1. Dataflow Networks

Streaming models

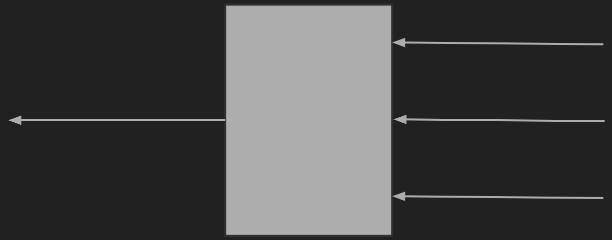
Working with infinitely streaming data

- multiple input streams
- single output stream

Process and create output stream as input comes in

- Ideally don't buffer

What goes in the box?



Dataflow networks

Dataflow networks take streams of values as inputs and produce streams of values as outputs

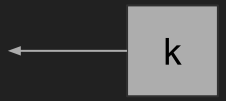
- type of values depends on the kind of network developed
    - floating point for approximation algorithms
    - images for streaming movies

- sequential components connected by buffered communication channels

- model assumes an underlying sequential language

Primitive components

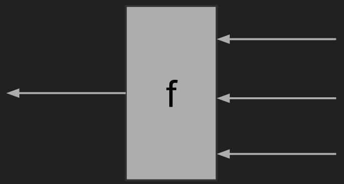
Constant k: produces an infinite stream of k



Primitive components

map f:	transforms one or more streams by applying f to the inputs

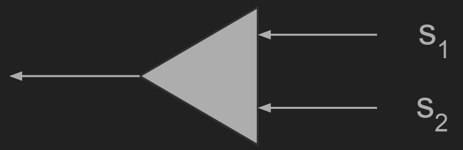
- blocks until all input streams have at least one value)
- transformation f written in underlying sequential language
- transformation f holds no state



Primitive components

followed by : produces a stream from the first element of s_1 followed by everything from s_2

- blocks until an element of s_1 arrives
- then simply forwards values that arrive on s_2

