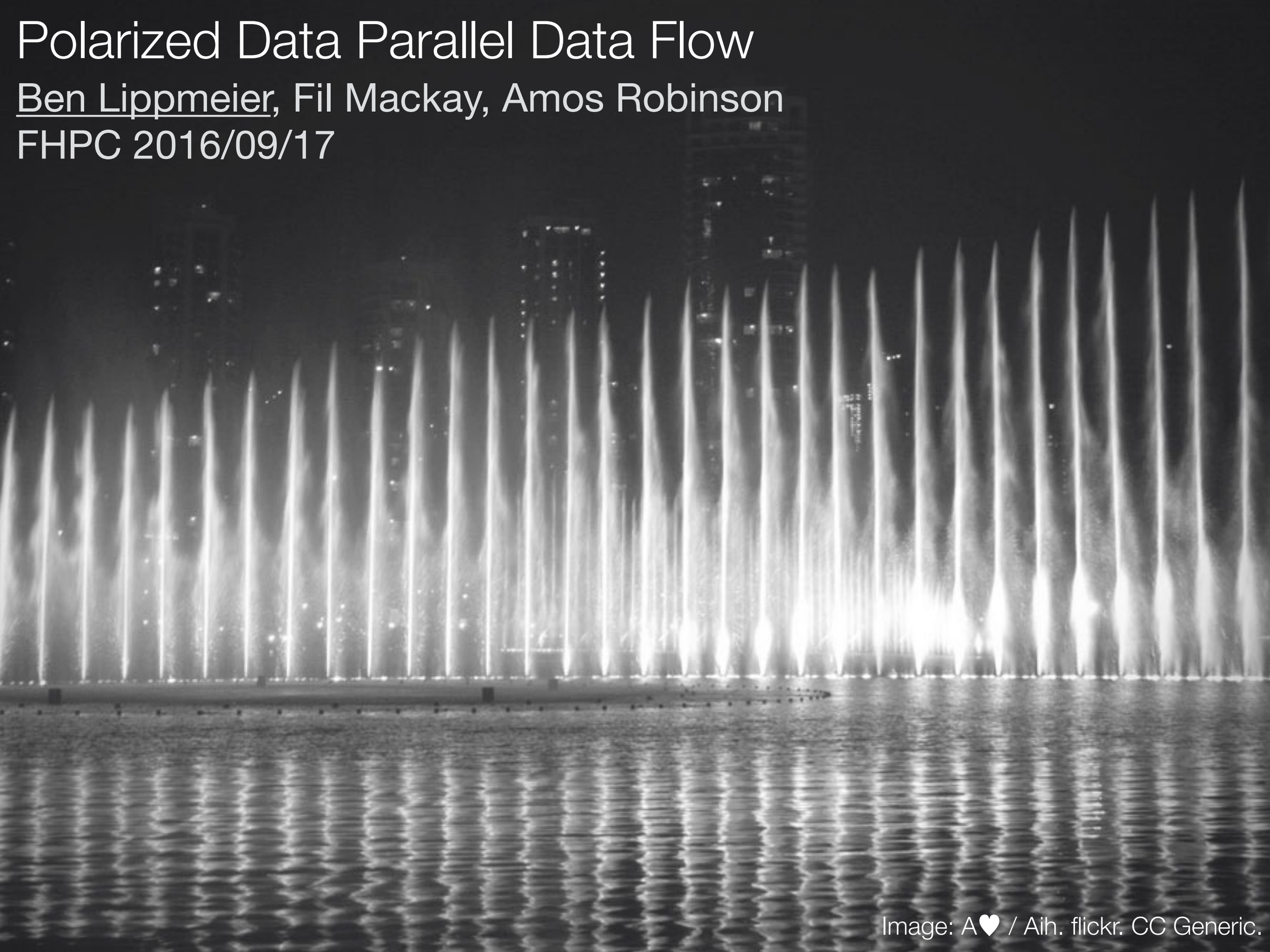
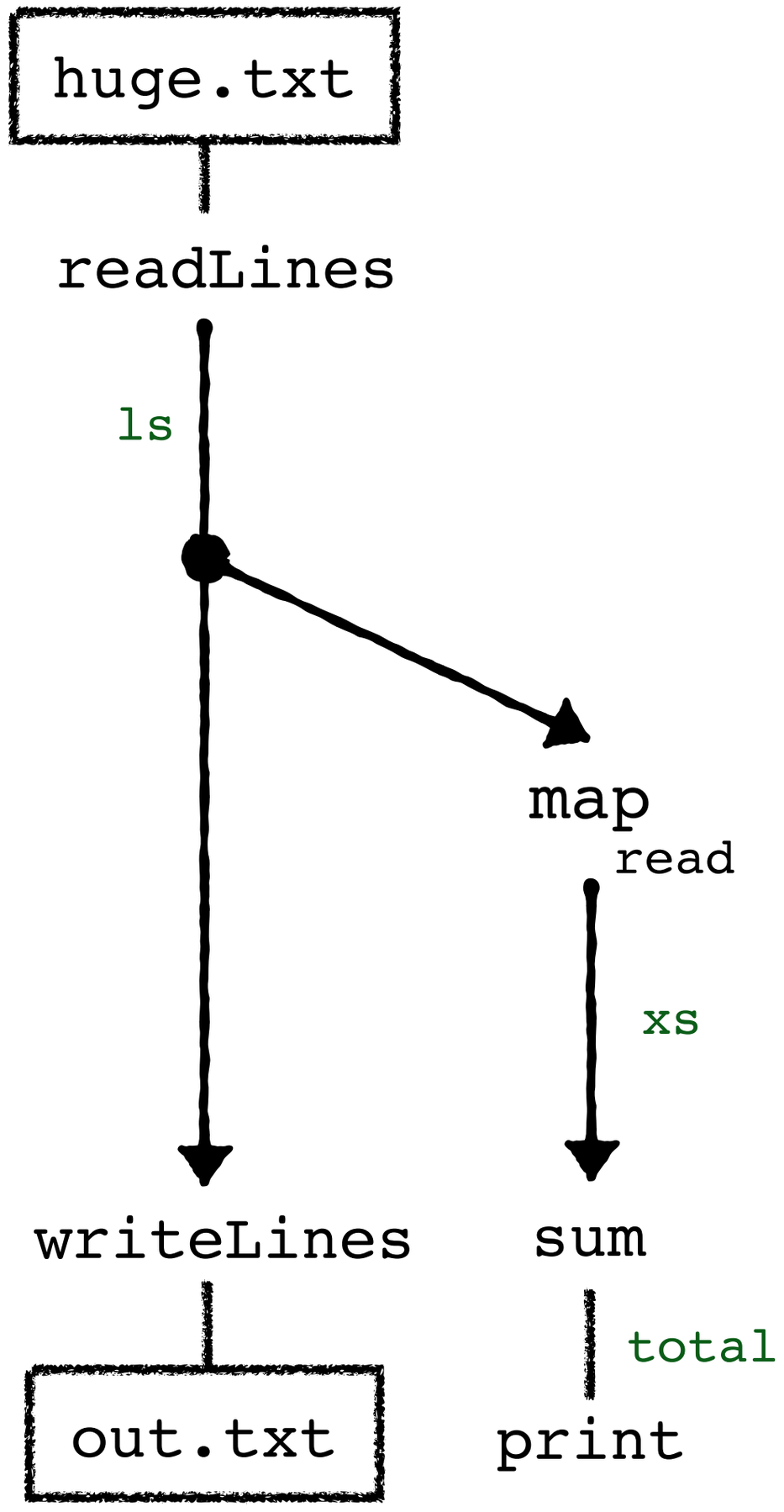
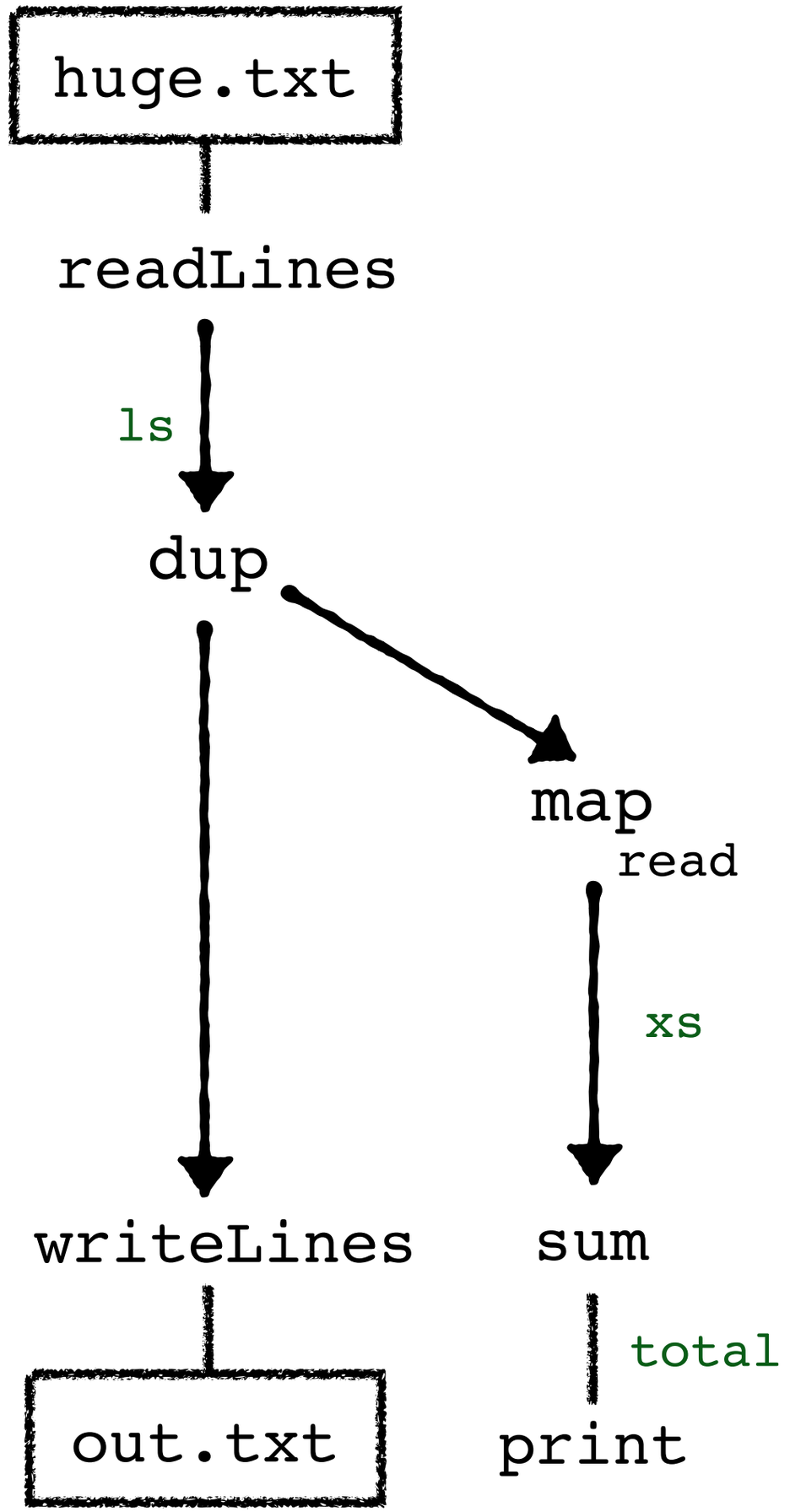


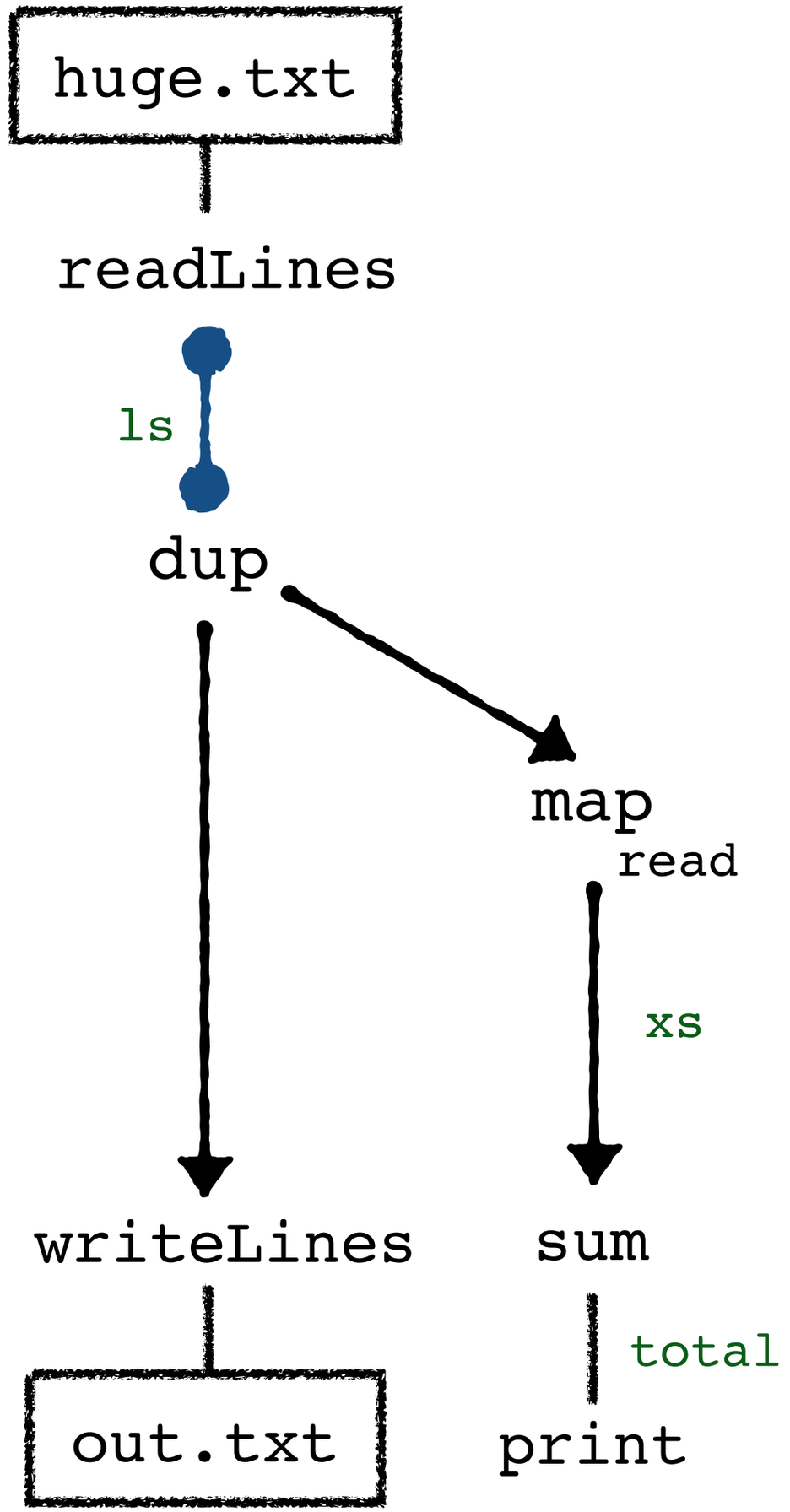
Image: A♥ / Aih. flickr. CC Generic.

