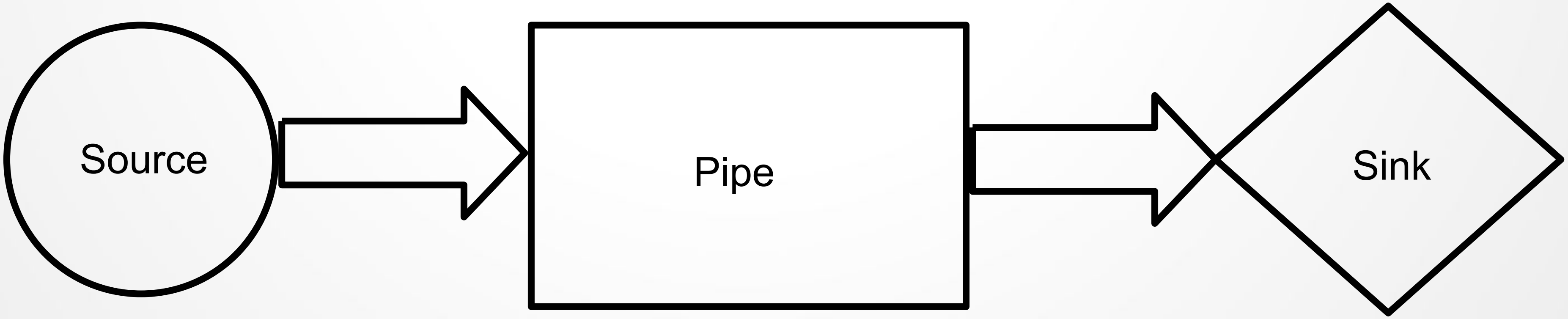
Dataflow

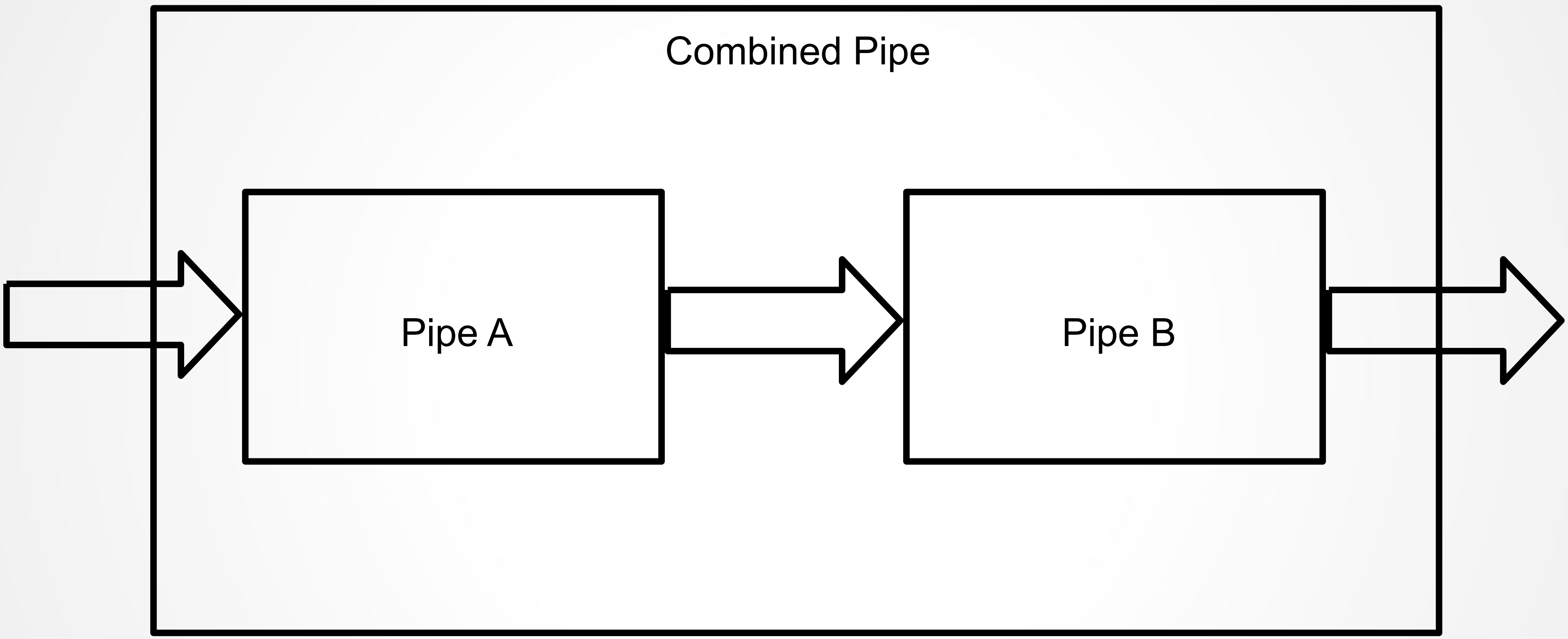
What is dataflow?