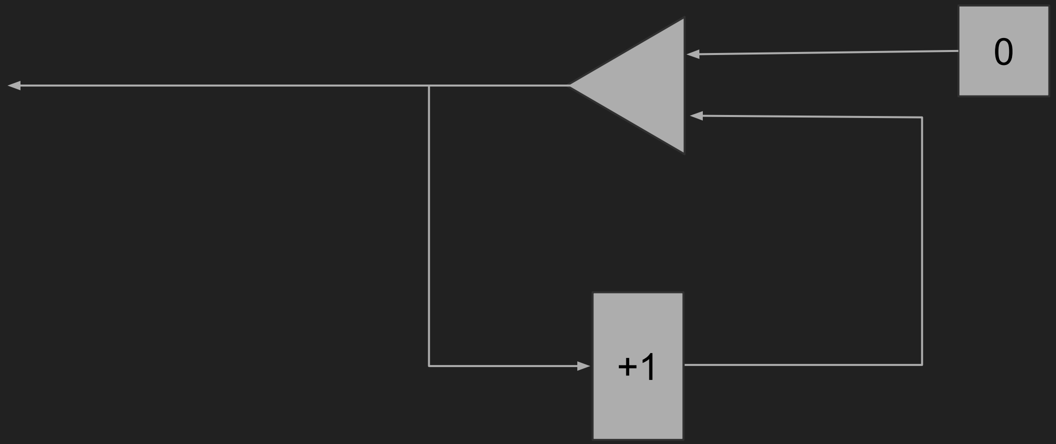


Primitive components

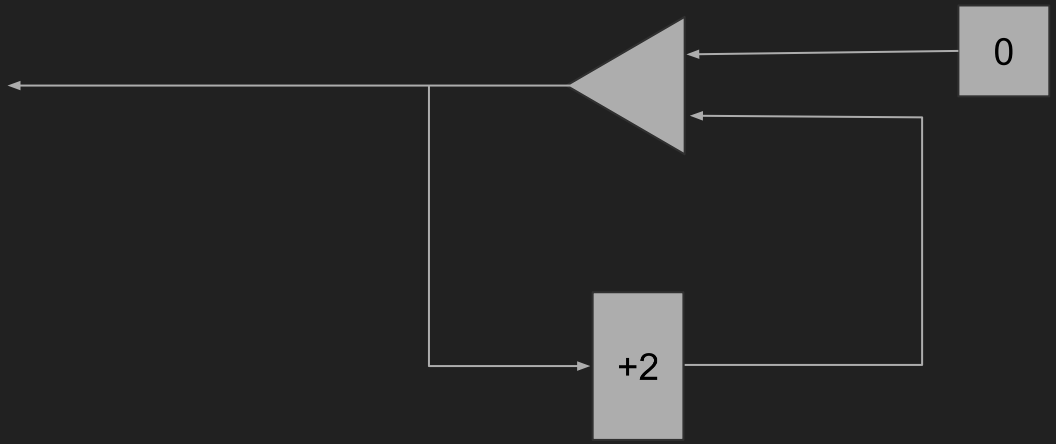
drop : produces a stream from the input stream by "dropping" the first element of the stream

- input *a b c d e f …* output *b c d e f …*
- discards the first element that arrives (produce no output)
- then simply forwards everything that arrives to its output