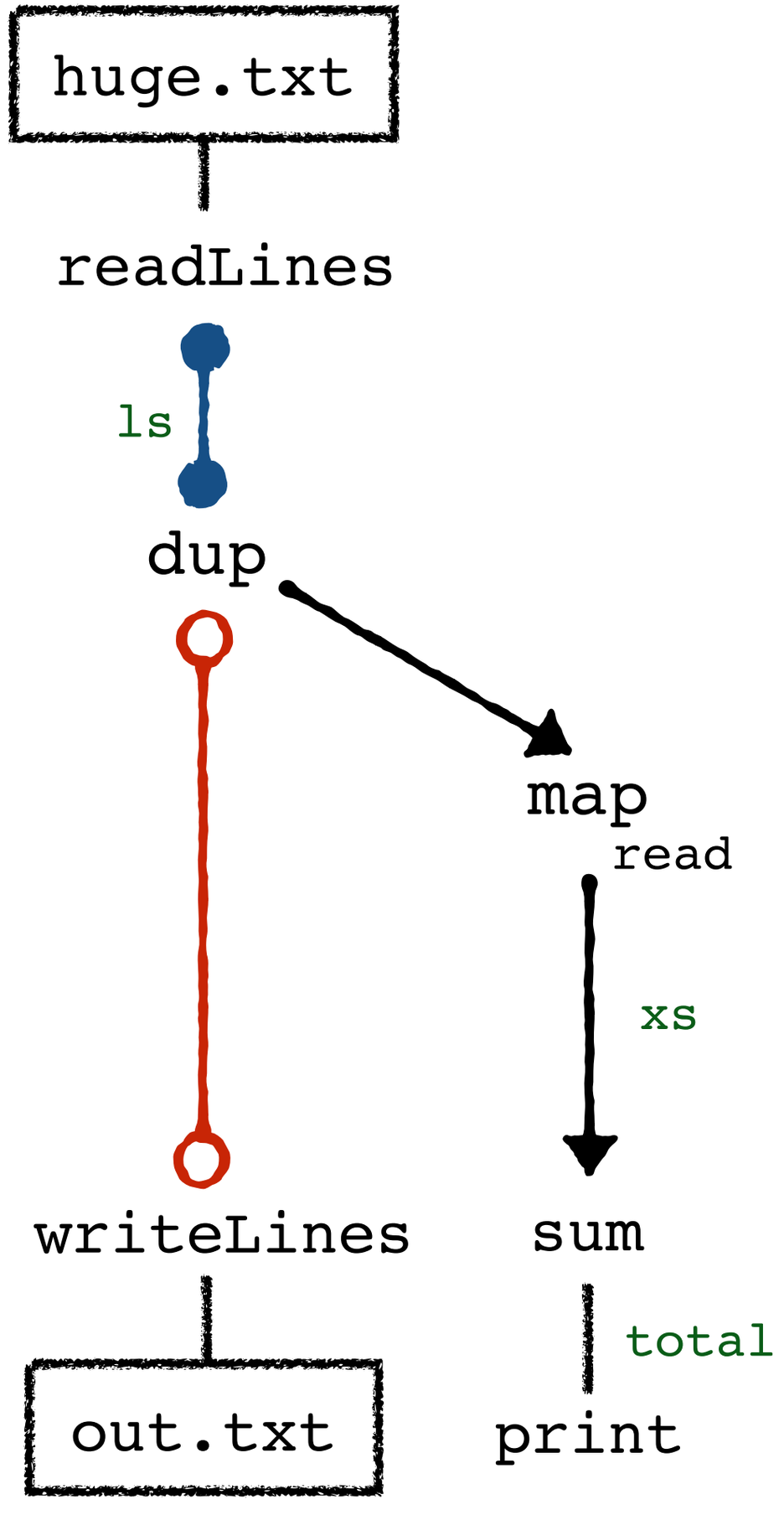
```
do ls <- readLines “huge.txt”
   writeLines “out.txt” ls
   let xs    = map read ls
   let total = sum xs
   print total
```

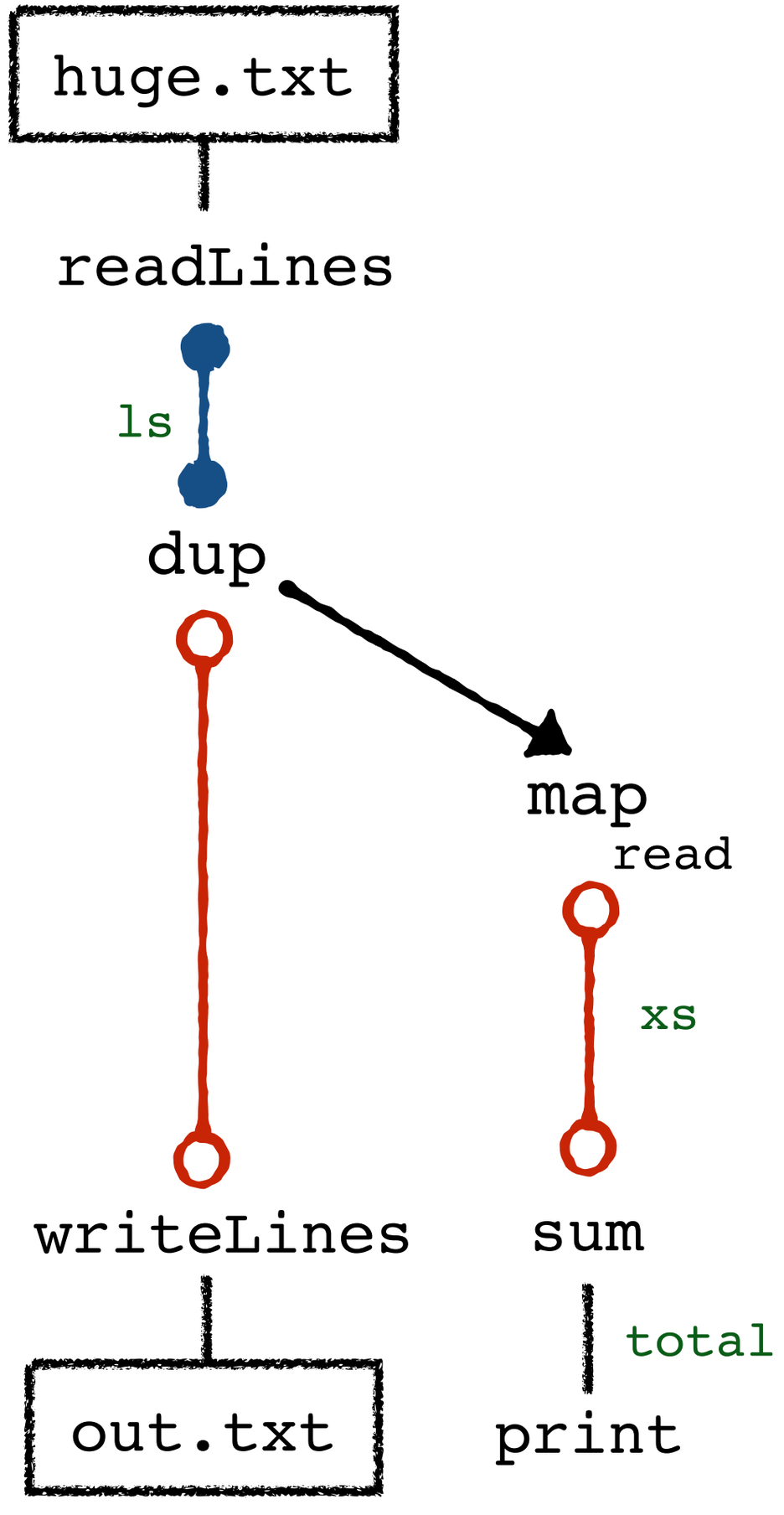
```
do ls <- readLines “huge.txt”
   writeLines “out.txt” ls
   print (sum (map read ls))
```

