**Exercise 2-1**

**RunMultiplier.groovy**

**def** processList = [ **new** Producer ( outChannel: connect1.out() ),

//insert here an instance of multiplier with a multiplication factor of 4

**new** Multiplier ( inChannel: connect1.in(),

outChannel: connect2.out(), factor: 4 ),

**new** Consumer ( inChannel: connect2.in() )

]

**Multiplier.groovy**

// write i \* factor to outChannel

outChannel.write (i \* factor)

// read in the next value of i

i = inChannel.read()

**Consumer.groovy**

//insert a modified println statement

println "Next integer multiplied by 4 is: ${i}"

i = inChannel.read()

**Output**

next: 2

next: Next integer multiplied by 4 is: 8

3

next: Next integer multiplied by 4 is: 12

4

next: Next integer multiplied by 4 is: 16

5

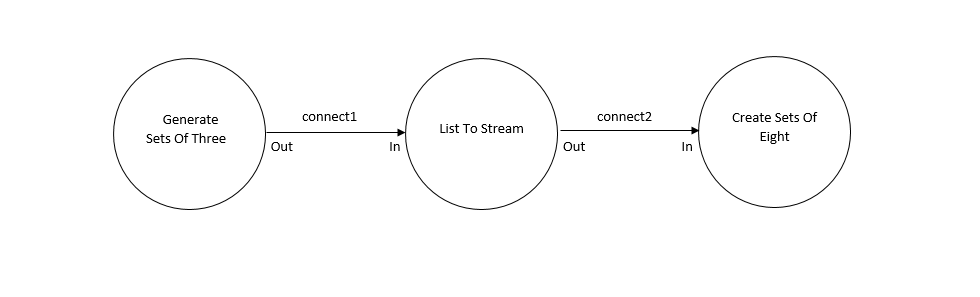
next: Next integer multiplied by 4 is: 20

2

next: Next integer multiplied by 4 is: 8

0

Finished

**Exercise 2-2**

**GenerateSetsOfThree.groovy**

//write the terminating List as per exercise definition

outChannel.write([-1,-1,-1])

**ListToStream.groovy**

// hint: output list elements as single integers

**for**(i **in** 0 ..< inList.size){

outChannel.write(inList[i])

}

inList = inChannel.read()

**CreateSetsOfEight.groovy**

// put v into outList and read next input

outList[i] = v

v = inChannel.read()

**Output**

Eight Object is [1, 2, 3, 4, 5, 6, 7, 8]

Eight Object is [9, 10, 11, 12, 13, 14, 15, 16]

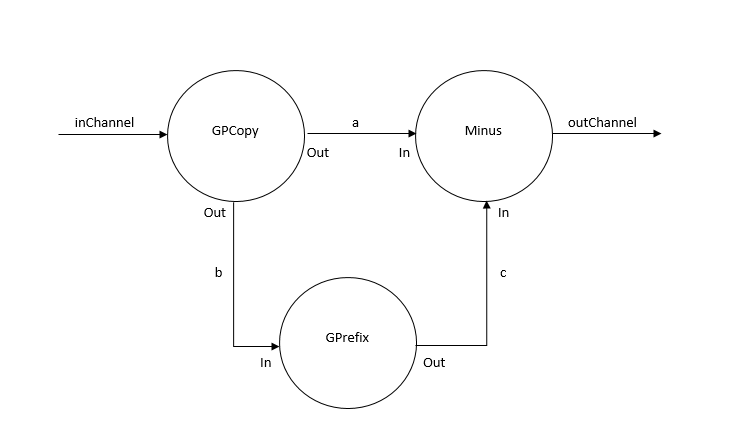
Eight Object is [17, 18, 19, 20, 21, 22, 23, 24]

Finished

**Questions**

1. To output 6 integers, change the line **for(i in 0 ..7)** to **for(i in 0 ..5)** in CreateSetsOfEight.groovy
2. Create a variable **size** that can be adjusted to any number and then pass it to the process, which will decide the size of each list generated.
3. The process will just get stuck, it will eventually read the termination number of -1 but then will have no number to read and will be stuck.