- Dataflow is a software scheme for data processing
- It is designed to resemble a pipeline of data transformations
  - By reading the pipeline one should be able to roughly understand what operations are made and in which order
- Three main components:
  - Sources: feed data into the pipeline (e.g. read data)
  - Pipes: transform or filter data
  - Sinks: terminate pipelines (e.g. write data)
- These components are combined to form a closed pipeline



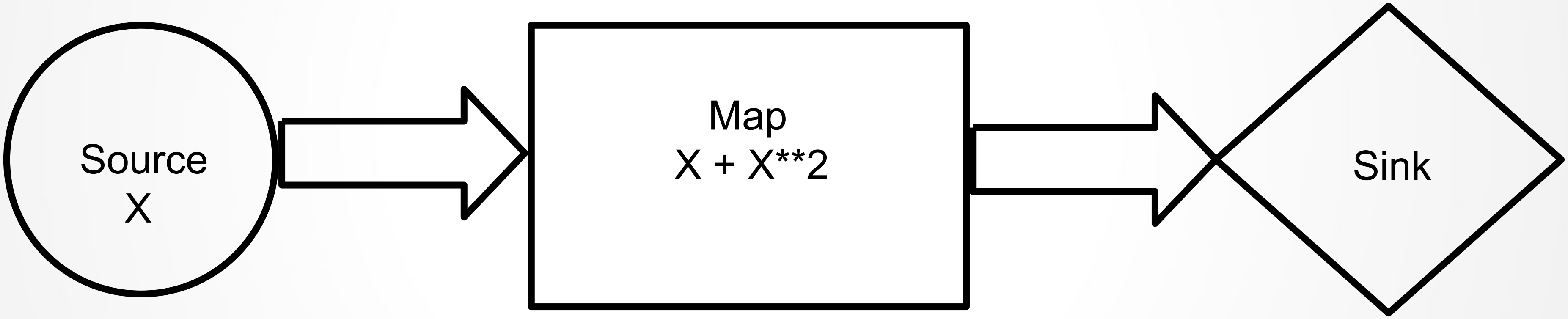
Combining pipes

- *Pipes* can be combined to form bigger ones



Map

- *Maps* are the most common way to create a pipe component
- They are just a function that takes some data and produces some new data