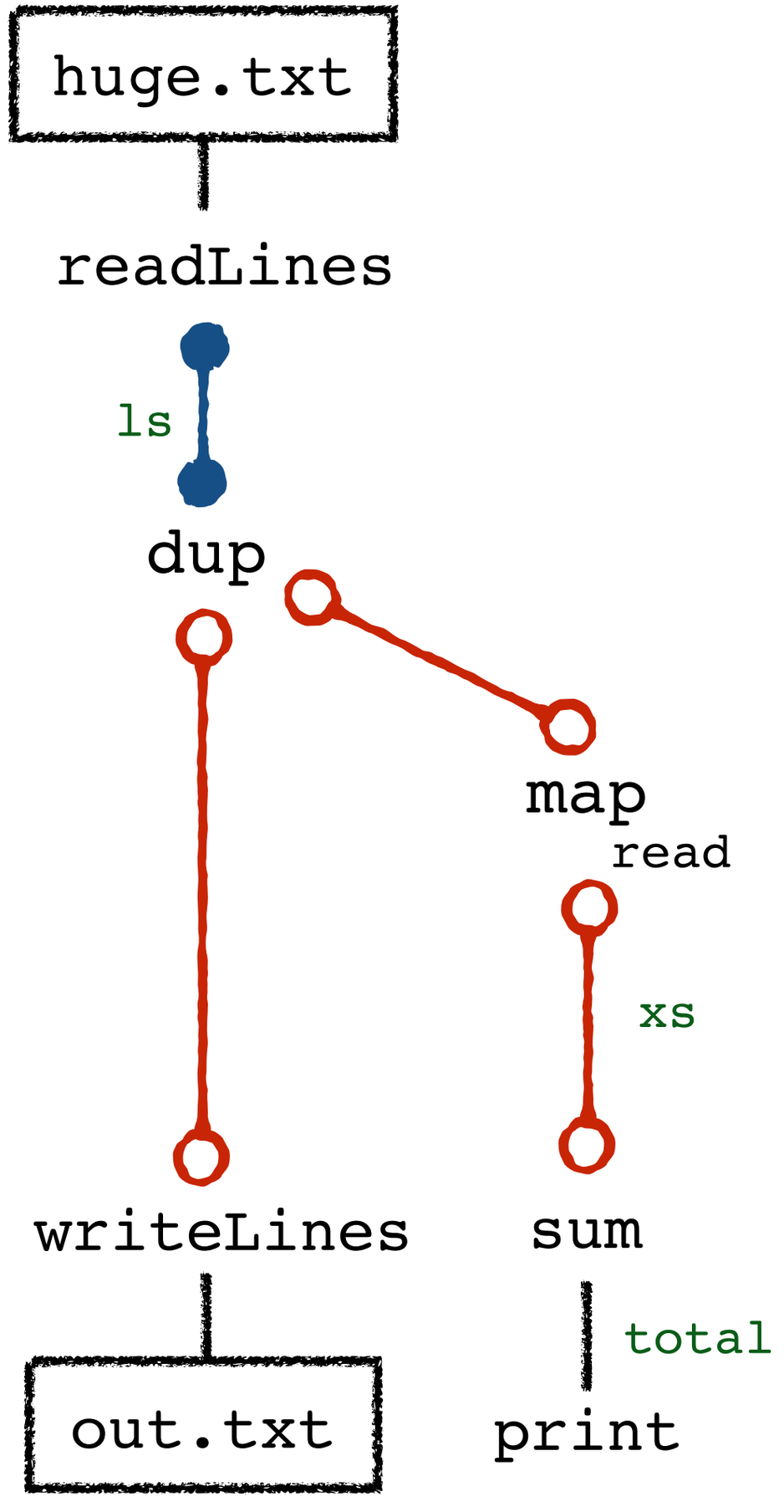

huge.txt
readLines
ls
map
read
xs
writeLines
sum
total
out.txt
print

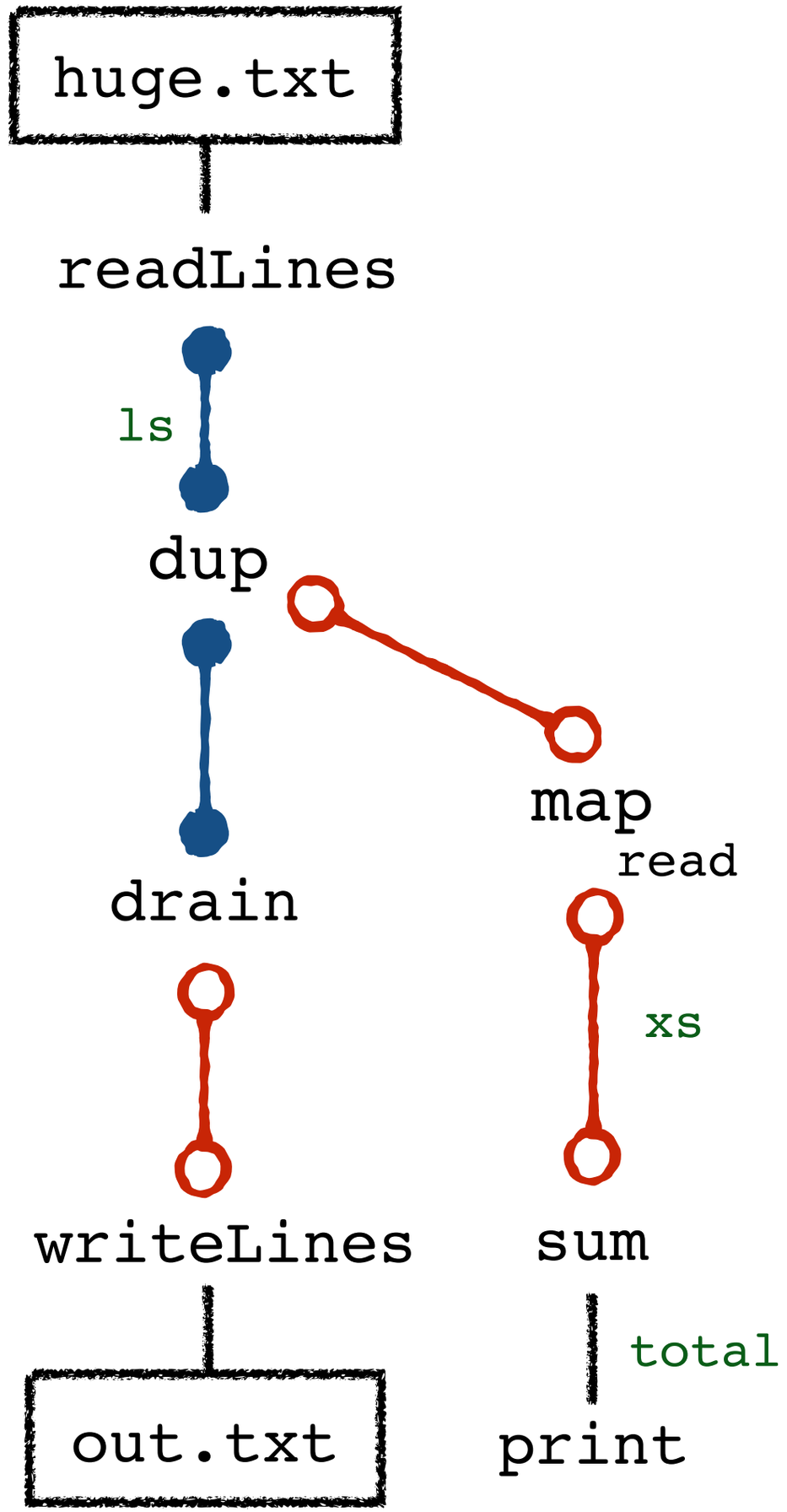

huge.txt
readLines
ls
dup
map
read
xs
sum
total
print
writeLines
out.txt

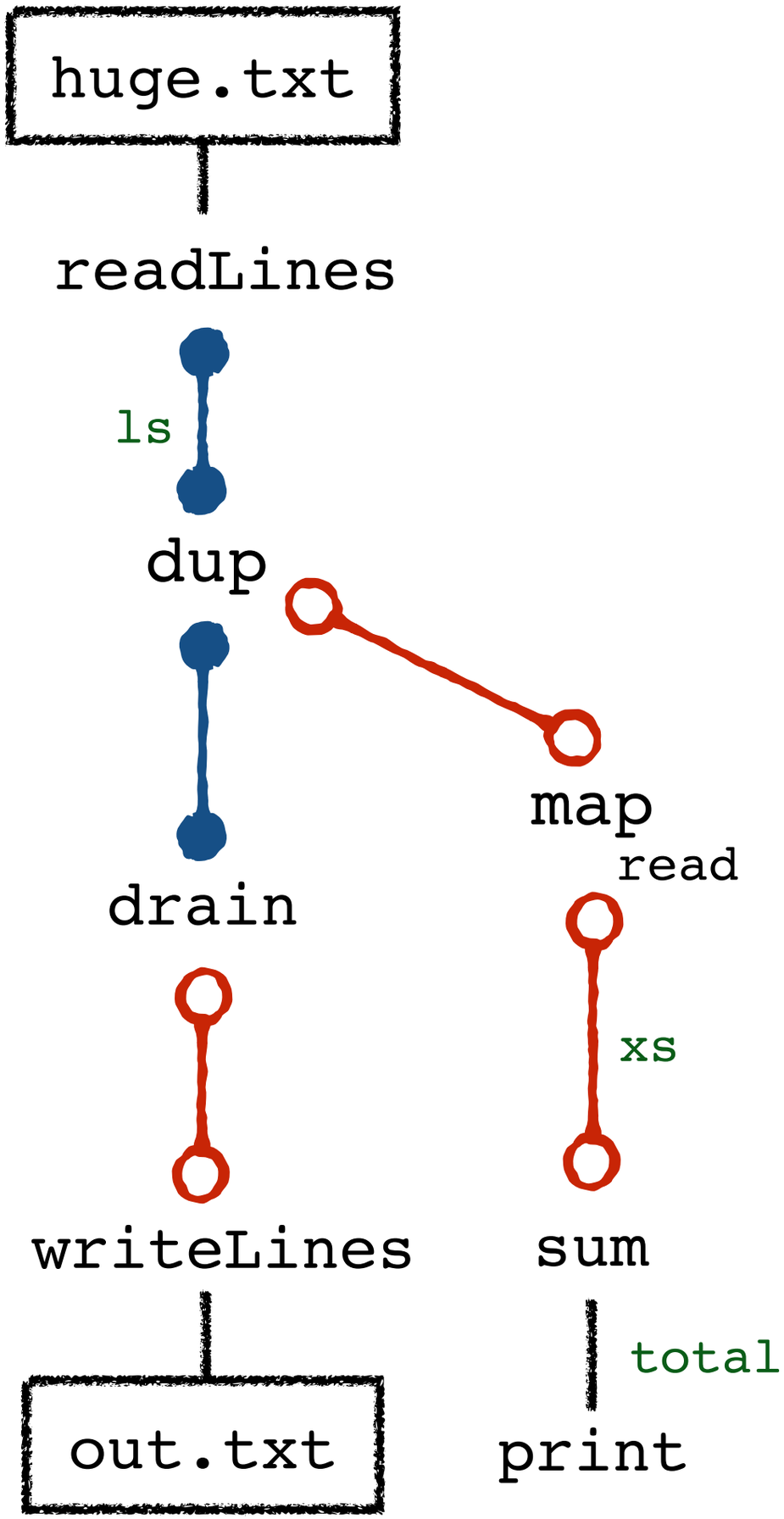

huge.txt
readLines
ls
dup
map
read
xs
writeLines
sum
total
out.txt
print

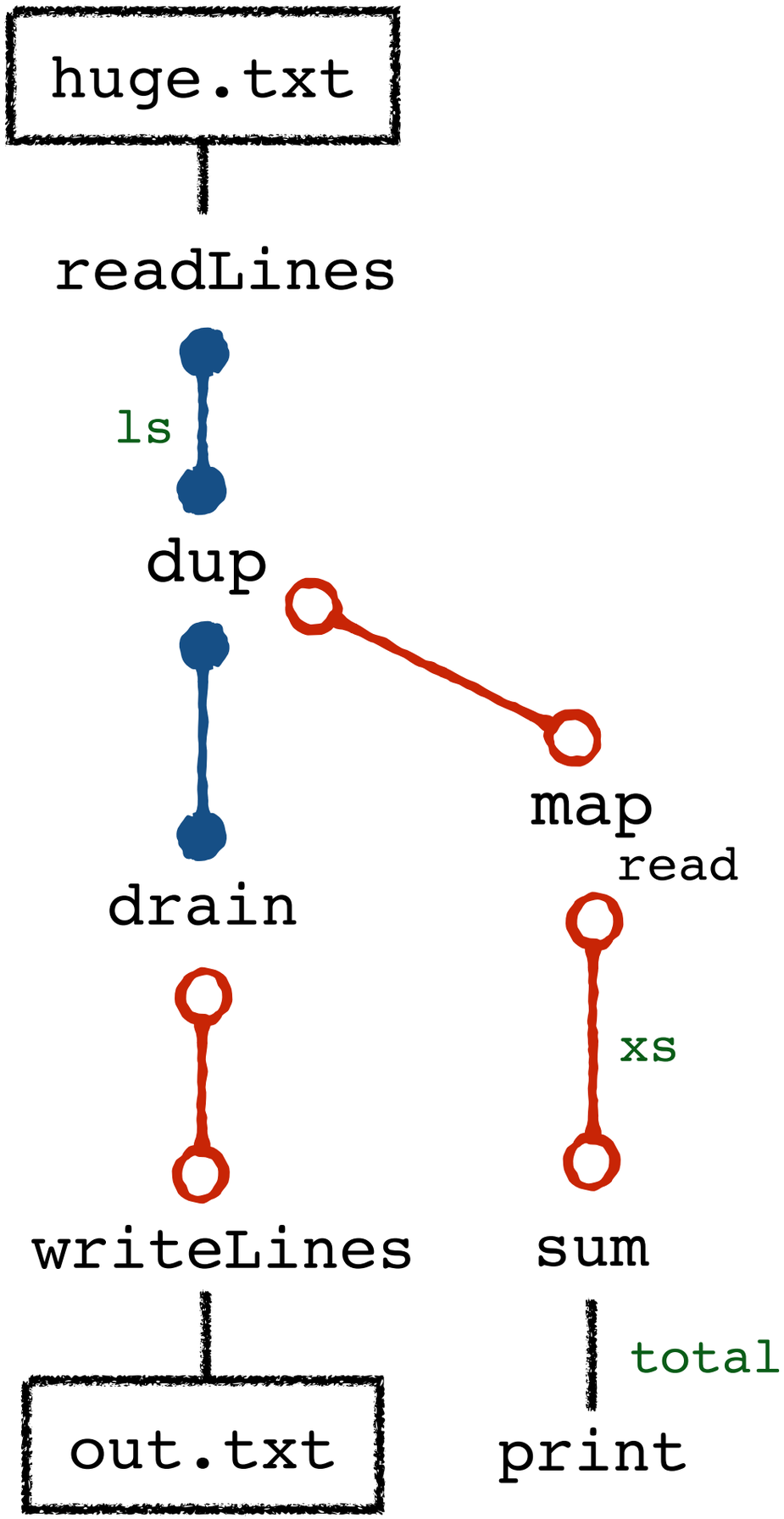

huge.txt
readLines
ls
dup
map
read
xs
sum
total
print
writeLines
out.txt


huge.txt
readLines
ls
dup
map
read
xs
writeLines
sum
total
out.txt
print

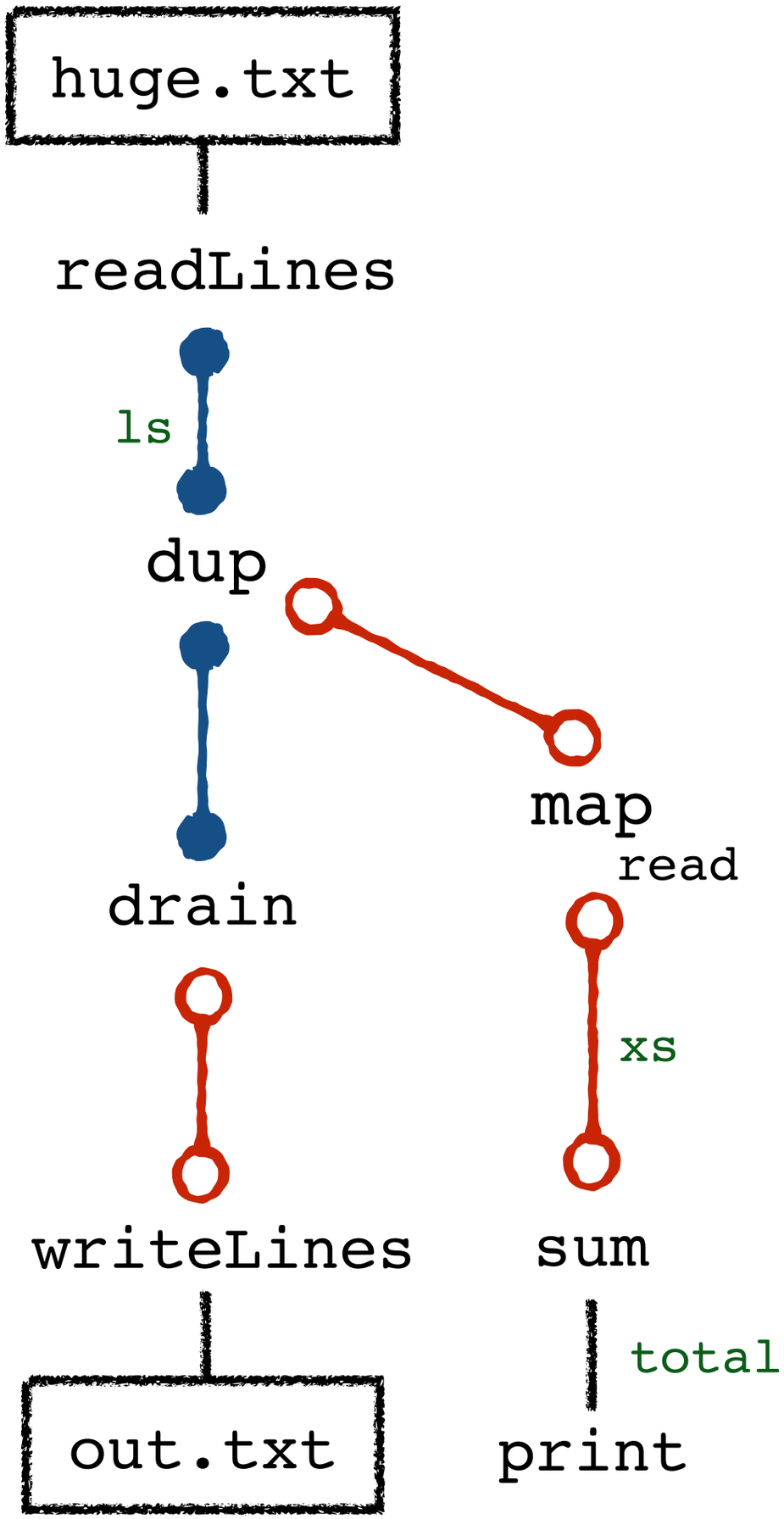

huge.txt
readLines
ls
dup
map
read
xs
writeLines
sum
out.txt
total
print


huge.txt
readLines
ls
dup
drain
writeLines
out.txt
map
read
xs
sum
total
print

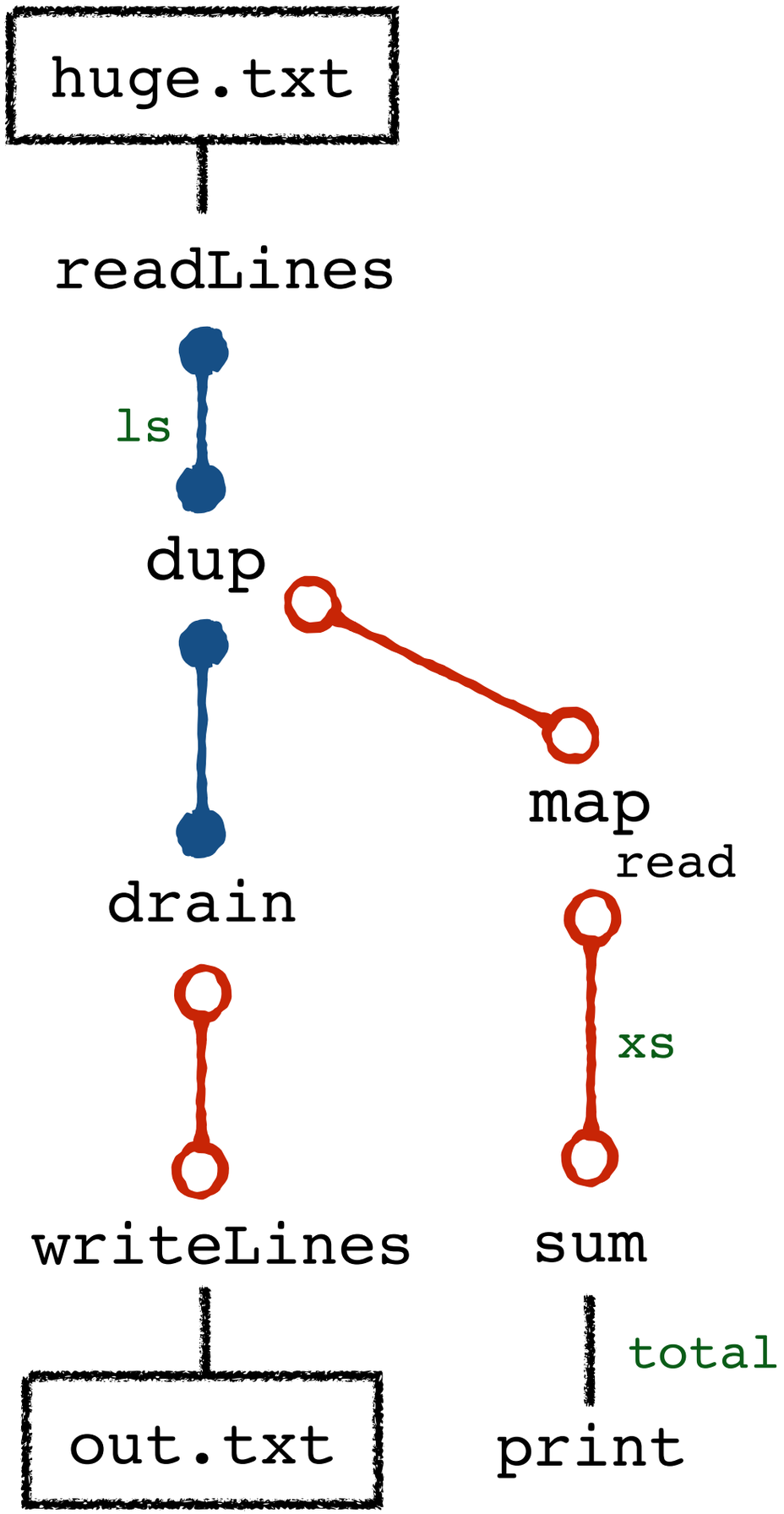

huge.txt
readLines
ls
dup
drain
writeLines
out.txt
map
read
xs
sum
total
print



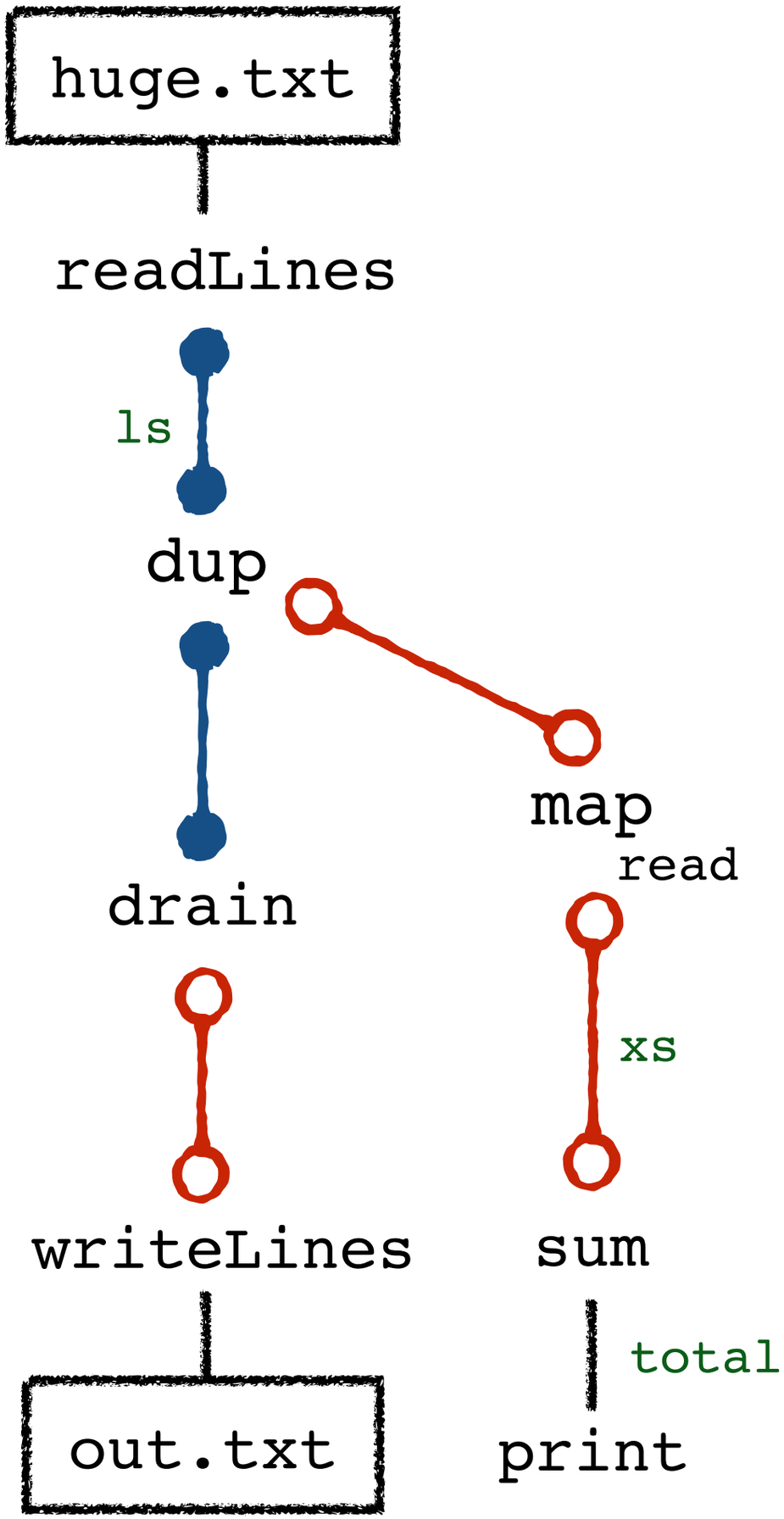

```
data Source m a = …
data SinK   m a = …
```



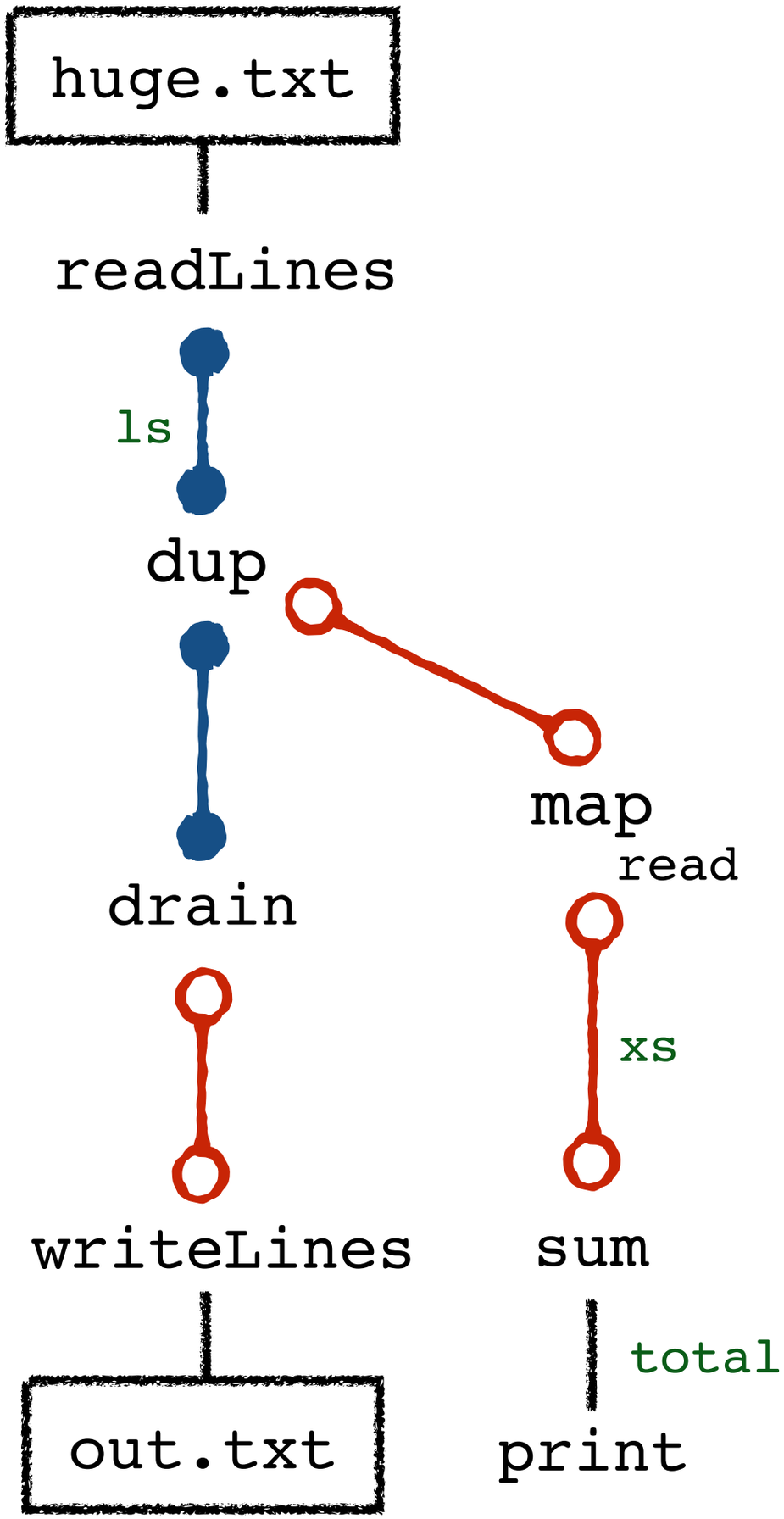

```
data Source m a = …
data SinK   m a = …

readLines  : Name -> m (S String)
writeLines : Name -> m (K String)
```