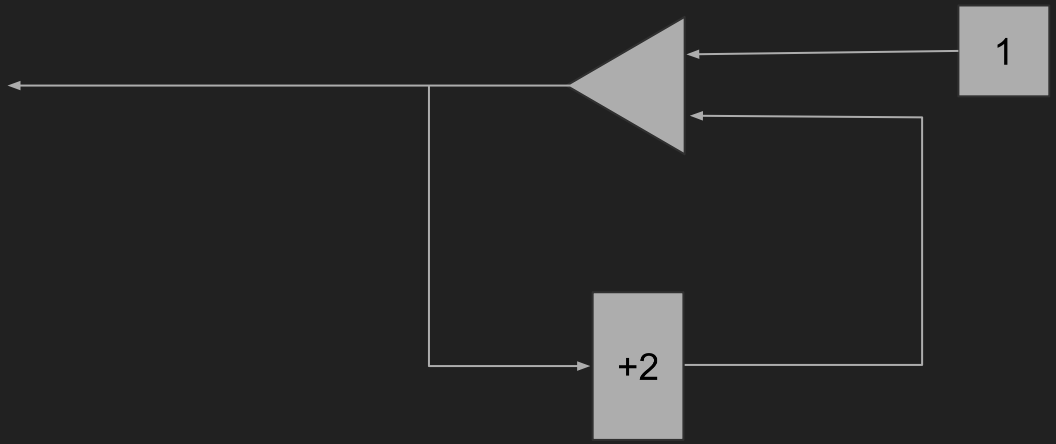
Sequences: nats



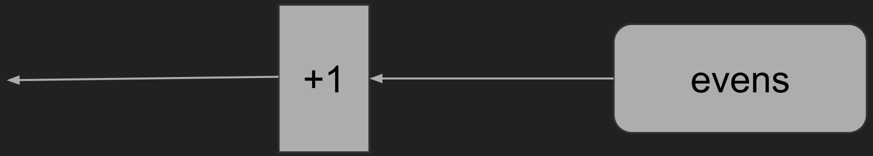
Sequences: evens



Sequences: odds



Sequences: odds

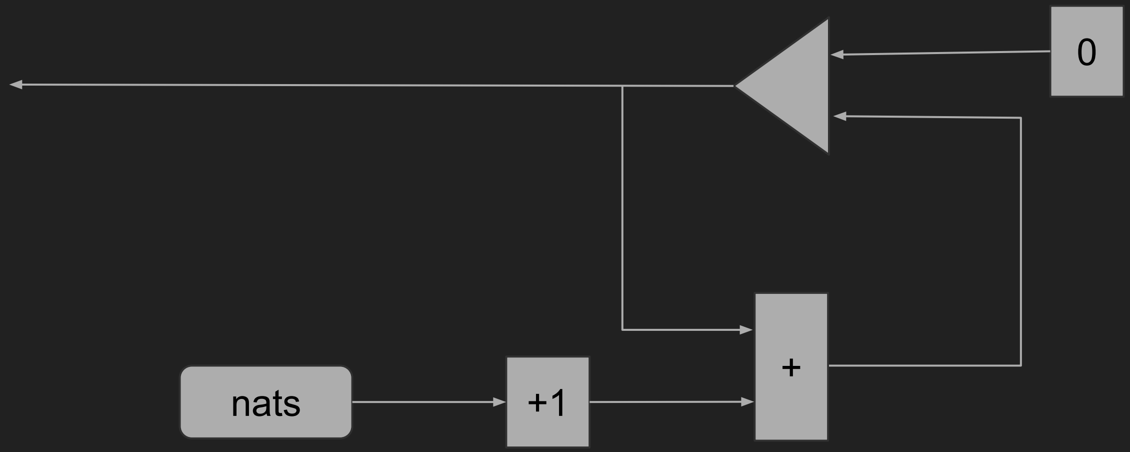


Sequences: triangular numbers

Want to create 0, 1, 3, 6, 10, 15, 21, …

Observe:

0 + 1	= 1
1 + 2	= 3
3 + 3	= 6
6 + 4	= 10
10 + 5	= 15
15 + 6	= 21
...	

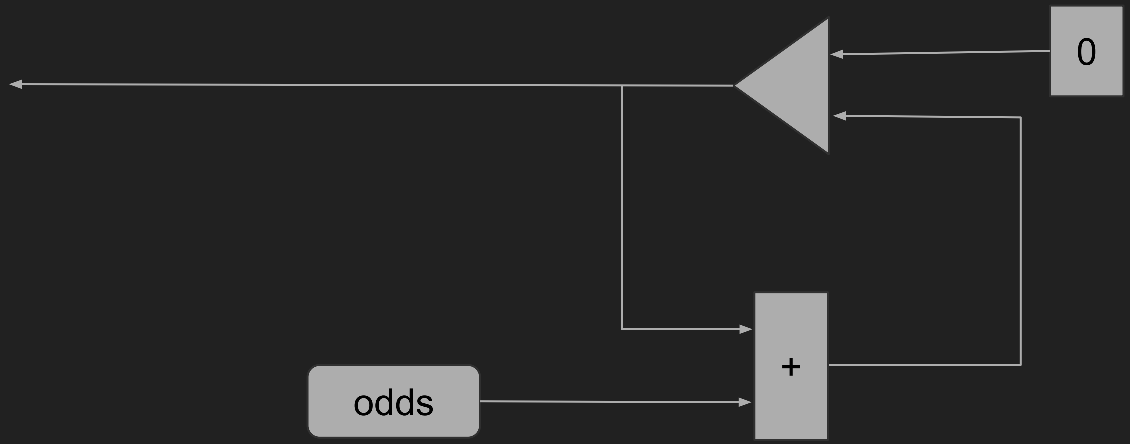


Sequences: square numbers

Want to create 0, 1, 4, 9, 16, 25, 36, …

Observe:

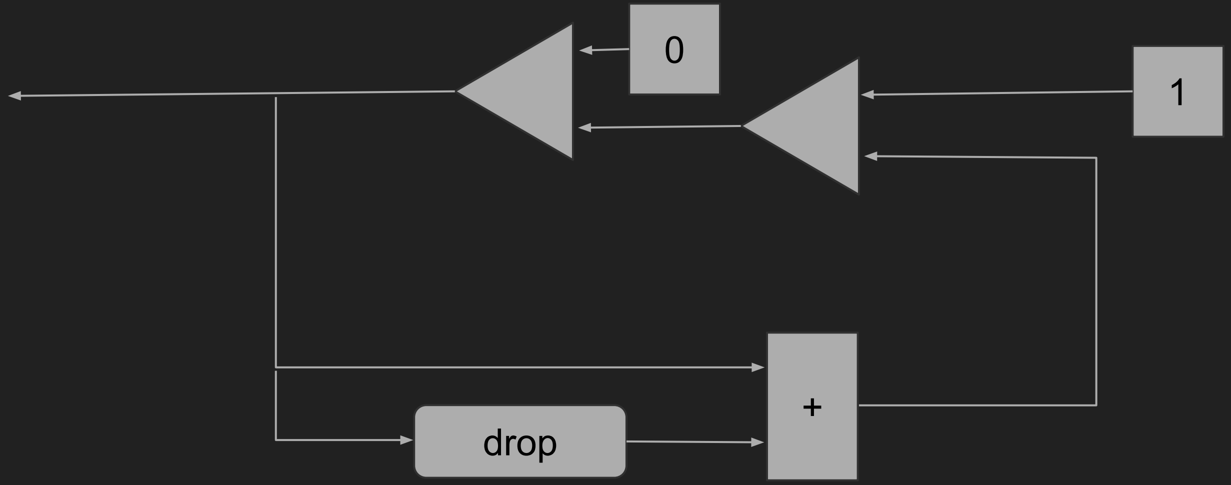
0 + 1 = 1
1 + 3 = 4
4 + 5 = 9
9 + 7 = 16
16 + 9 = 25
25 + 11 = 36
...