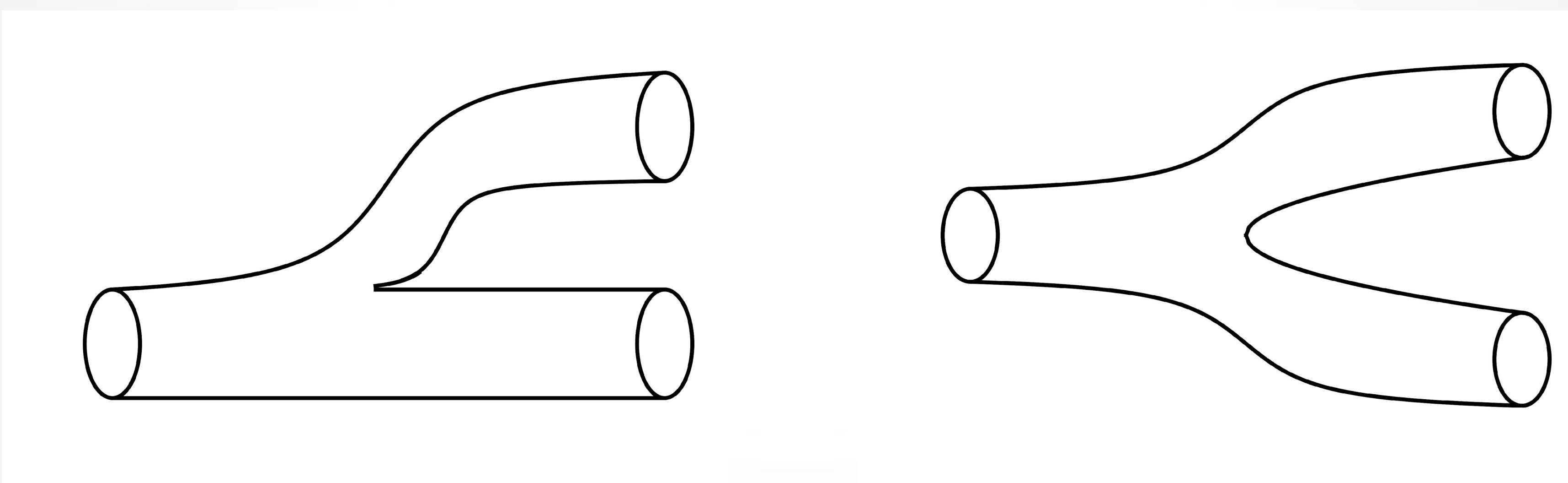


Splitting pipes

- Pipes can be split using *forks* and *branches*
- Forks and branches are topologically equivalent, the difference is purely a matter of interpretation
  - Branches: thought of as a secondary workflow with lower priority*
  - Forks: Each arm is thought of as having the same priority*