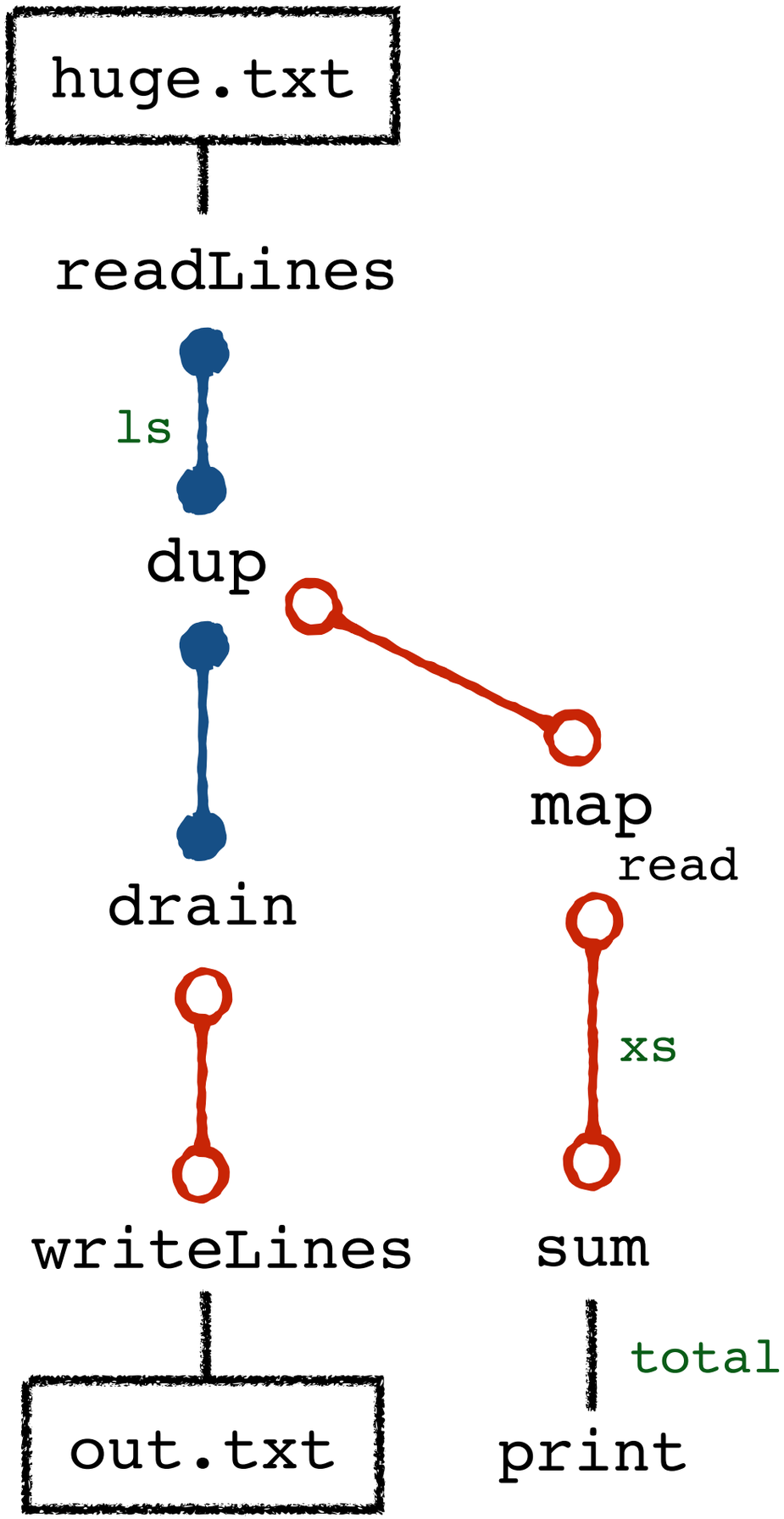

```
data Source m a = …
data SinK   m a = …

readLines  : Name -> m (S String)
writeLines : Name -> m (K String)
dup        : S a -> K a -> S a
```



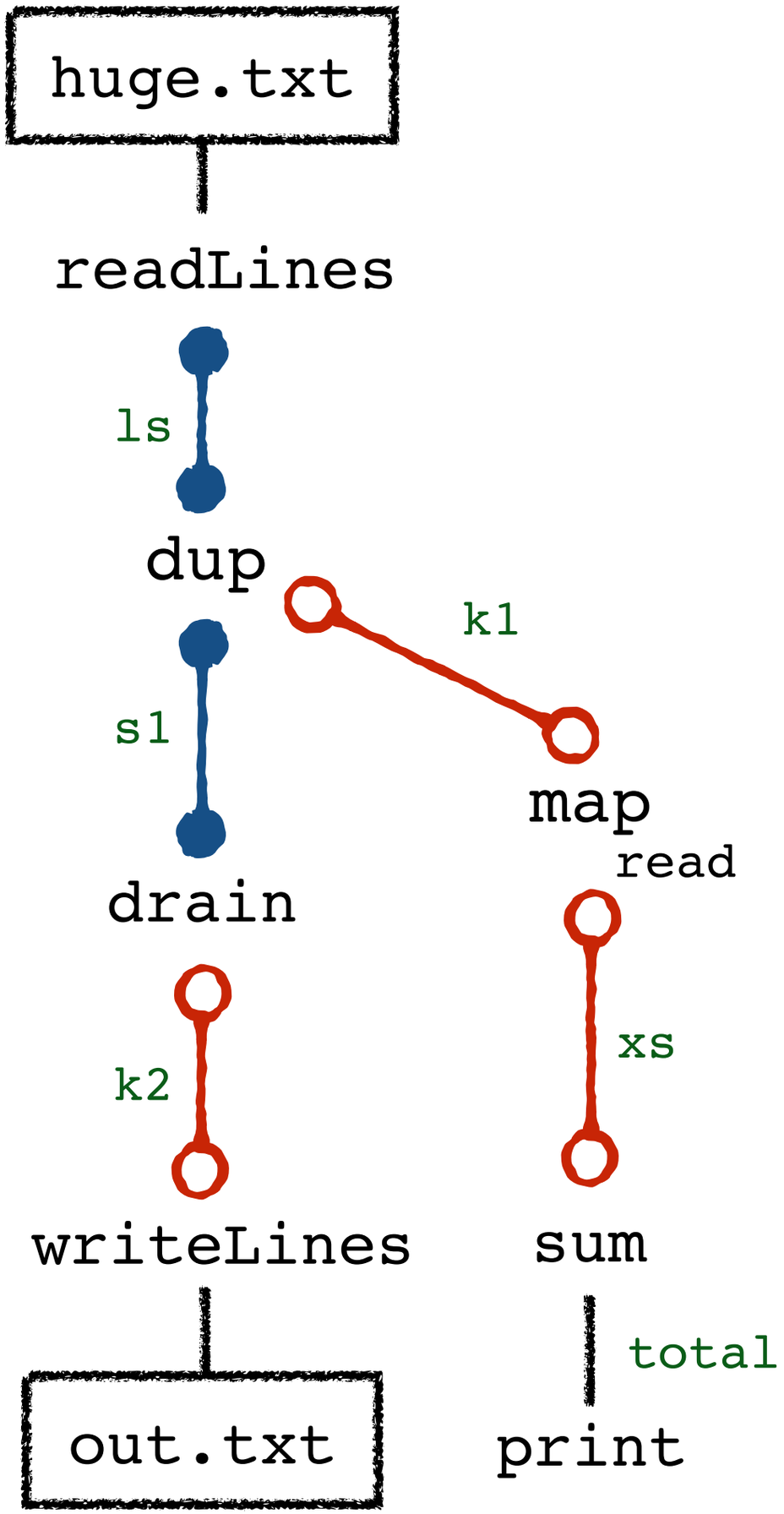

```
data Source m a = …
data SinK   m a = …

readLines  : Name -> m (S String)
writeLines : Name -> m (K String)
dup        : S a -> K a -> S a
sum        : m (K a, Ref Nat)
```




```
data Source m a = …
data SinK   m a = …

readLines  : Name -> m (S String)
writeLines : Name -> m (K String)
dup        : S a -> K a -> S a
sum        : m (K a, Ref Nat)
map   : (a -> b) -> K b -> K a
```

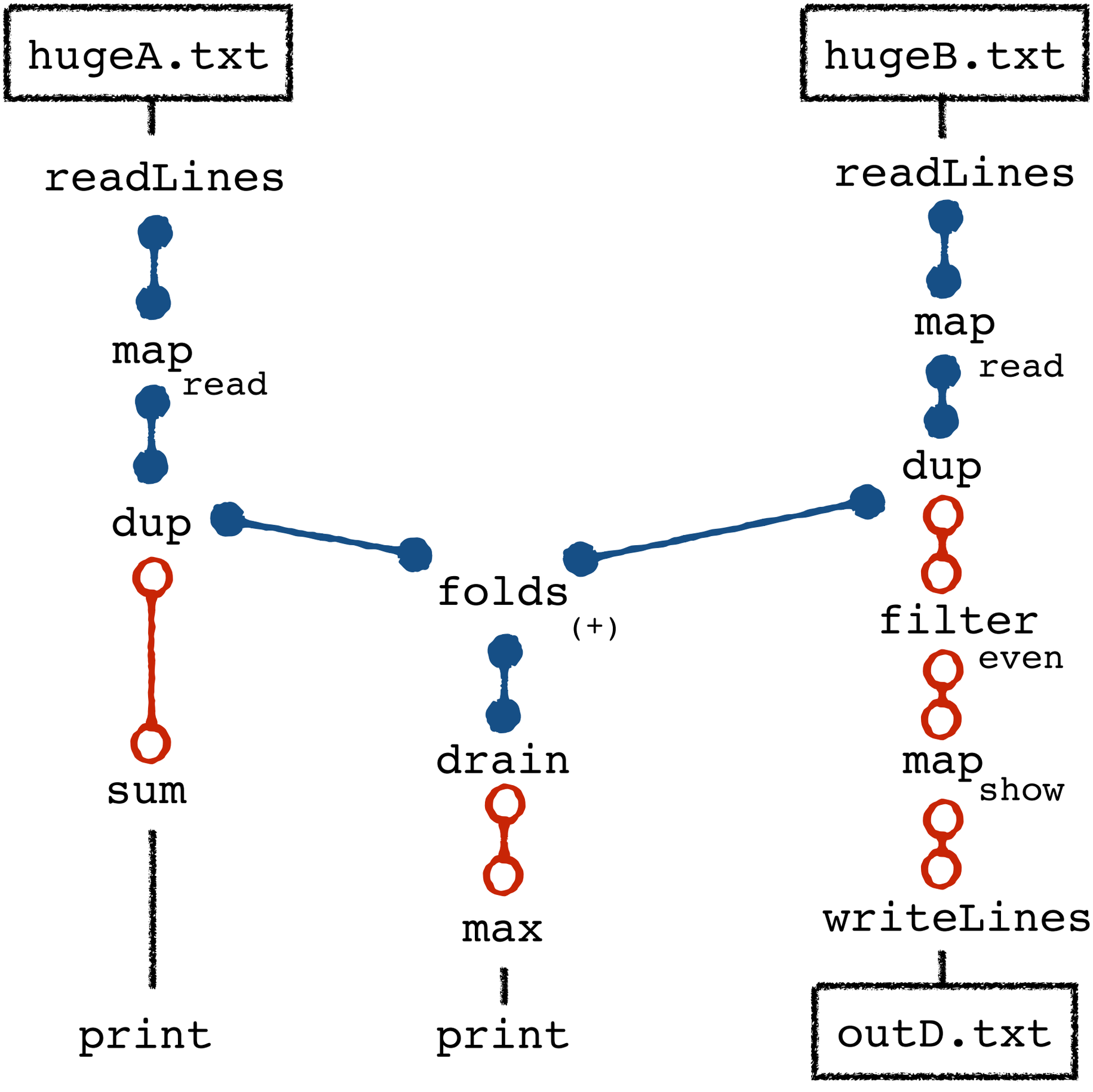


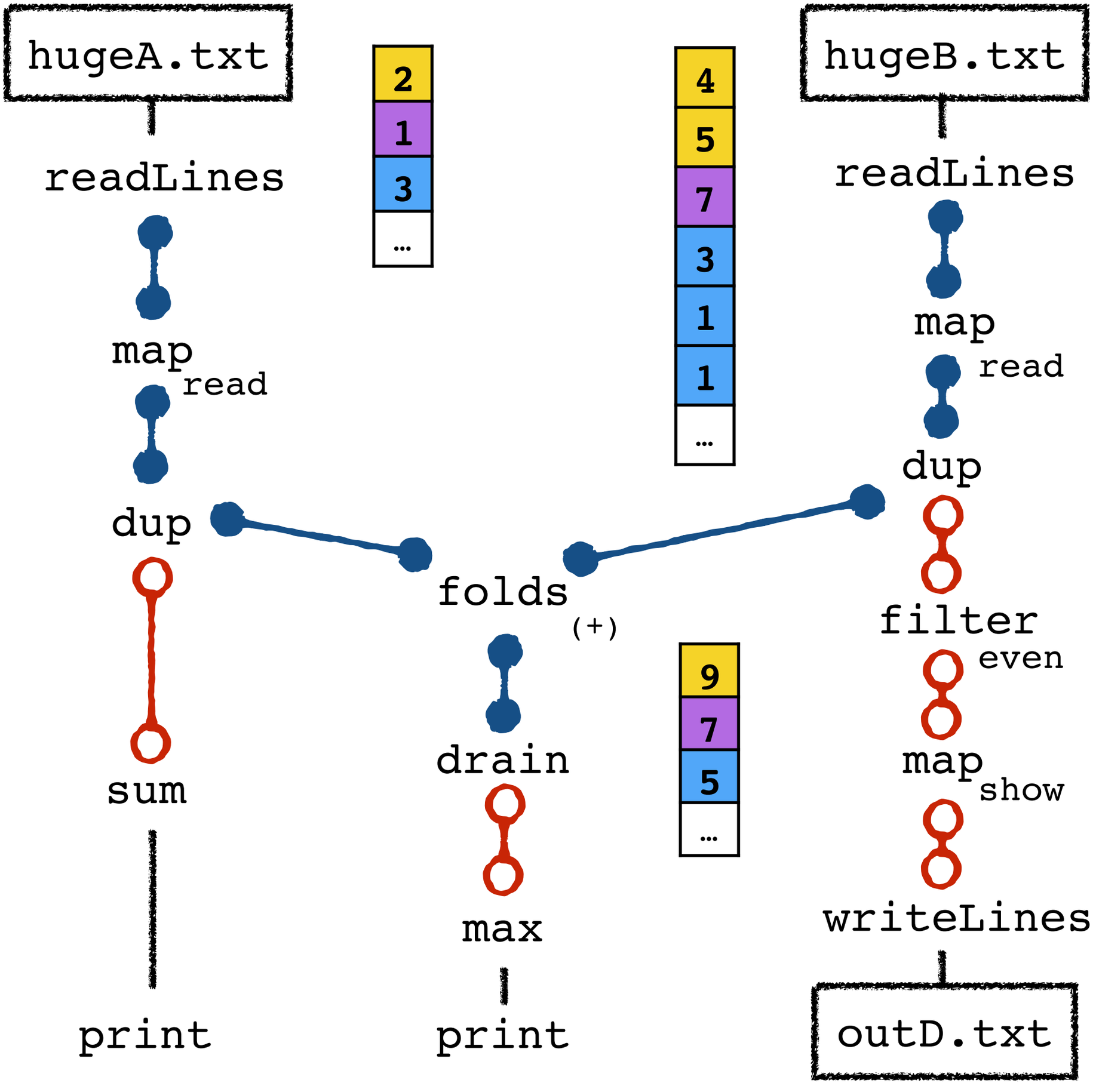

```
data Source m a = …
data SinK   m a = …

readLines  : Name -> m (S String)
writeLines : Name -> m (K String)
dup        : S a -> K a -> S a
sum        : m (K a, Ref Nat)
map   : (a -> b) -> K b -> K a
drain      : S a -> K a -> m ()
```




```
data Source m a = …
data SinK   m a = …

readLines  : Name -> m (S String)
writeLines : Name -> m (K String)
dup        : S a -> K a -> S a
sum        : m (K a, Ref Nat)
map   : (a -> b) -> K b -> K a
drain      : S a -> K a -> m ()
```

```
do ls    <- readLines “huge.txt”
   (ref, xs) <- sum
   k1    <- map read xs
   k2    <- writeLines “out.txt”
   s1    <- dup ls k1
   drain s1 k2
   total <- readRef ref
   print total
```