Sequences: Fibonacci numbers

Want to create 0, 1, 1, 2, 3, 5, 8, 13, 21, …

Each number in the sequence is the sum of the previous two

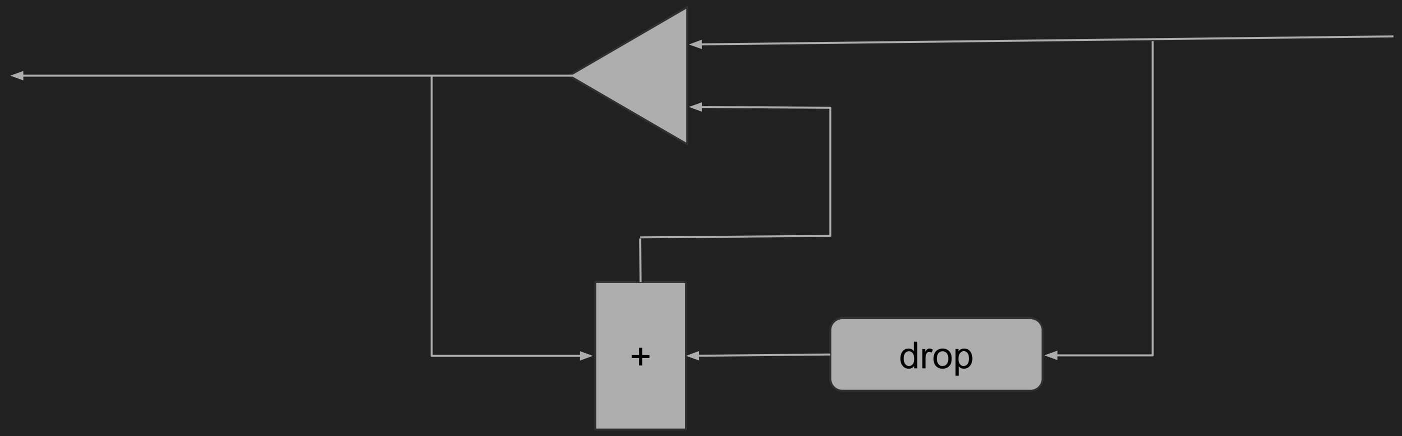


Transformation: partial sums

Input:	a	b	c	d	e	f	...
Output:	a	a+b	a+b+c	a+b+c+d	a+b+c+d+e	a+b+c+d+e+f	…

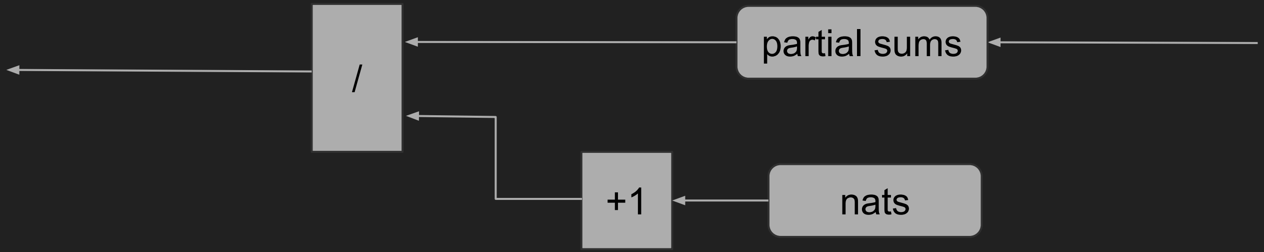
Transformation: partial sums

Input:	a	b	c	d	e	f	...
Output:	a	a+b	a+b+c	a+b+c+d	a+b+c+d+e	a+b+c+d+e+f	…



Transformations: running averages

Input:	a	b	c	d	e	...
Output:	a/1	(a+b)/2	(a+b+c)/3	(a+b+c+d)/4	(a+b+c+d+e)/5	...



Definitions

A dataflow network with inputs I and outputs O is a finite network of components where:

1. ever component is either a primitive component or an already defined dataflow network

2. every component's input is either in I or connected to exactly one output

3. every component's output can be connected to zero or more inputs and can also appear in O

Main theorem

A cycle in a dataflow network is a path from the output of some component back to an input of the same component by following links in the network

Theorem: *If every cycle in a dataflow network goes through the lower input of at least one "followed by" primitive component, then the dataflow network computes a function from its input streams to its output stream*

2. Stream Programming

From dataflow networks to streams programming

Dataflow networks are fundamentally a graphical model

We'll see how to program graphical models next homework

Another way to program over streaming data is to consider a stream to be an infinite list and write list processing functions as usual

Infinite lists

Most languages do not allowing you to define infinite lists:

- You'd spend all your time building it

Yet, Haskell (and some other languages) lets you define infinite lists:

```
from k = k : (from (k + 1))
```

where `from 10` would be the list [10, 11, 12, 13, 14, 15, 16, …]

It does so using *lazy evaluation*

Lazy evaluation

Haskell uses lazy evaluation everywhere (but relevant mostly at function calls)

> It only evaluates an expression if it *need* its value

Consider the function

```
test a b = a
```

The body does not use the second argument at all

In Haskell, `test 10 (loop 0)` returns `10` *even if* `(loop 0)` *is an infinite loop*

Lists in Haskell

If L is a list, `a : L` is a list with first element a and rest of the list L

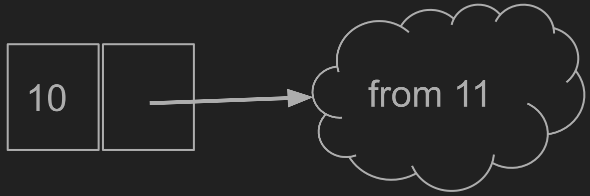
```
10 : [1, 2, 3]   →  [10, 1, 2, 3]
```

But `a : L` is lazy in its second argument

- it does not evaluate L until L is needed
  (e.g., somebody trying to access the elements of L)

With `from k = k : (from (k + 1))`

`from 10` is perfectly well defined



Lists in Haskell

If L is a list, `a : L` is a list with first element a and rest of the list L

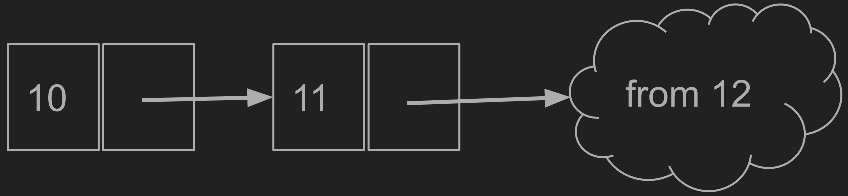
```
10 : [1, 2, 3]   →  [10, 1, 2, 3]
```