Branch

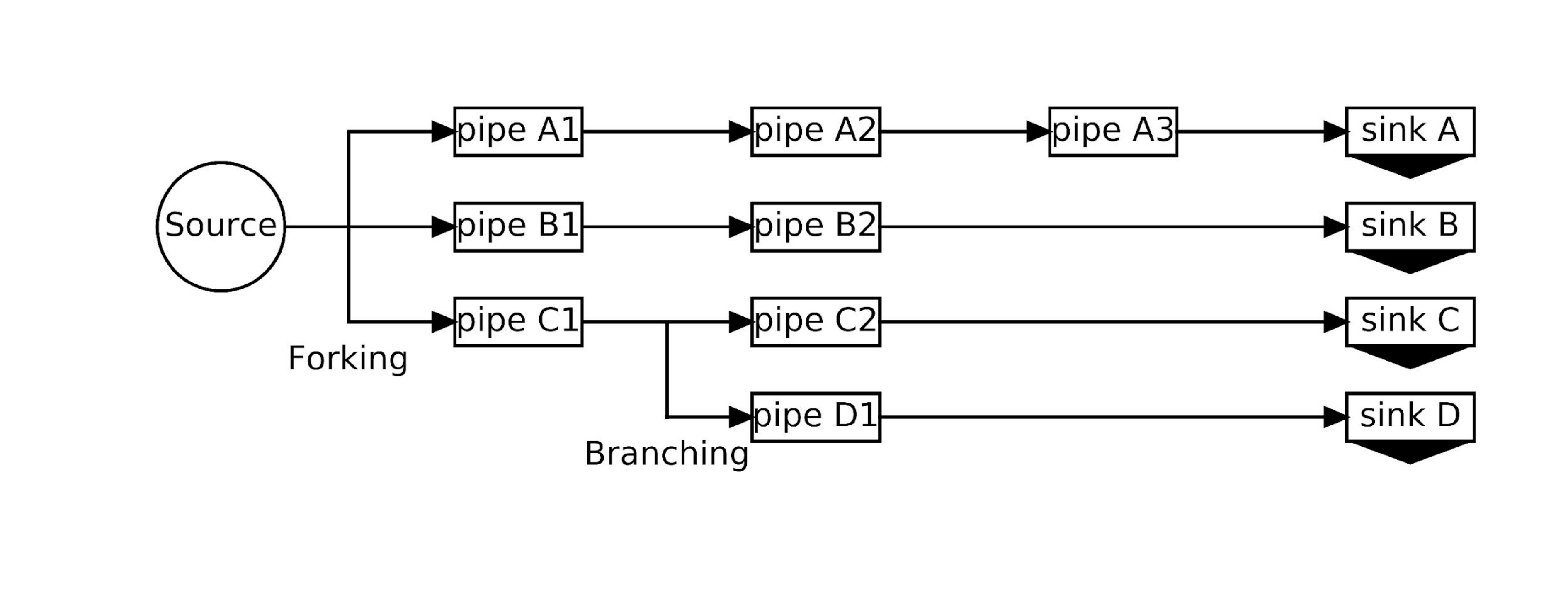
Fork



* There is no actual priority ordering

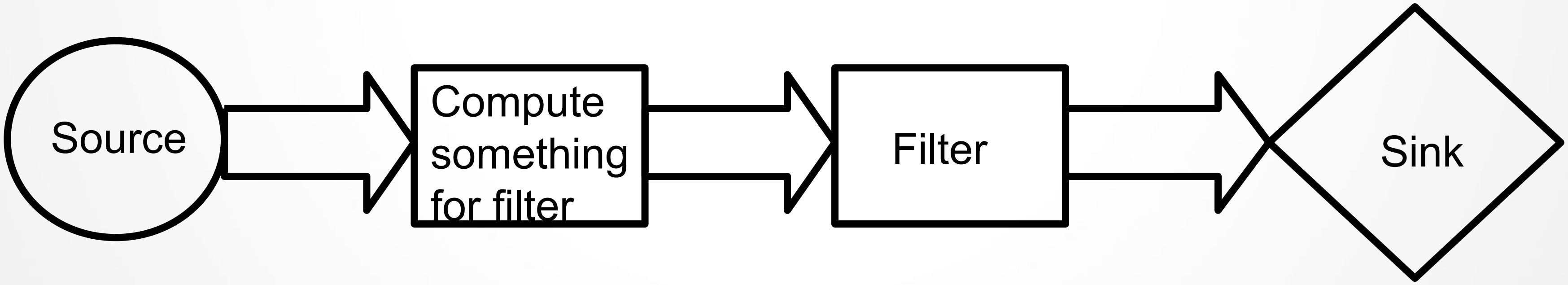
Complex structures

- With this scheme we can create complex structures
- Every line must be terminated by a sink!



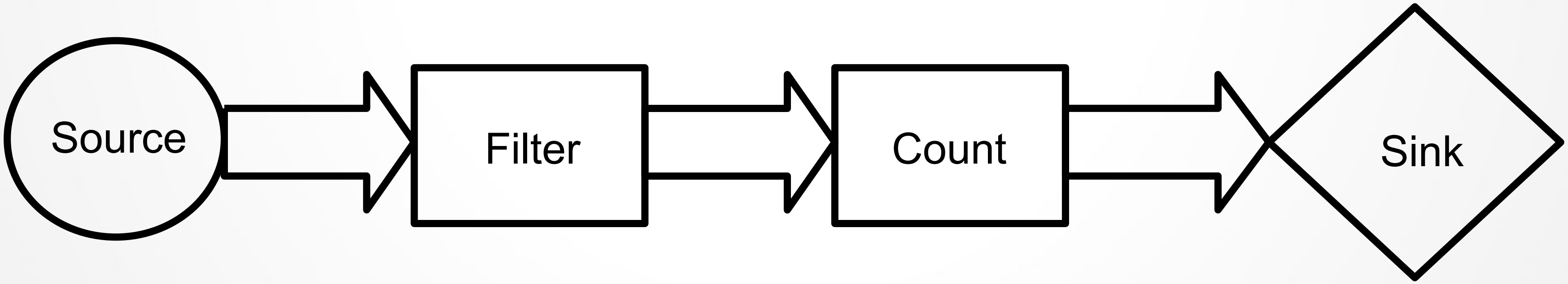
Filters

- Filters can stop the workflow (for a specific iteration) when some condition is satisfied
- They can perform any complex operation on the input data to decide, but:
  - Usually the calculations are useful for something else
  - The outcome is not stored
- Thus, the preferred strategy is something like