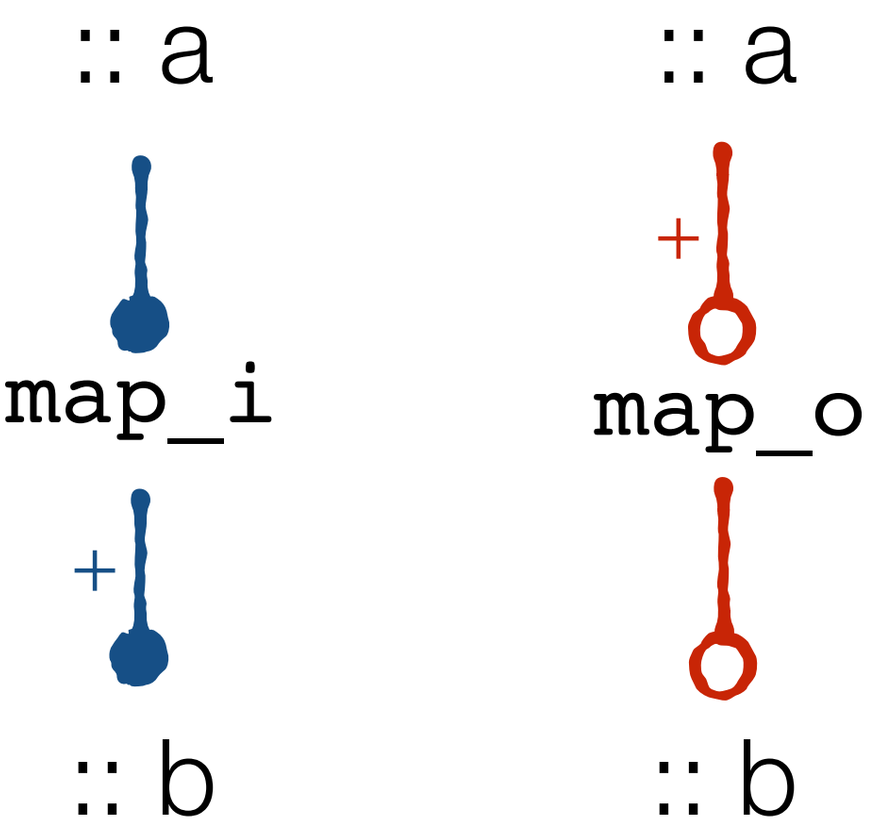

hugeA.txt
readLines
map
read
dup
sum
print
folds
(+)
drain
max
print
hugeB.txt
readLines
map
read
dup
filter
even
map
show
writeLines
outD.txt


hugeA.txt
readLines
map
read
dup
sum
print
2
1
3
…
4
5
7
3
1
1
…
folds
(+)
drain
max
print
9
7
5
…
hugeB.txt
readLines
map
read
dup
filter
even
map
show
writeLines
outD.txt

# “Polarity Versions”

pull from output
induces
pull from input

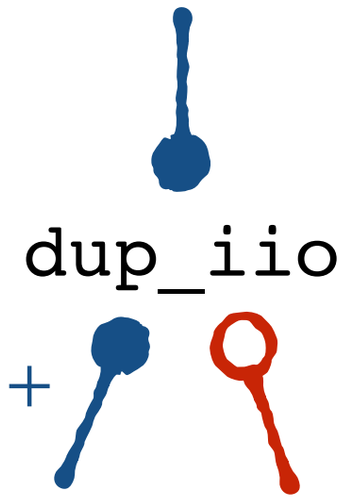
“pully”

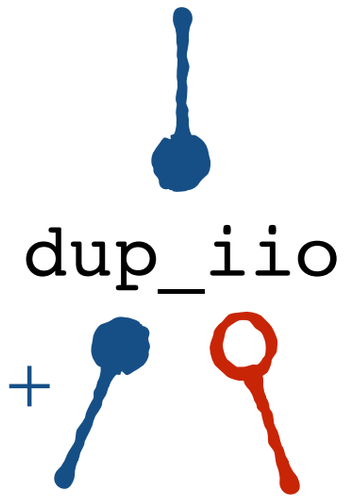


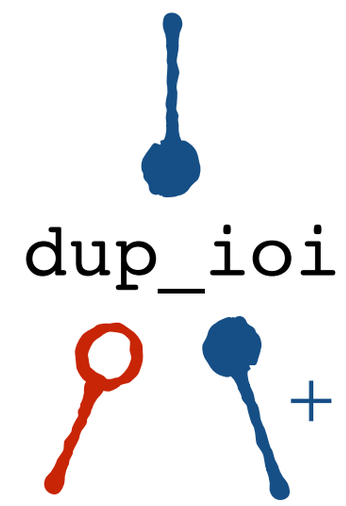

push to input
induces
push to output

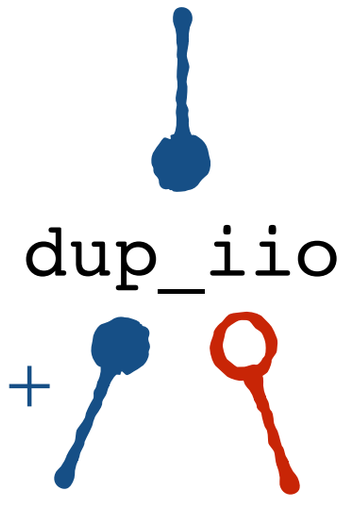
“pushy”

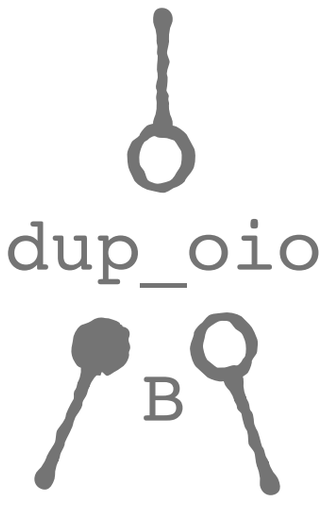
```
map_i   :: (a -> b) -> Source a -> m (Source b)
map_o   :: (a -> b) -> Sink   b -> m (Sink   a)
```