But `a : L` is lazy in its second argument

- it does not evaluate L until L is needed
  (e.g., somebody trying to access the elements of L)

With `from k = k : (from (k + 1))`

`from 10` is perfectly well defined



Primitive components

cst a = a : (cst a)

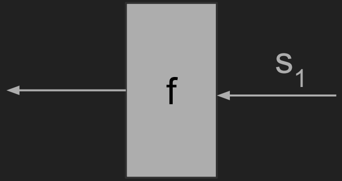
map f (h : t) = (f h) : (map f t)

map2 f (h_1 : t_1) (h_2 : t_2) = (f h_1 h_2) : (map2 f t_1 t_2)

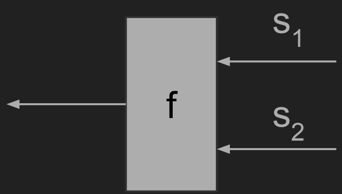
fby (h : t) s = h : s

tail (h : t) = t

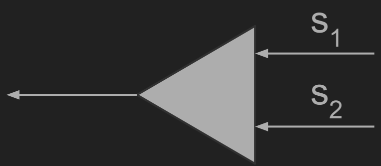
cst k



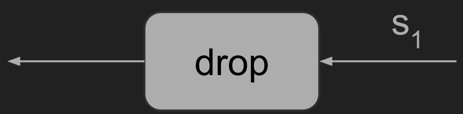
map f s_1



map2 f s_1 s_2

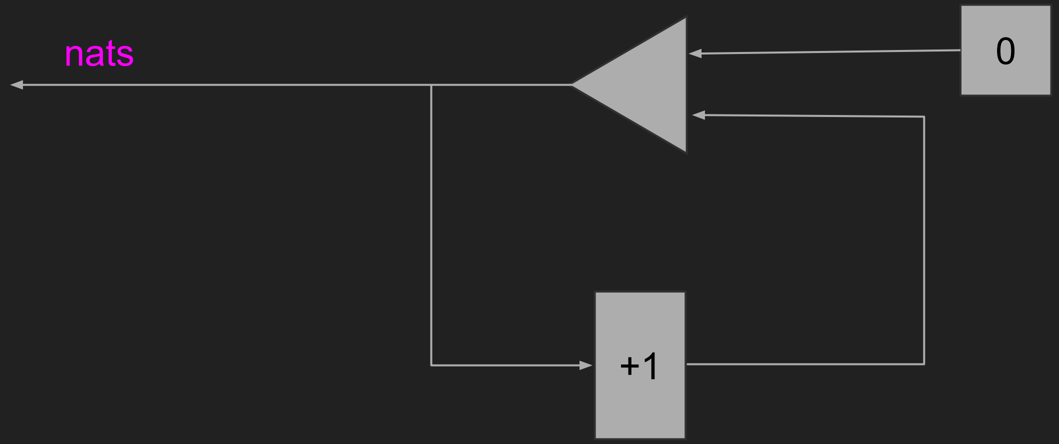


fby s_1 s_2



tail s_1

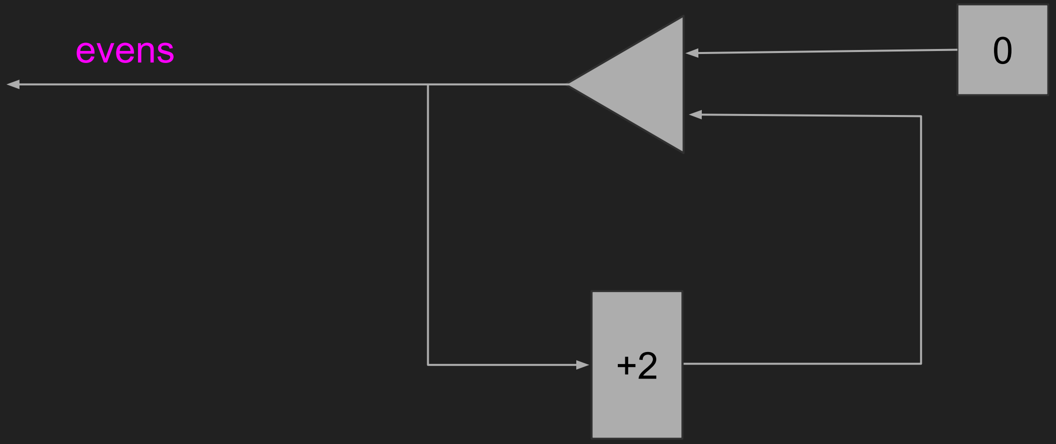
Example: nats



plus1 s = map (\a -> a + 1) s

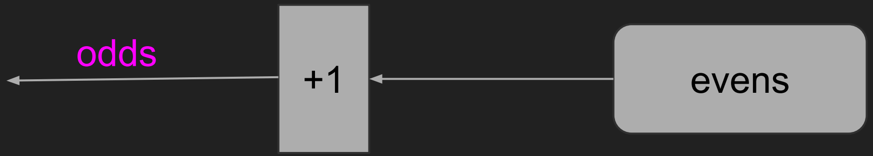
nats = fby (cst 0) (plus1 nats)

Example: evens



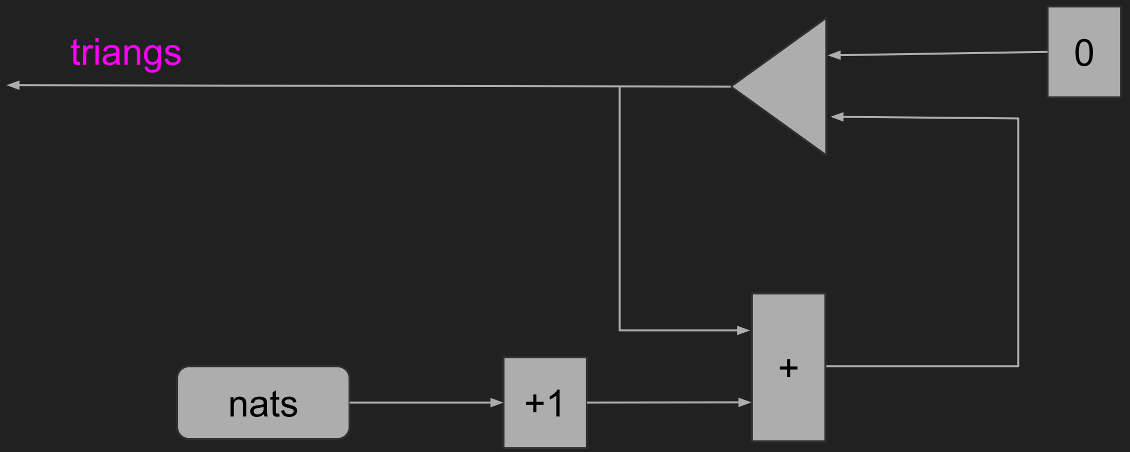
evens = fby (cst 0) (plus1 (plus 1 evens))