**Exercise 3-1**

**Differentiate**

**Minus.groovy**

parRead2.run()

// output one value subtracted from the other

// be certain you know which way round you are doing the subtraction!!

outChannel.write (read0.value - read1.value)

**Differentiate.groovy**

**def** differentiateList = [ **new** GPrefix ( prefixValue: 0,

inChannel: b.in(),

outChannel: c.out() ),

**new** GPCopy ( inChannel: inChannel,

outChannel0: a.out(),

outChannel1: b.out() ),

// insert a constructor for Minus

**new** Minus ( inChannel0: a.in(),

inChannel1: c.in(),

outChannel: outChannel )

]

**Output - Minus**

Differentiated Numbers

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

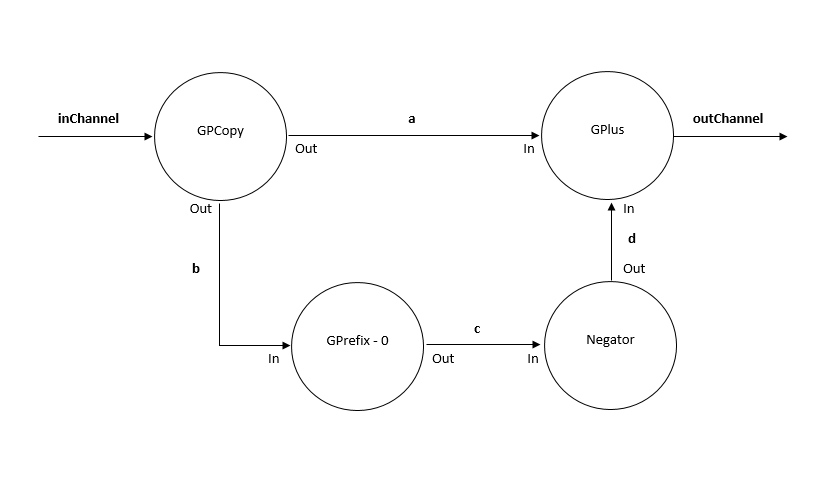
16

17

18

19

20

**DifferentiateNeg**

**Negator.groovy**

//output the negative of the input value

outChannel.write(-inChannel.read())

**DifferentiateNeg.groovy**

**def** differentiateList = [ **new** GPrefix ( prefixValue: 0,

inChannel: b.in(),

outChannel: c.out() ),

**new** GPCopy ( inChannel: inChannel,

outChannel0: a.out(),

outChannel1: b.out() ),

//insert a constructor for Negator

**new** Negator (inChannel: c.in(),

outChannel: d.out() ),

**new** GPlus ( inChannel0: a.in(),

inChannel1: d.in(),

outChannel: outChannel )

]

**Output - negator**

Differentiated Numbers - Negator

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

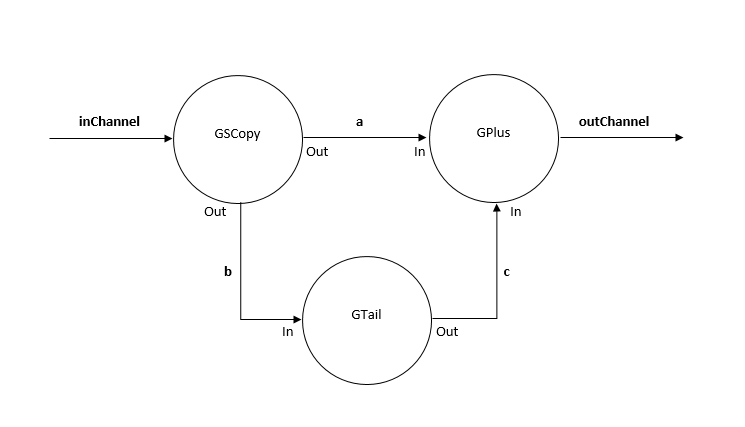
18

19

20

**Questions**

I believe the Negator approach is much better, as it lets us reuse the GPlus which we are already familiar with. Negator process is also much simpler to create and connect to other processes as you don’t need to pay attention to the way channels are connected as you would have to with the Minus process since subtraction isn’t commutative and you have to make sure the correct number is subtracted e.g. 2-1 =/= 1-2

**Exercise 3-2**

**GSCopy.groovy**

**def** i = inChannel.read()

// output the input value in sequence to each output channel

outChannel0.write(i)

outChannel1.write(i)

**GSquares.groovy Version A**

**def** testList = [ **new** GNumbers ( outChannel: N2I.out() ),

**new** GIntegrate ( inChannel: N2I.in(),

outChannel: I2P.out() ),

// you will need to modify this twice

//first modification is to insert a constructor for GSPairsA

// then run the network using TestGSCopy

**new** GSPairsA ( inChannel: I2P.in(),

outChannel: outChannel)

//second modification replace the constructor for GSPairsA with GSPairsB

// then run the network again using TestGSCopy

// you will then be able to compare the behaviour and to

// explain why this happens!