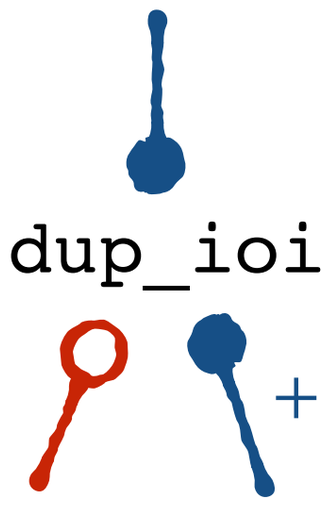

dup_iio

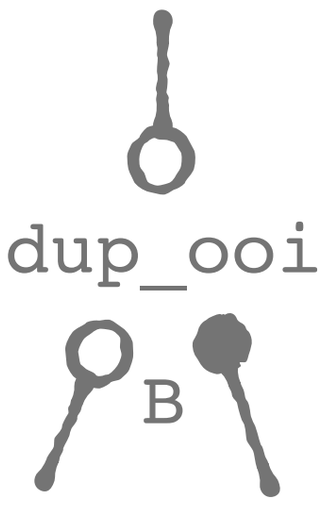

dup_iio

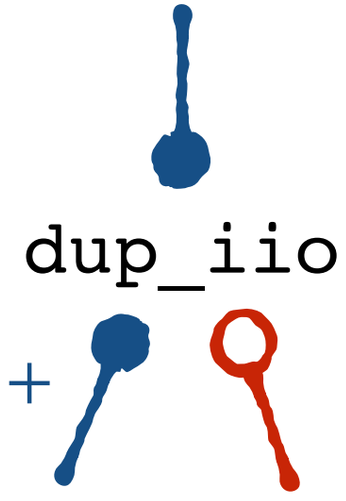

dup_ioi

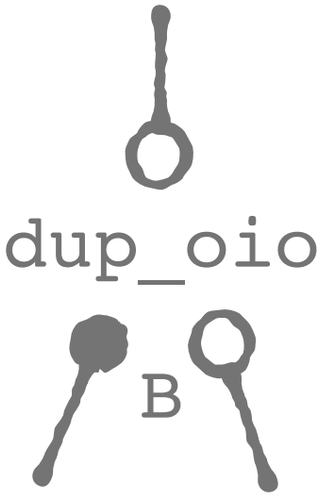

dup_iio
+

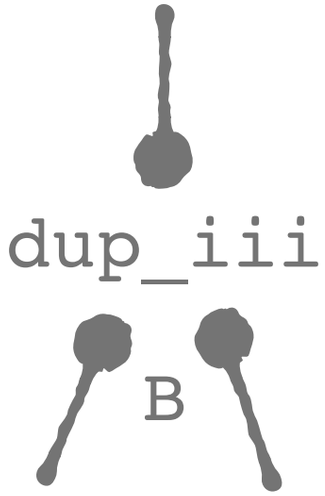

dup_oio
B

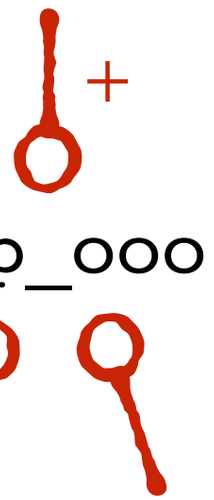

dup_ioi
+

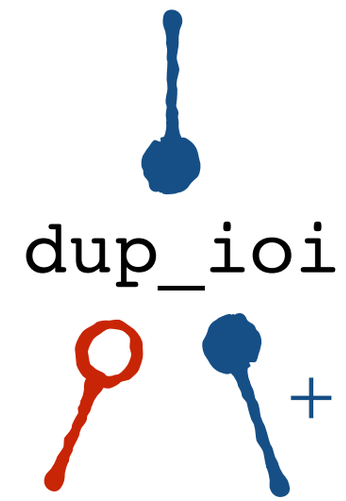

dup_ooi
B

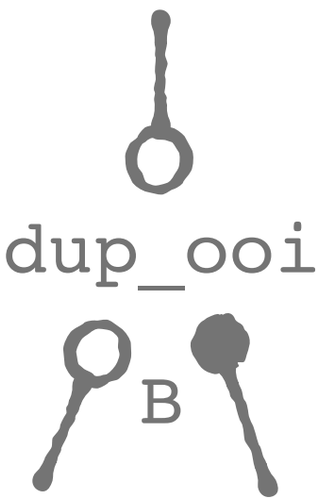

dup_iio
+

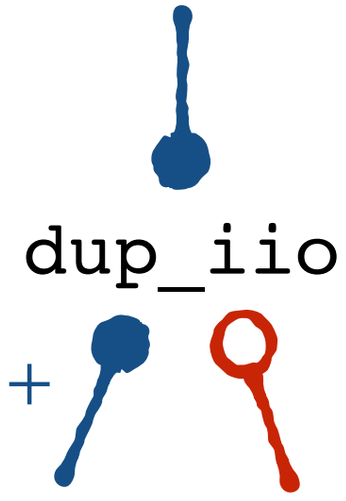

dup_oio
B

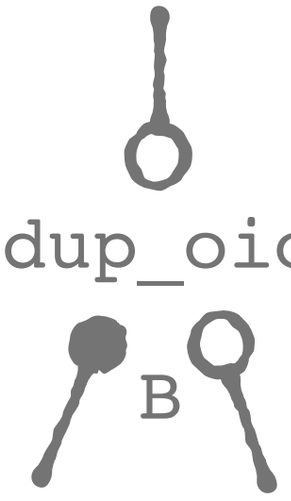

dup_iii
B

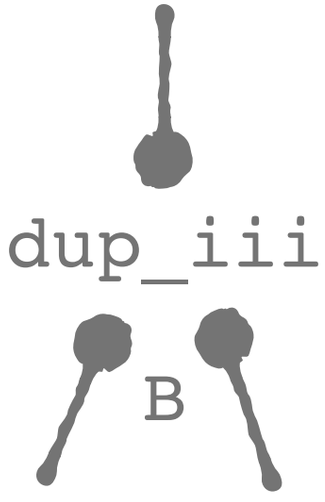

+
dup_ooo

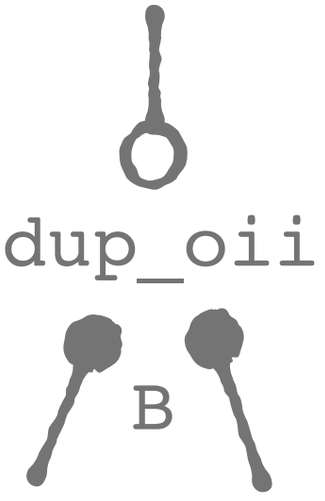

dup_ioi
+

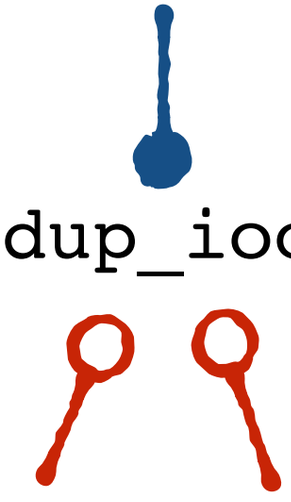

dup_ooi
B

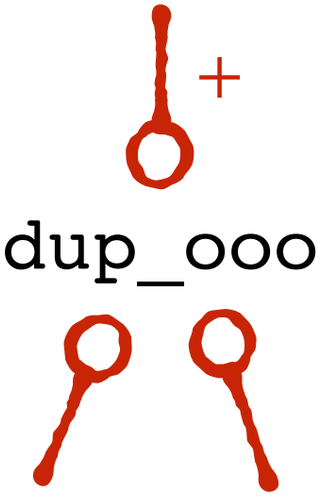

dup_iio
+

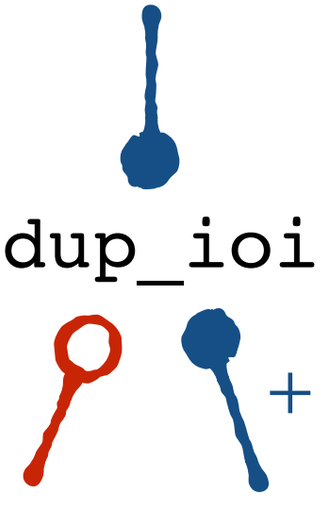

dup_oio
B

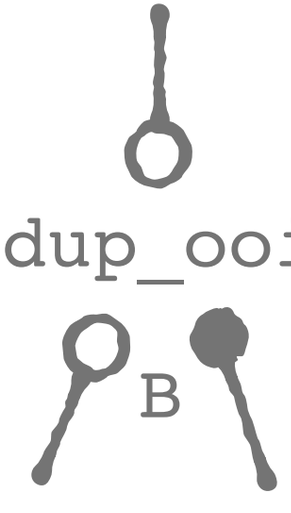

dup_iii
B


dup_oii
B

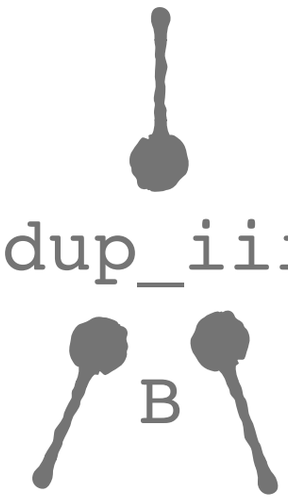

dup_ioo


dup_ooo
+

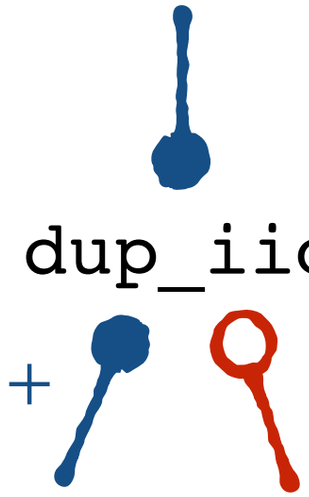

dup_ioi
+


dup_ooi
B

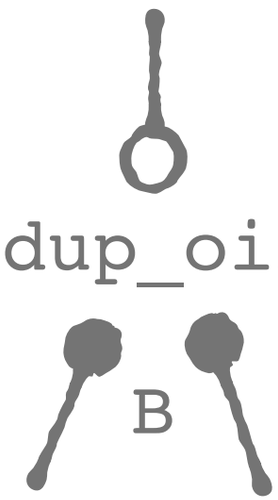
dup_iii



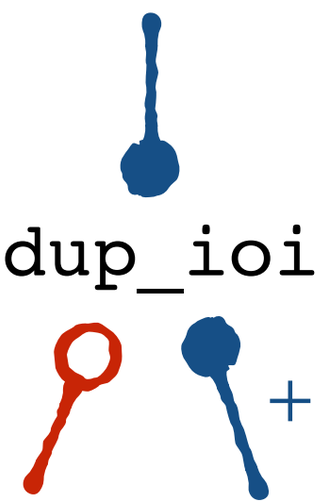

dup_iio




dup_oii

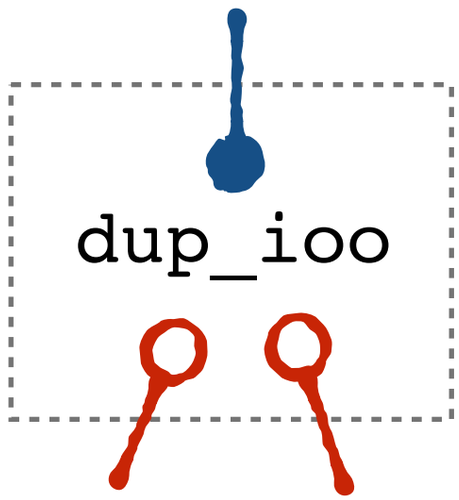



dup_ioi

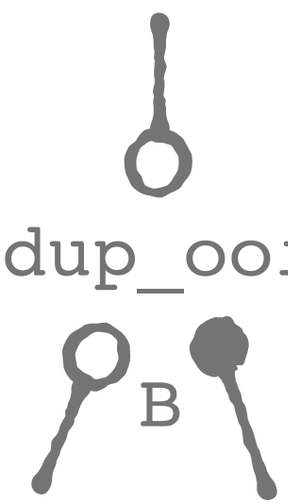



dup_oio

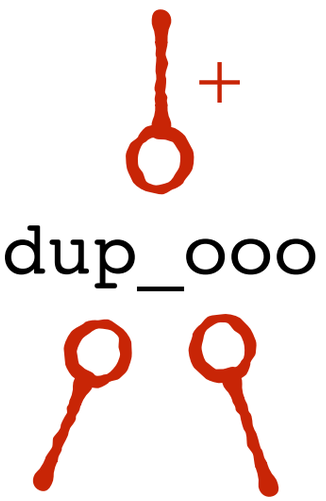

dup_ioo

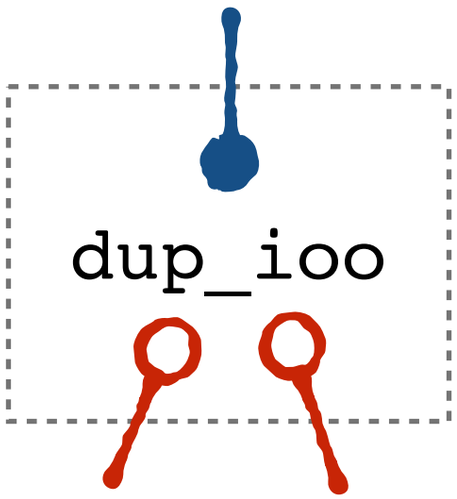


dup_ooi

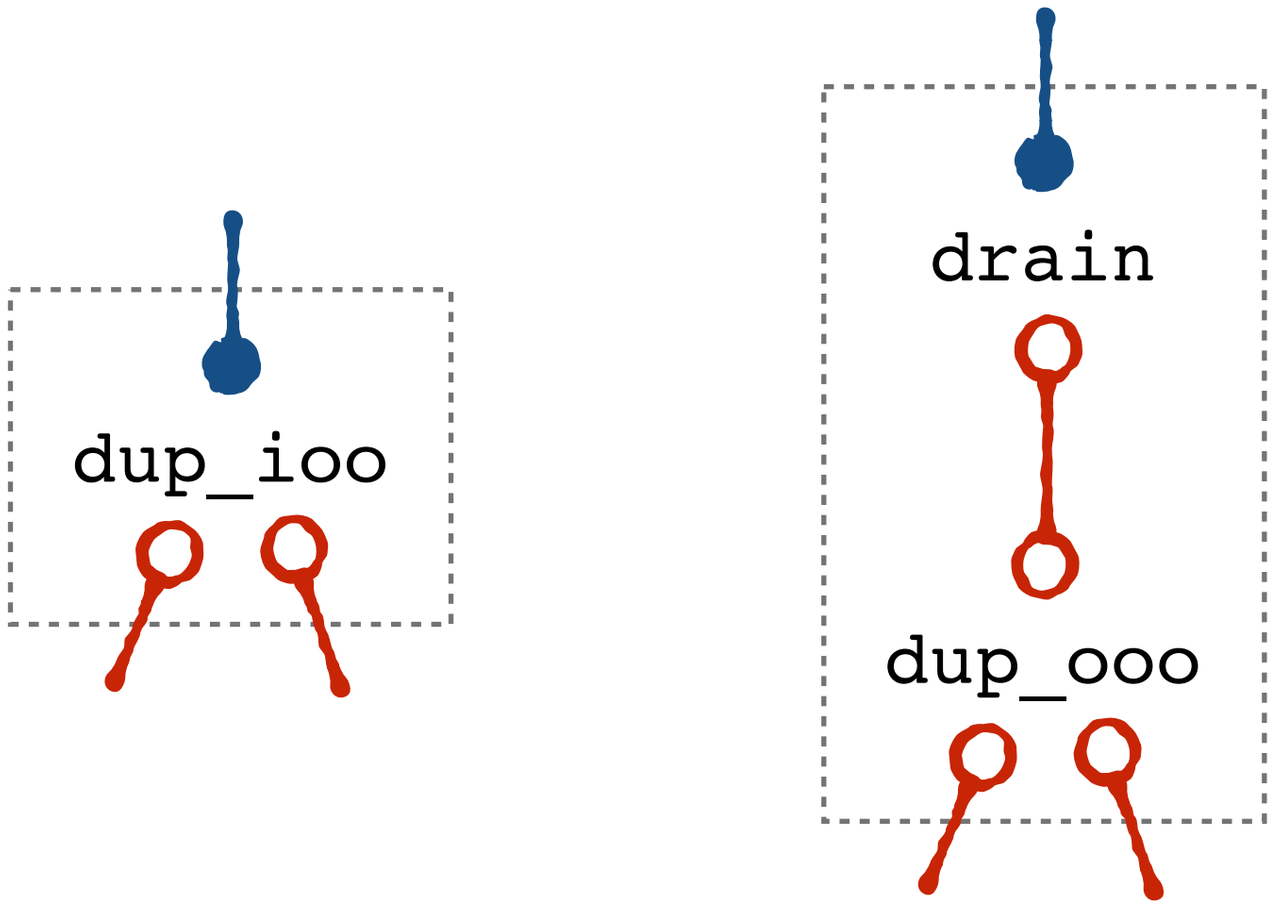


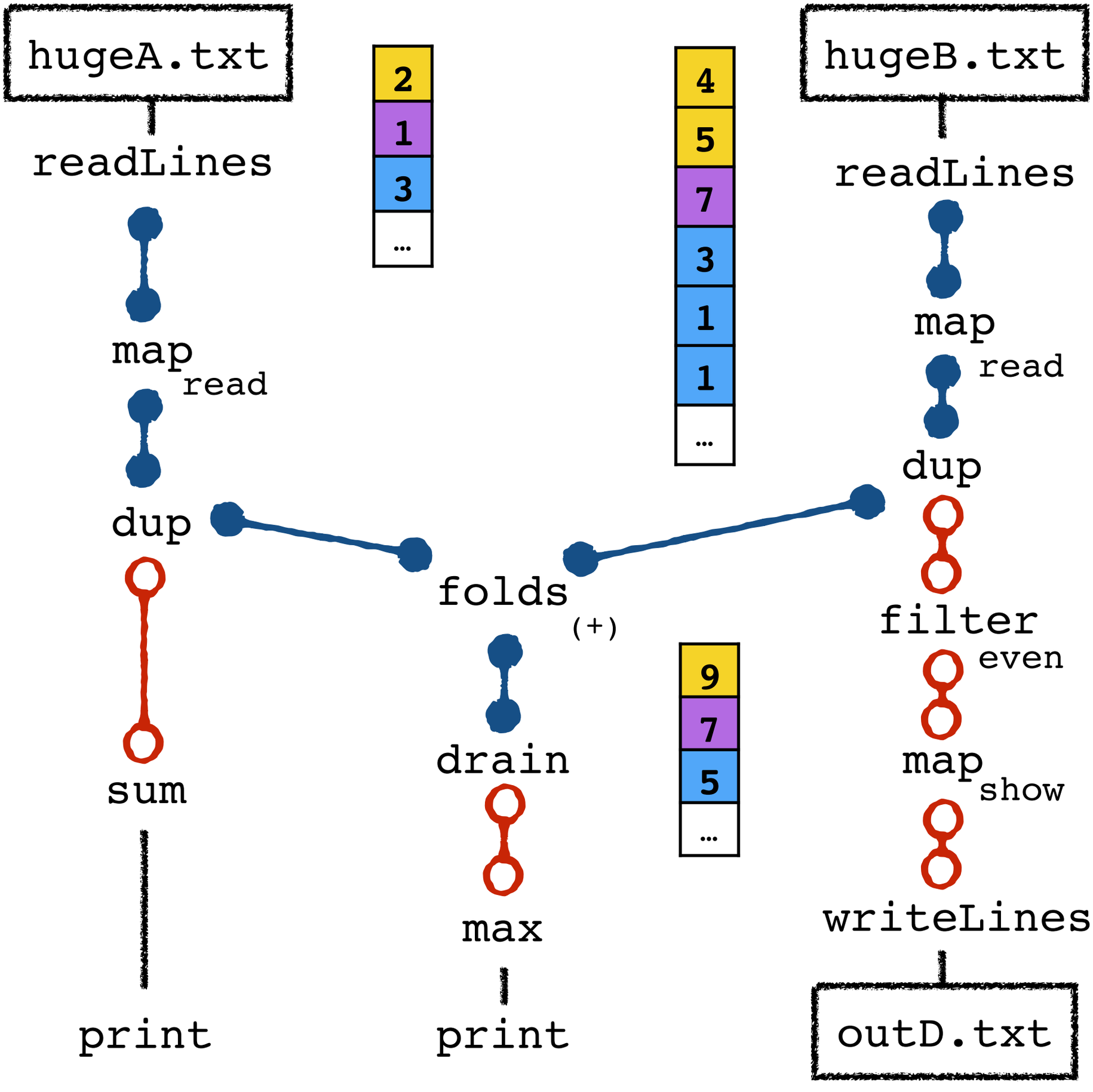

dup_ooo

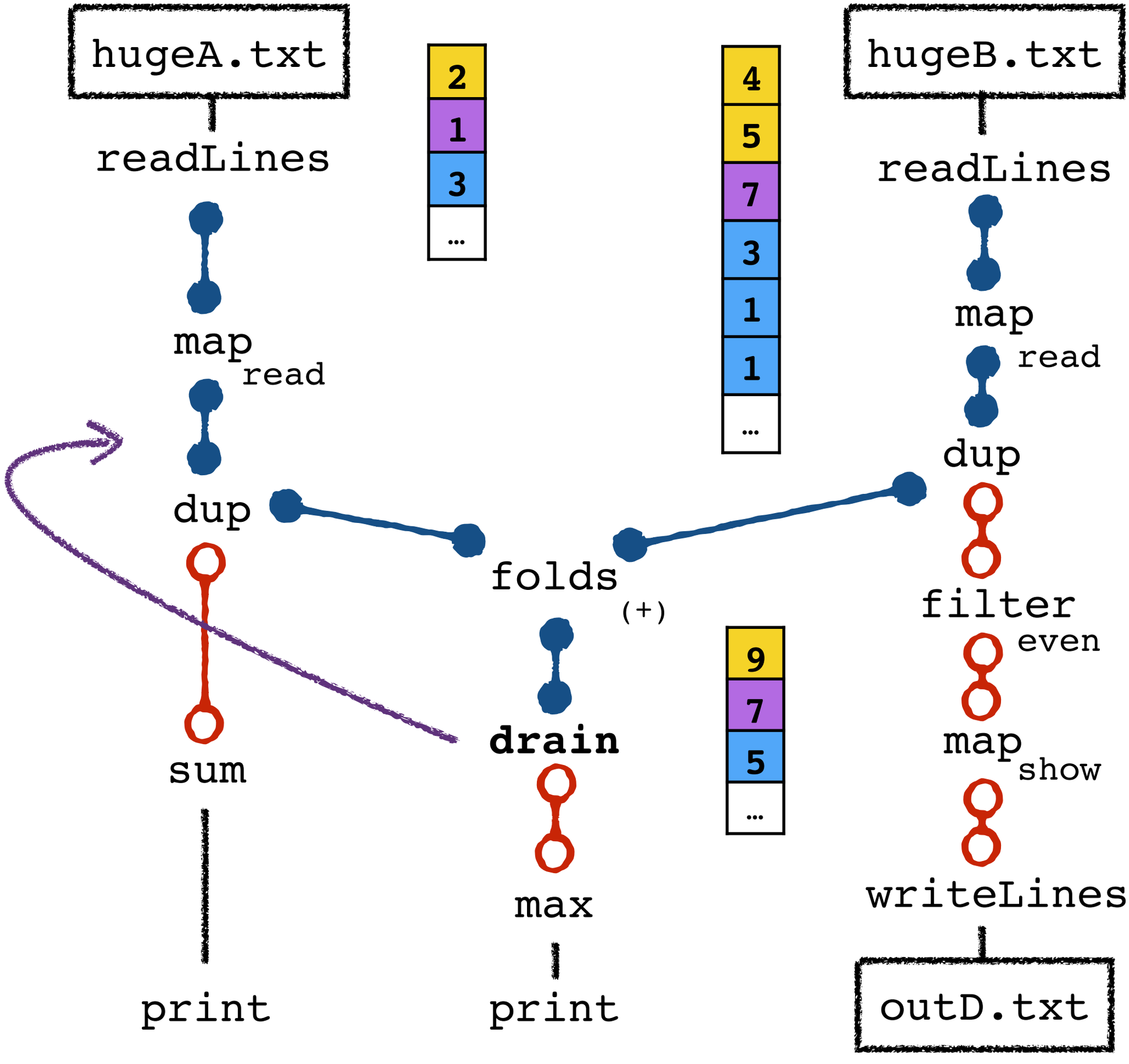


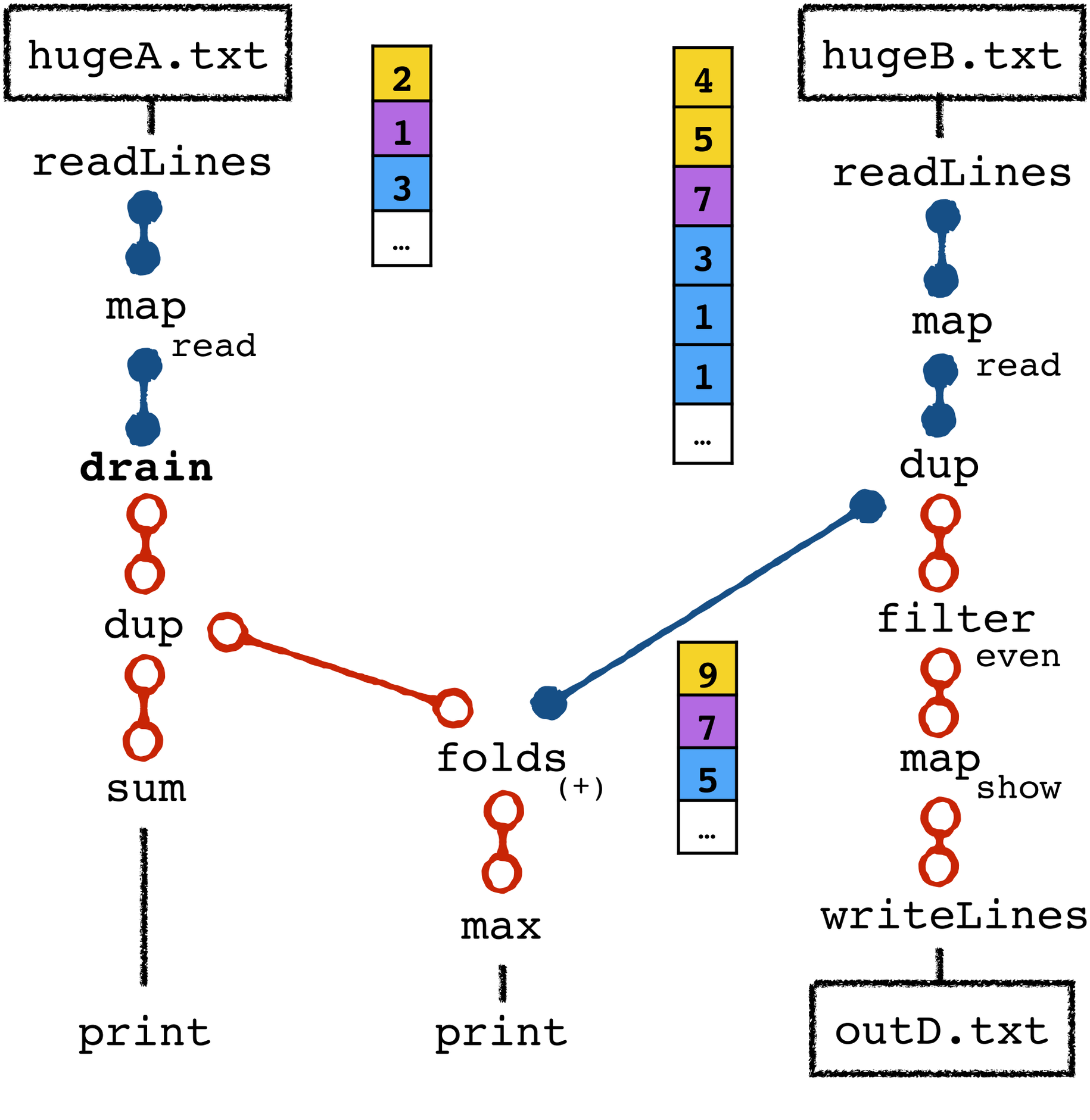

dup_ioo

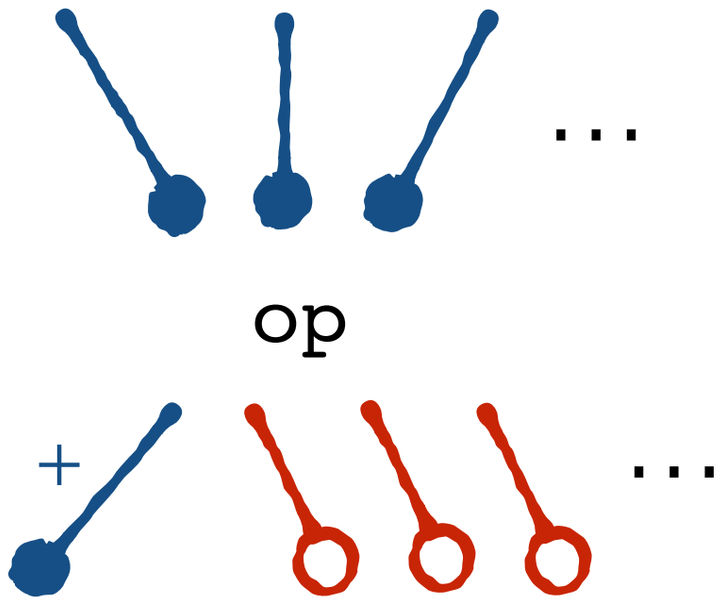
"Active Version"