Example: odds



odds = plus1 evens

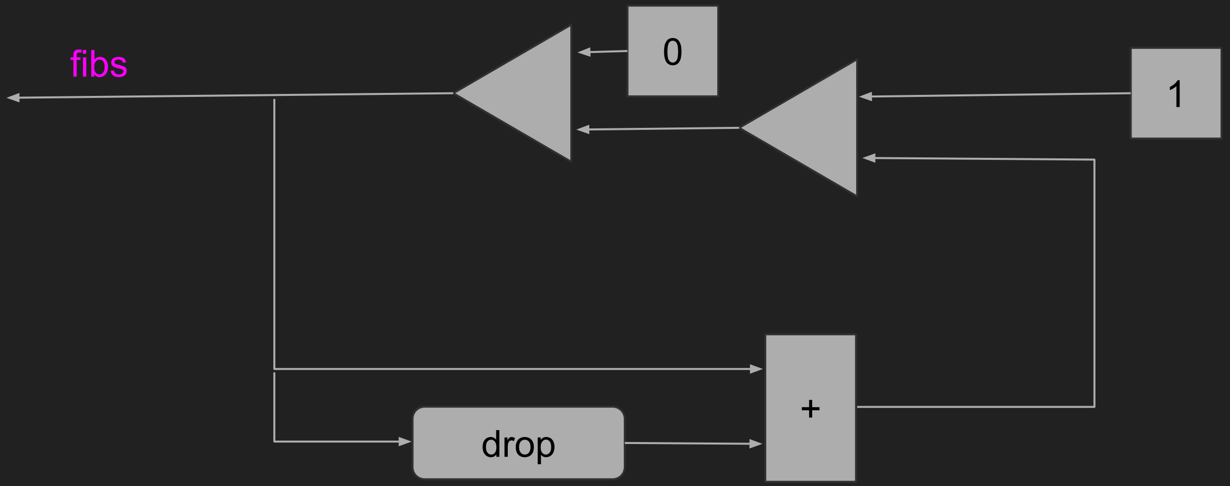
Example: triangular numbers



plus s_1 s_2 = map2 (\a -> \b -> a + b) s_1 s_2

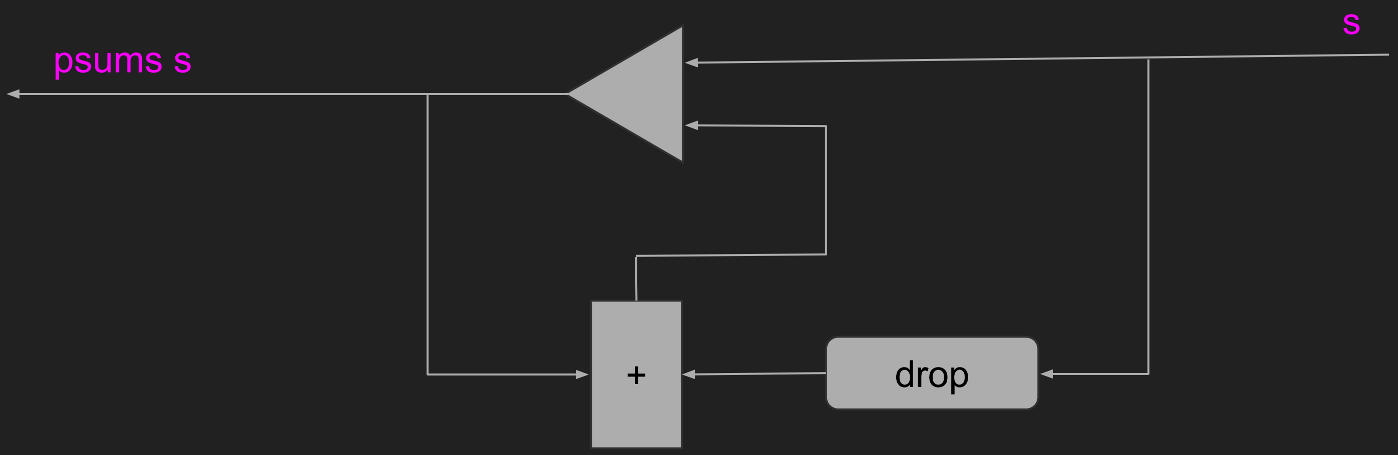
triangs = fby (cst 0) (plus triangs (plus1 nats))

Example: Fibonacci numbers



fibs = fby (cst 0) (fby (cst 1) (plus fibs (tail fibs)))

Example: partial sums



psums s = fby s (plus (psums s) (tail s))

Recursive stream programs

Sieve of Eratosthenes - how to compute the stream of prime numbers:

- sieving a stream: take a stream of values, keep the first value, and sieve the rest of the stream *after* removing all multiples of the first value
- sieving the naturals numbers starting from 2 yields the prime numbers

```
divides c x = (mod x c == 0)
sieve s = fby s (sieve (filter (\a -> not (divides (head s) a)) (tail s)))

primes = sieve (plus1 (plus1 nats))
```