]

**Output**

Squares

**GSquares.groovy Version B**

**def** testList = [ **new** GNumbers ( outChannel: N2I.out() ),

**new** GIntegrate ( inChannel: N2I.in(),

outChannel: I2P.out() ),

// you will need to modify this twice

//first modification is to insert a constructor for GSPairsA

// then run the network using TestGSCopy

**new** GSPairsB ( inChannel: I2P.in(),

outChannel: outChannel )

//second modification replace the constructor for GSPairsA with GSPairsB

// then run the network again using TestGSCopy

// you will then be able to compare the behaviour and to

// explain why this happens!

]

**Output**

Squares

1

4

9

16

25

36

49

64

81

100

**Questions**

GSPairsA has returned no output, whereas GSPairsB has returned the output for square numbers correctly. I believe that is the case due to the order of sequential output that both processes take, GSPairsB’s outChannel0 outputs to GTail, which completes the processes and then in sequence outChannel1 can now output and it goes on until terminated. However, GSPairsA’s outChannel0 outputs to GPlus which needs 2 input values to complete its process, but since GSCopy is sequential, it will not write a value to outChannel1 since outChannel0 did not complete its process because GPlus did not complete its process and so on. That is why GSPairsB works and GSPairsA does not work.

**Exercise 3-3**

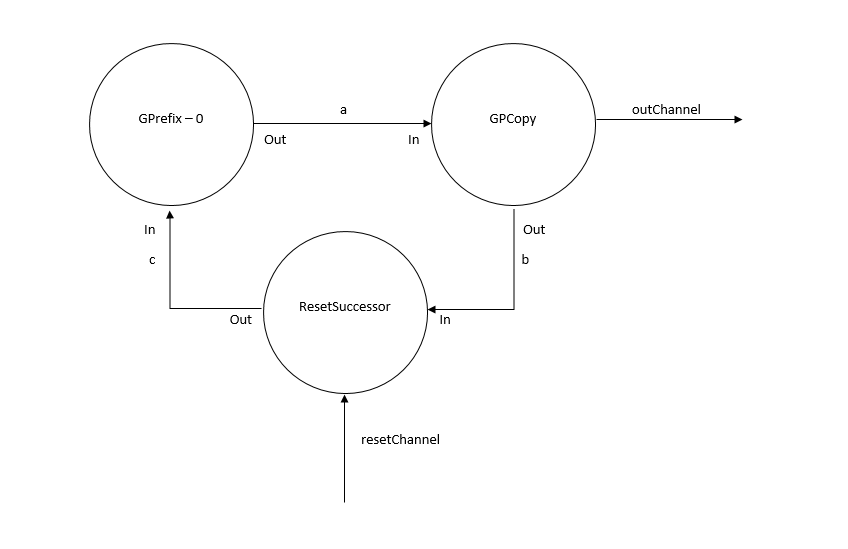
**Questions**

GPrint prints the output in a not ordered, non-tabular way which looks really messy, whereas GParPrint will print each processes at the correct stage in a neat, order manner. Therefore, it is much easier to just build this new process that is far more dynamic and pleasant in its printing manner.

**Exercise 4-1**

**Questions**

1. Removing the **inChannel.read()** line will make it so the original value will not be passed into ResetPrefix, so it will keep GSuccessor process busy, after the reset value has been sent out to GCopy, GSuccessor can finally send out the original number to ResetPrefix which will then continue circulating alongside the reset value. The system now has 2 numbers circulating about it. Adding a third value to the system will deadlock it as the other 2 processes will be busy and will not be able to read in any values and the only ready process ResetPrefix will now also be busy due to a new number being passed in.

**Exercise 4-2**

**ResetSuccessor.groovy**

// deal with inputs from resteChannel and inChannel

// use a priSelect

**def** index = alt.priSelect()

**if** (index == 0){

**def** resetValue = resetChannel.read()

inChannel.read()

outChannel.write(resetValue + 1)

}

**else** {

**def** inputValue = inChannel.read()

outChannel.write(inputValue + 1)

}

**ResetNumbers.groovy**

**def** testList = [ **new** GPrefix ( prefixValue: initialValue,

outChannel: a.out(),

inChannel: c.**in**() ),

**new** GPCopy ( inChannel: a.**in**(),

outChannel0: outChannel,

outChannel1: b.out() ),

// requires a constructor for ResetSuccessor

**new** ResetSuccessor ( inChannel: b.**in**(),

outChannel: c.out(),

resetChannel: resetChannel )

]

**Questions**