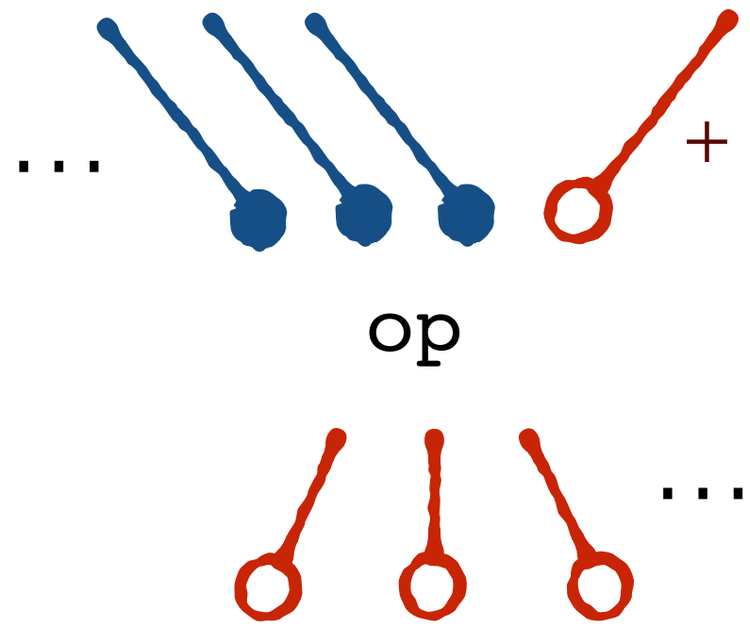

hugeA.txt
readLines
map
read
dup
sum
print
2
1
3
…
4
5
7
3
1
1
…
folds
(+)
drain
max
print
9
7
5
…
hugeB.txt
readLines
map
read
dup
filter
even
map
show
writeLines
outD.txt


hugeA.txt
readLines
map
read
dup
sum
print
2
1
3
…
4
5
7
3
1
1
…
folds
(+)
drain
max
print
9
7
5
…
hugeB.txt
readLines
map
read
dup
filter
even
map
show
writeLines
outD.txt


hugeA.txt
readLines
map
read
drain
dup
sum
print
2
1
3
…
folds
(+)
max
print
4
5
7
3
1
1
…
9
7
5
…
hugeB.txt
readLines
map
read
dup
filter
even
map
show
writeLines
outD.txt




OK




OK

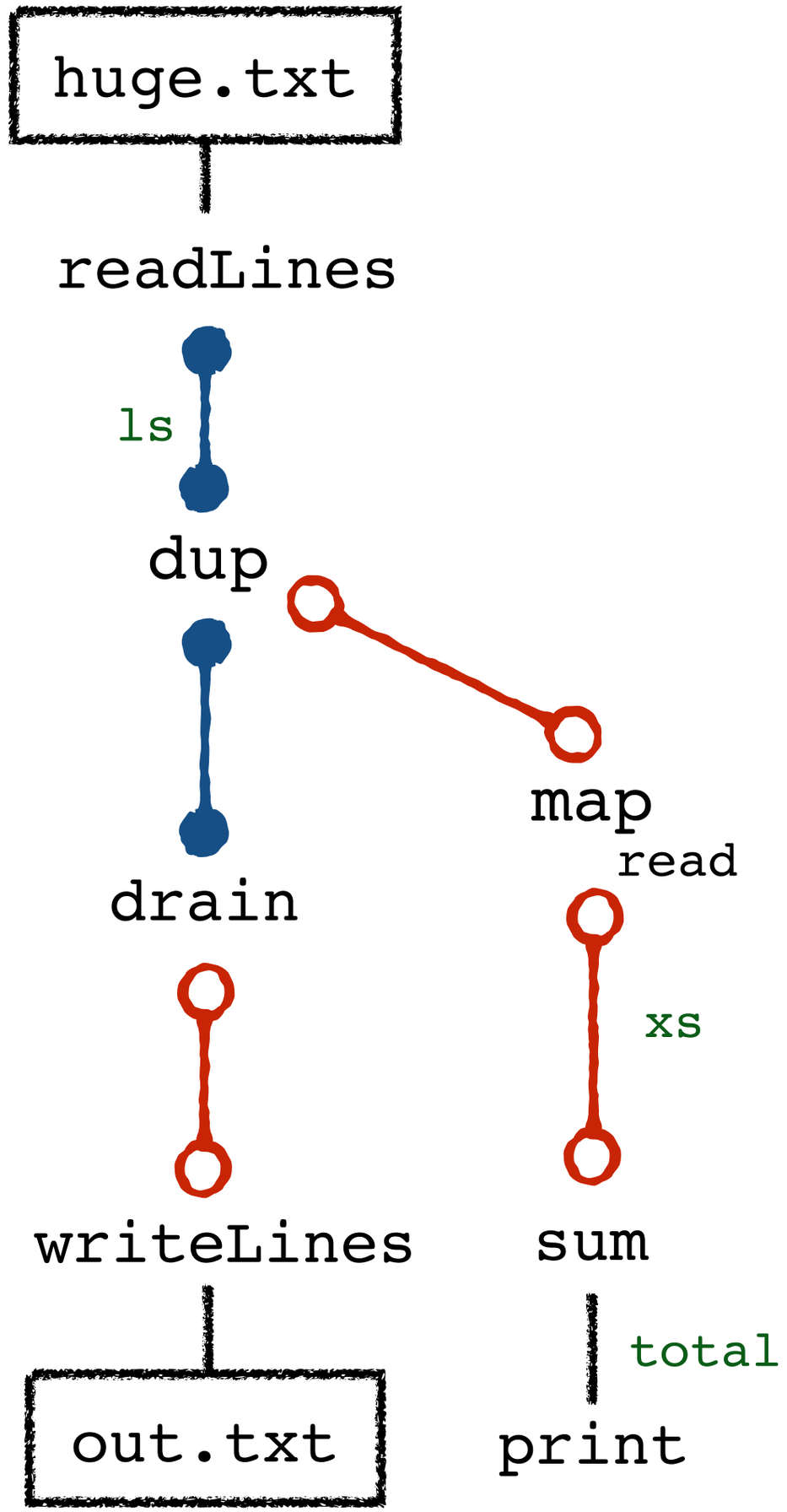
stream index  monad  element type

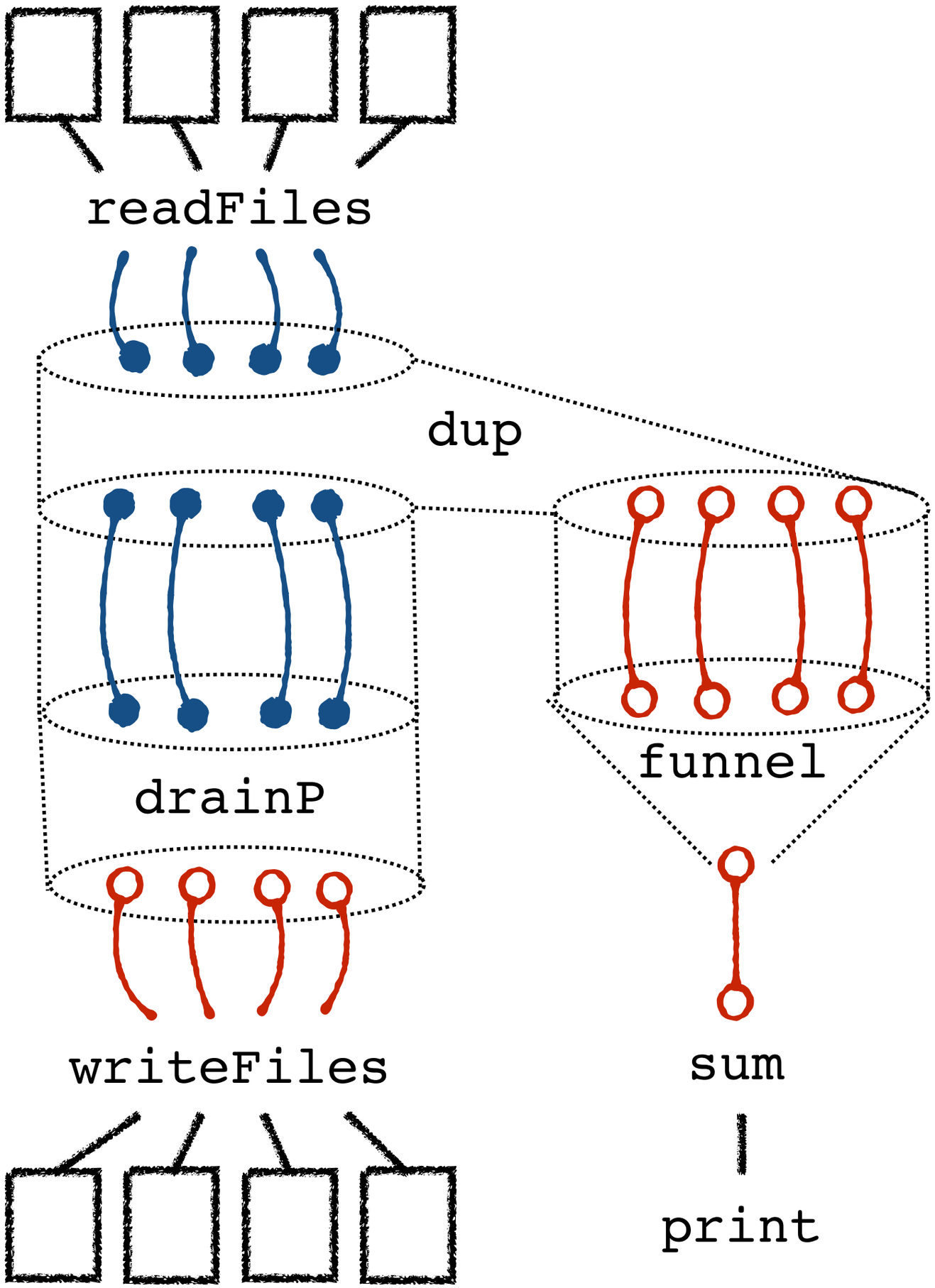
```
data Sources ix m e
    = Sources
    { sArity :: ix
    , sPull  :: ix -> (e -> m ()) -> m () -> m ()}
```

eat  eject

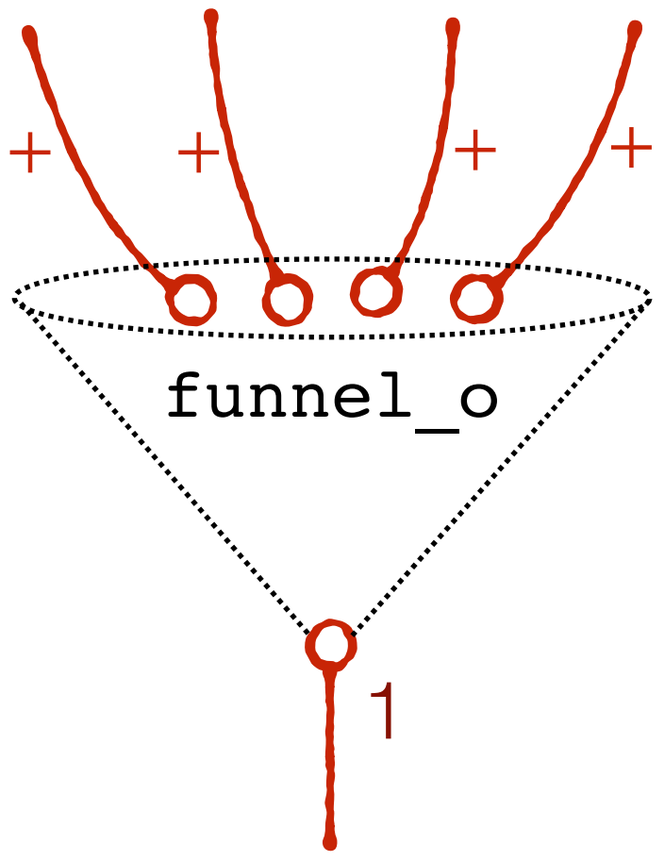
```
data Sinks ix m e
    = Sinks
    { kArity :: ix
    , kPush  :: ix -> e -> m ()
    , kEject :: ix -> m () }
```

```
map_i  :: Monad m
       => (ix -> a -> b)
       -> Sources ix m a -> m (Sources ix m b)

map_i f   (Sources n pullA)
 = return (Sources n pullB)
 where
       pullB i eatB eject
        = pullA i eatA eject
        where
              eatA x = eatB (f i x)
```


huge.txt
readLines
ls
dup
drain
writeLines
out.txt
map
read
xs
sum
total
print


readFiles
dup
drainP
funnel
writeFiles
sum
print

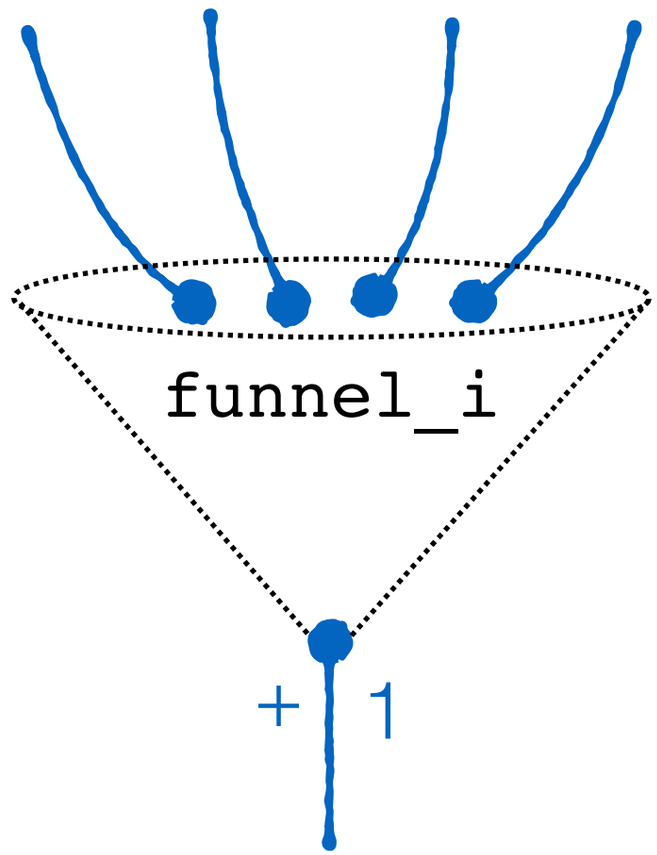
## “You push”