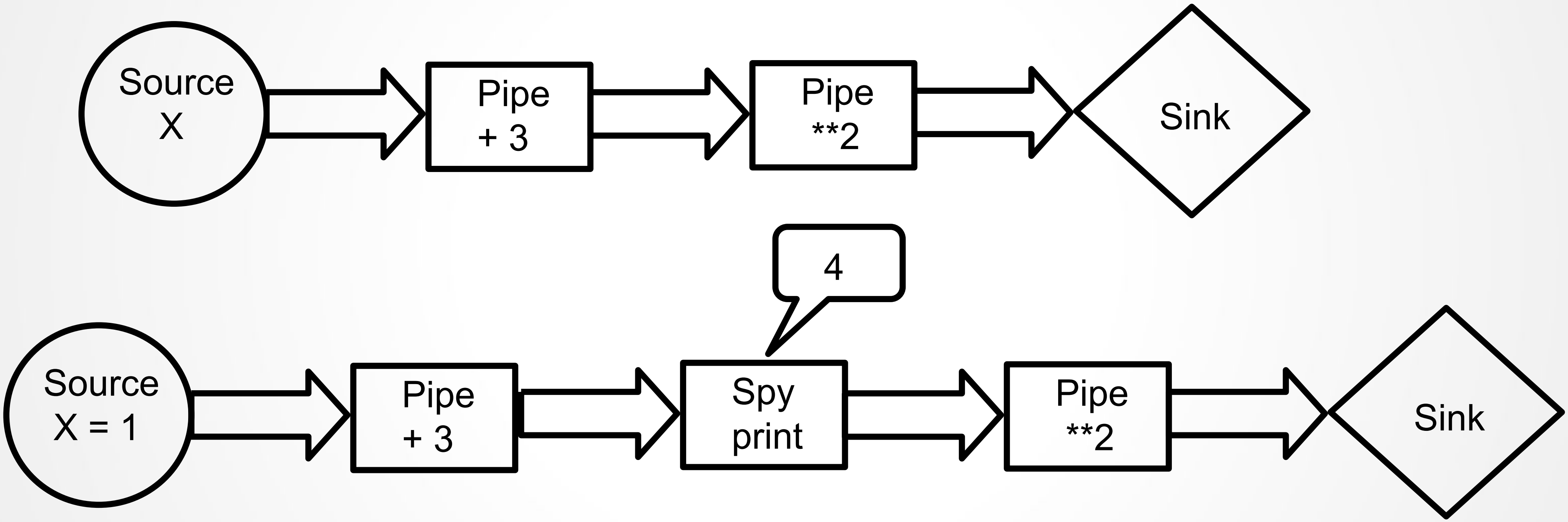
Futures

- Futures are variables that will only hold a value at the end of execution
- They are useful to collect data during the execution of the program
- The most obvious example is counting how many times data has gone through a specific pipe of the workflow
- In this example *count* is a component that uses a future to store a value and adds +1 every time data goes through it. At the end of the process, the value used by *count* is accessible



Other components

- Spies: minimalistic branches used for “spying” the value of some variable



*** Spy is not supposed to modify the input data, but there is no protection against it. If the input data is mutable, the operation may change it (be a good person and don’t do it).

Other components

- Slice: select only a range of iterations. It works exactly as a python slice
  - slice(X) = 0, 1, ..., X-1
  - slice(X, Y) = X, X + 1, ... Y – 1
  - slice(X, Y, S) = X, X + S, ... X + nS : X + nS < Y

A final remark

The data is not sent from one component to another directly. Instead, they share a common namespace that each component can read and write to. The namespace is cleared on each iteration.

Let’s check some actual code

How to make a source

- The key aspect of this machinery is that everything is made with generators
  - Forget about *return*, it's the time to *yield*
- Sources provide data from anywhere (disk, database, a website)...
- Let's do a programme that takes the numbers in Lost:

```
def lost_numbers():
    numbers = [4, 8, 15, 16, 23, 42]
    for number in numbers:
        yield dict(number = number)
```

- This generator creates a dictionary that will work as a common namespace during one iteration. It contains (for now) the number we are going to deal with

How to make a map

- *Maps* transform data
- We need to specify the labels of the input and output data
- Let’s do some operations to our numbers: add 42 and take the sqrt
- First we define our operations:

```
def number_adder(y):
    def adder(x):
        return x + y
    return adder
```

```
from math import sqrt
```