naturally concurrent
input streams are contending
for a shared output

uncontrolled order of consumption
elements pushed in
non-deterministic order

## “I pull”

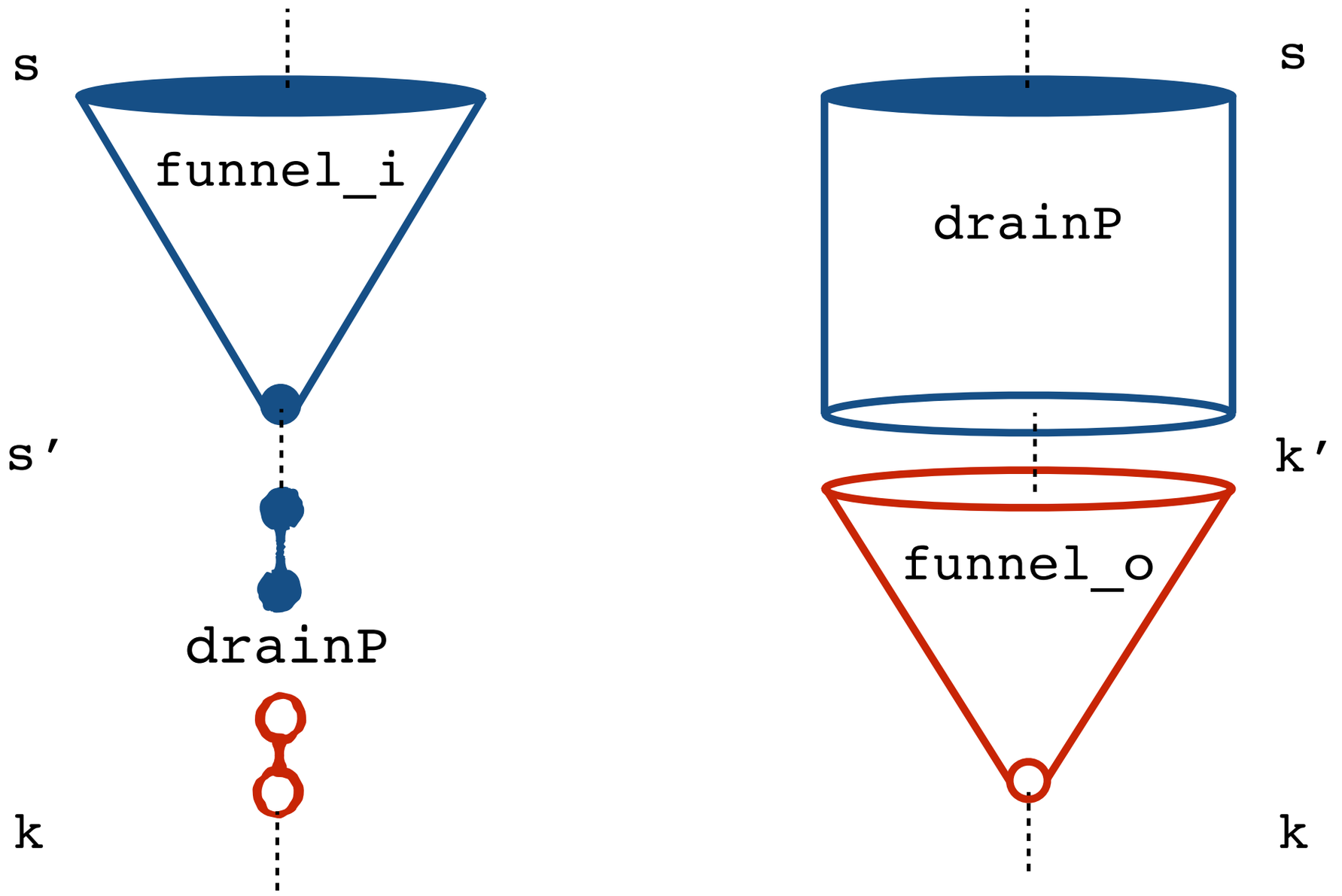



naturally sequential
read from the input streams
one after the other

controlled order of consumption
drain entire stream first,
or round robin element-wise

# “Drain Fattening”

```
   (funnel_i                 s >>= λs'. drainP s' k)
=> (funnel_o (arity s) k >>= λk'. drainP s  k')
```

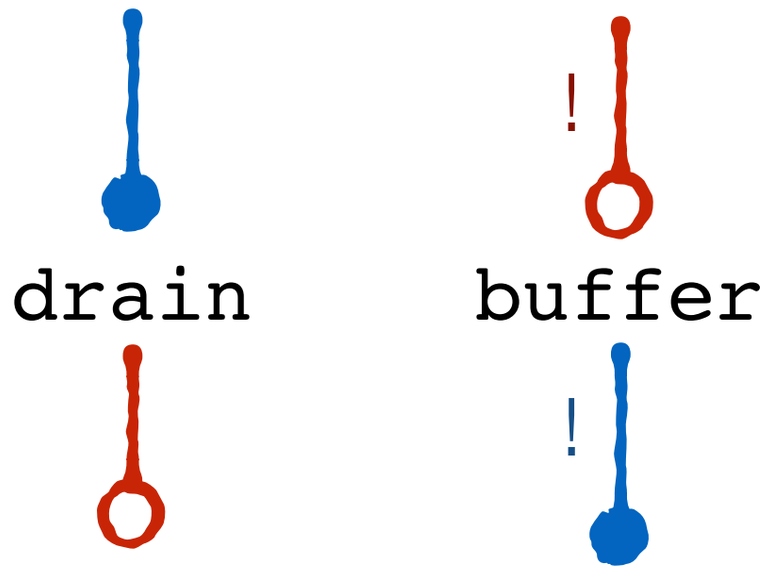


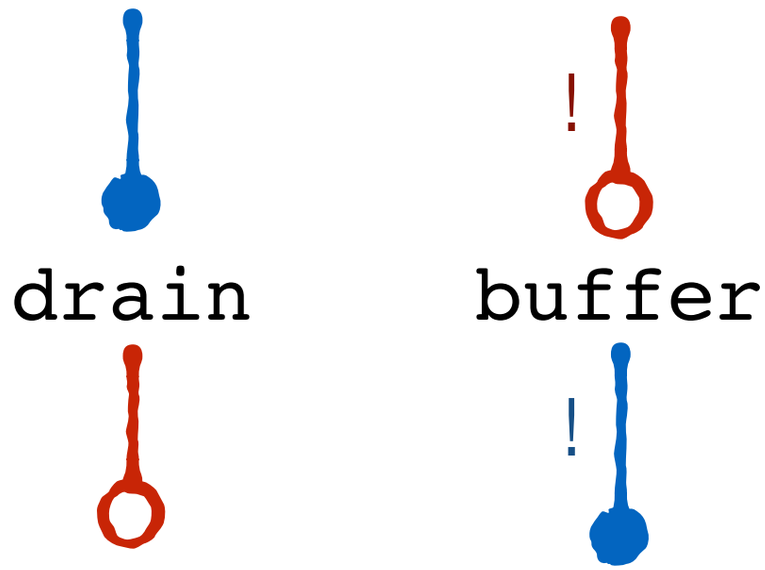
`repa-flow` package (on Hackage)

Used in production at Vertigo,
financial data processing ~ few GBs.

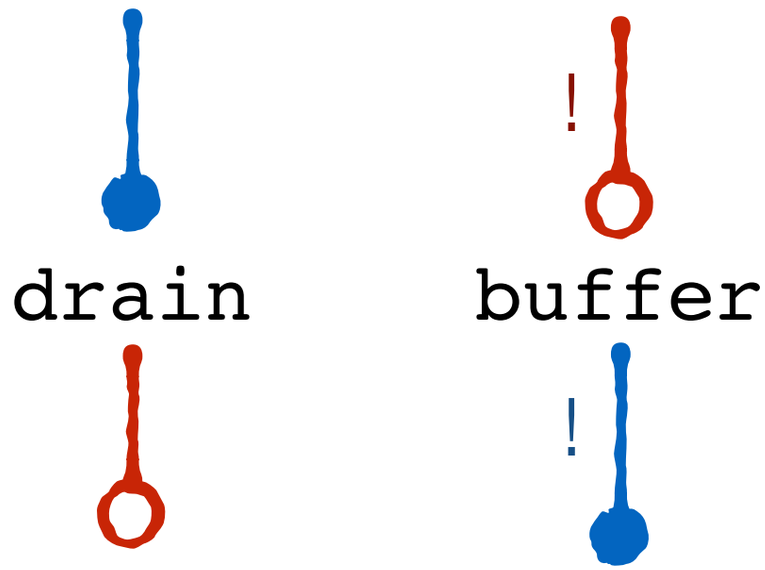
Implementation uses chunked streams.
Absolute performance primarily depends on the library used to process the chunks.

Questions?


drain
!
buffer
!



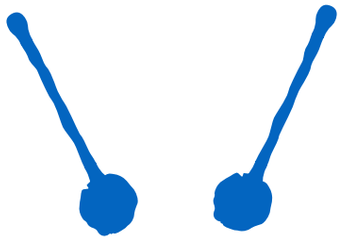

# “operator is in control”




**“operator is in control”**

**“context is in control”**

(a * b)

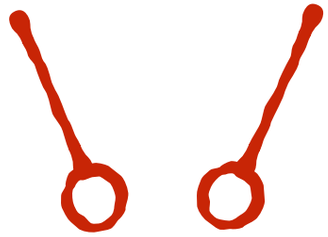


zipDrain

drain

buffer

(a + b)



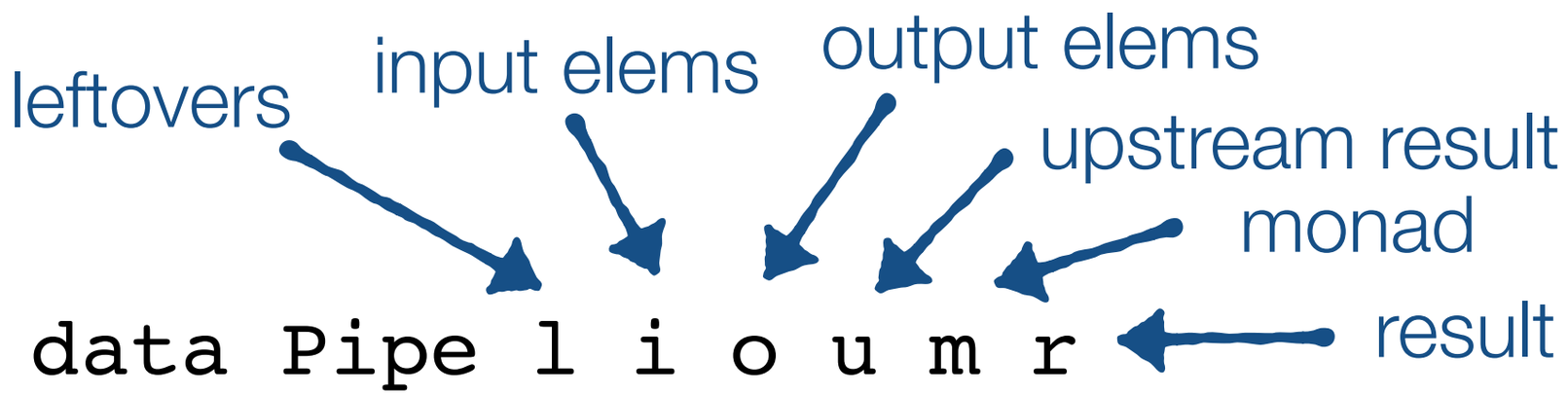
altBuffer

**“operator is in control”**

**“context is in control”**

# Comparison

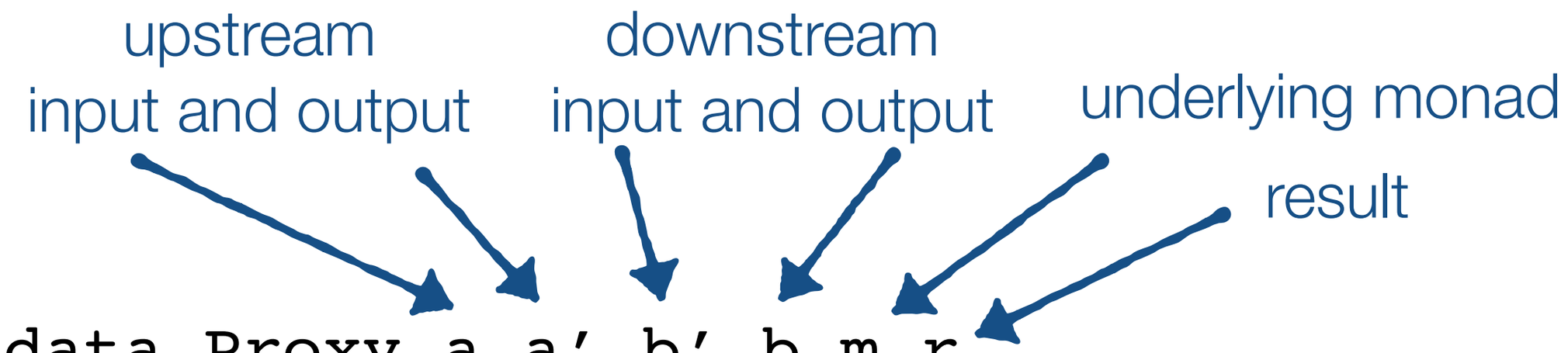
# **conduit** – *Michael Snoyman*




```
data Pipe l i o u m r
  = HaveOutput (Pipe l i o u m r) (m ()) o
  | NeedInput  (i -> Pipe l i o u m r)
               (u -> Pipe l i o u m r)
  | Done r
  | PipeM (m (Pipe l i o u m r))
  | Leftover (Pipe l i o u m r))
```

- Pipe is an instance of Monad.
- Data can flow both ways through the pipe, and yield a final result.
- Single stream, single element at a time.
- Individual Sources created by ‘yield’ action.
- Combine pipes/conduits with fusion operators.

# **pipes** – *Gabriel Gonzelez*




```
data Proxy a a' b' b m r

  = Request a' (a  -> Proxy a' a b' b m r)
  | Respond b  (b' -> Proxy a' a b' b m r)
  | M          (m    (Proxy a' a b' b m r))
  | Pure r
```

- Proxy / Pipe is an instance of Monad.
- Data can flow both ways through the pipe, and yield a final result.

# **machines** – *Edward Kmett*

```
newtype MachineT m k o
  = MachineT
  { runMachine :: m (Step k o (MachineT m k o))

type Machine k o
  = forall m. Monad m => MachineT m k o

type Process a b = Machine (Is a) b)

type Source b    = forall k. Machine k b
```

- Like streams as used in Data.Vector stream fusion,
  except the step function returns a whole new Machine (stream)

- Clean and general API, but not sure if it supports array fusion.
  Machines library does not seem to attempt fusion.

# `repa-flow vs others`

- Repa flow provides chunked, data parallel database-like operators with a straightforward API.

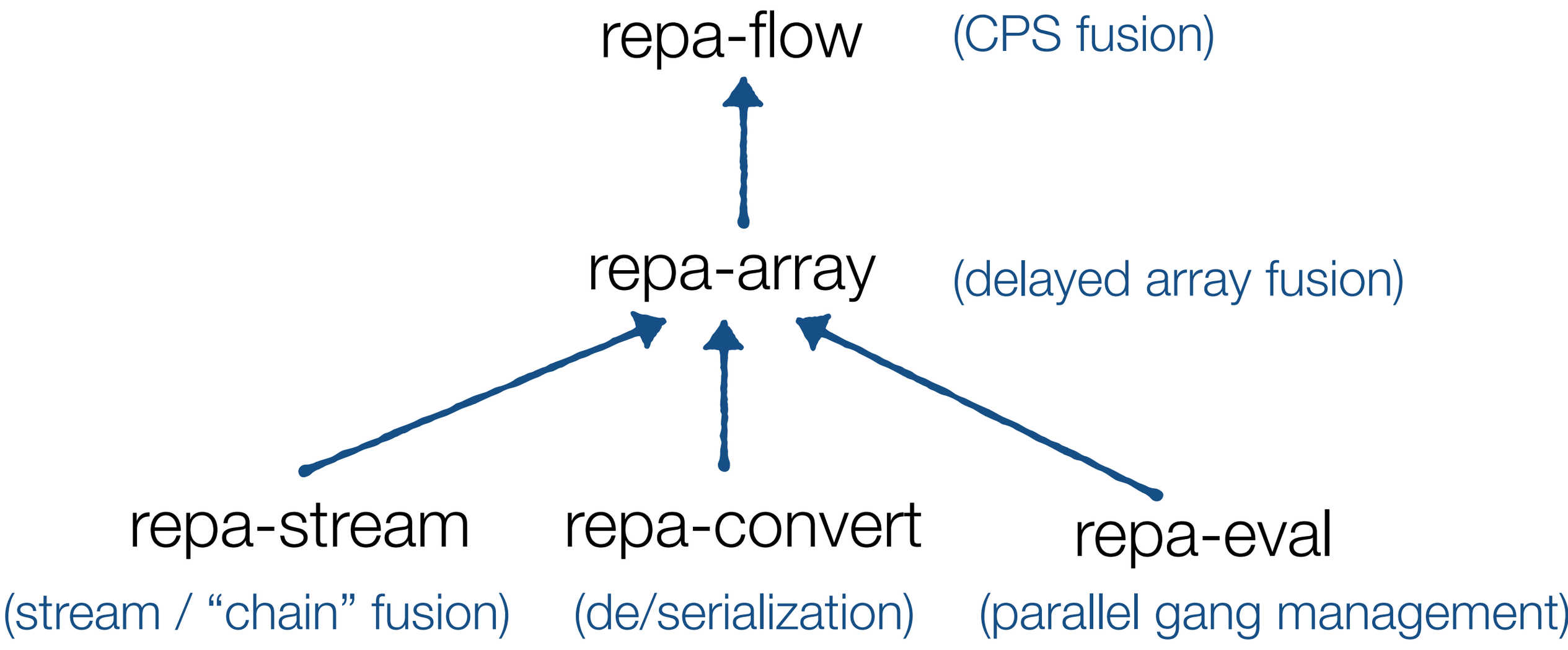
- Sources and Sinks are values rather than computations. The “Pipe” between them created implicitly in IO land.

- API focuses on simplicity and performance via stream and array fusion, rather than having the most general API.

- Suspect we could wrap single-stream Repa flow operators as either Pipes or Conduits, but neither of the former seem to naturally support data parallel flows.

# Implementation

Image: gullevek. flickr. CC-NC-SA.


repa-flow
(CPS fusion)
repa-array
(delayed array fusion)
repa-stream
(stream / “chain” fusion)
repa-convert
(de/serialization)
repa-eval
(parallel gang management)