- Then, we make them pipes!

```
add_42    = fl.map(number_adder(42), args="number"   , out="number+42")
take_sqrt = fl.map(          sqrt, args="number+42", out="final result")
```

- (Optional) We can combine them into a bigger pipe

```
do_everything = fl.pipe(add_42, take_sqrt)
```

How to make a sink

- *Sinks* terminate pipelines, usually by storing data
- We only need to specify the label of the input data
- Let's create two different sinks: one that prints the number and another one that writes it to a text file.
- First we define our operations:

```
def file_writer(file):
    def write(data):
        file.write(f"{data}\n")
    return write
```

No need to define print

- Then, we make them sinks!

```
with open("demonstration_0.txt", "w") as file:
    print_  = fl.sink(            print, args="final result")
    write   = fl.sink(file_writer(file), args="final result")
```

How to put everything together

- To put everything together we only need to push data through the pipe!

```
fl.push(source = lost_numbers(),
        pipe   = fl.pipe(add_42,
                         take_sqrt,
                         fl.fork(print_,
                                 write)),
        result = ())
```

- If we save this in a file demonstration_0.py, then we just need to run
- python demonstration_0.py
- Check that the numbers are printed on screen and that the test file has been written

An example with futures

- Futures are not used in the main program, but as part of a component
- The easiest way to use a future is with *RESULT*:
  - *RESULT* is a decorator that takes a generator and creates a new future for it
  - The decorated function will give you an object with two attributes: future and sink
  - The future attribute will allow you to access the final value

```
def RESULT(generator_function):
    def proxy(*args, **kwds):
        future = Future()
        coroutine = generator_function(future, *args, **kwds)
        next(coroutine)
        return FutureSink(future, coroutine)
    return proxy
```

- A generic generator that uses a future will be something like

```
@fl.RESULT
def my_generator_with_future(future):
    try:
        while True:
            input_data = yield # get data for this iteration
            # do something
    finally:
        future.set_result(something_calculated_along_the_way)
```

An example with futures

- Let's count how many iterations we go through

```
@fl.RESULT
def counter(future):
    count = 0
    try:
        while True:
            yield
            count += 1
    finally:
        future.set_result(count)
```

- To use this object we need to define a sink

```
count = counter()
# and count.sink would be used as a pipe component
```

- But now we cannot plug it in the middle of the pipeline, it's a sink!
- That's what branches are for!

An example with futures

- But how do we access the result?
- In the *result* argument of *push*!
- To access it, we need to assign the result of *push* to a variable
- Putting everything together:

```
result = fl.push(source = lost_numbers(),
                 pipe   = fl.pipe(fl.branch(count.sink),
                                  add_42,
                                  take_sqrt,
                                  fl.fork(print_,
                                          write )),
                 result = dict(n_numbers = count.future))

print(f"We have looped over {result.n_numbers} numbers")
```

- If we save this in a file demonstration_1.py, then we just need to run
- python demonstration_1.py
- Check that the numbers are printed along with the new sentence and that they are written to the file

An example with filters

- Filters work on a function that return a boolean value
- Let’s filter out odd values from our code
- First, we define our operation

```
def is_even(x):
    return bool(x % 2 == 0)
```

- And then, the filter. In this case we are going to take the initial number

```
keep_even = fl.filter(is_even, args="number")
```

- Putting everything together

```
result = fl.push(source = lost_numbers(),
                 pipe   = fl.pipe(keep_even,
                                  fl.branch(count.sink),
                                  add_42,
                                  take_sqrt,
                                  fl.fork(print_,
                                          write )),
                 result = dict(n_numbers = count.future))
print(f"We have looped over {result.n_numbers} numbers")
```

Let’s build a “city”

Build your own city

Build a city that has:
- Source: 1e4 random numbers following a Poisson(50)
- Pipes:
    - Count events in
    - Slice to keep only every 3rd number starting from the 5th one
    - Count events after slicing
    - Perform x + x**2 + x**3
    - If the result is divisible by 7 save the original number to divisible_by_7.txt
    - If the sum of the digits is divisible by 3 save the modified number to divisible_by_3.txt
    - If neither condition is satisfied save the original number to useless.txt
    - Add counters to check how many numbers satisfy each condition

EXTRA:
- Write a test :)