Concurrency and occam-π

occam Exercises (preliminary)

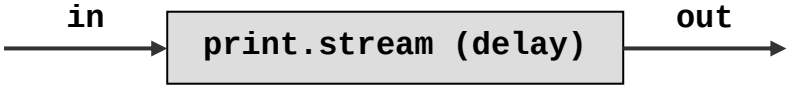
Exercise 1:

[For all these exercises, starter files are given in your **exercises** folder. The file for this one is **q1.occ**.]

S0 and **S1** are two processes that output a stream of (**INT**) numbers:

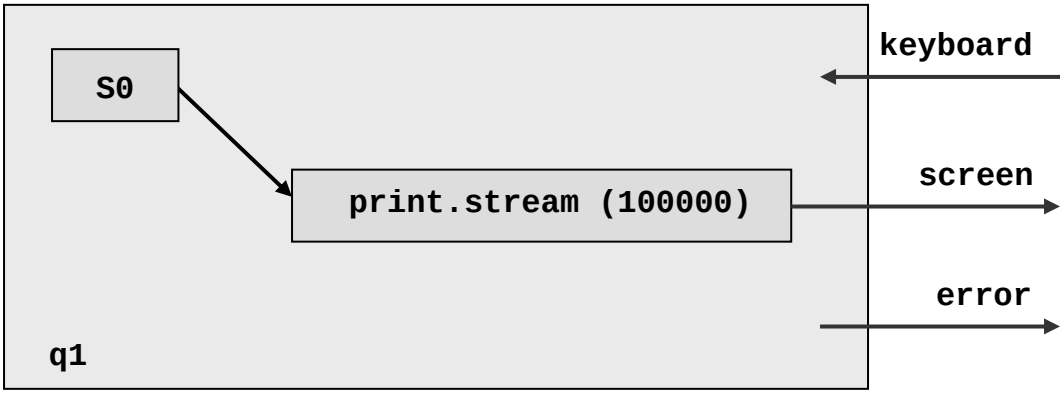
S0 outputs the even numbers (0, 2, 4, 6, …) and **S1** outputs the odd numbers (1, 3, 5, 7, …). Your starter file declares **PROC**s for **S0** and **S1**, but their bodies are only **SKIP** (do nothing). Edit those bodies so they do what they are supposed to do.

print.stream is a process that prints an input stream of (**INT**) numbers to a (**BYTE**) output channel, which will below be connected to the **screen!** output channel of the **q1** process:



print.stream prints one number per line with a delay of (at least) **delay** microseconds after each line. The full coding is given in your starter file.

The *last* process given in your starter file is **q1**, whose header contains the standard (**BYTE**) channels (**keyboard?**, **screen!** and **error!**) currently required by the *Transterpreter*. Test your **S0** process by changing the **SKIP** (initially in the body of **q1**) into the following circuit:



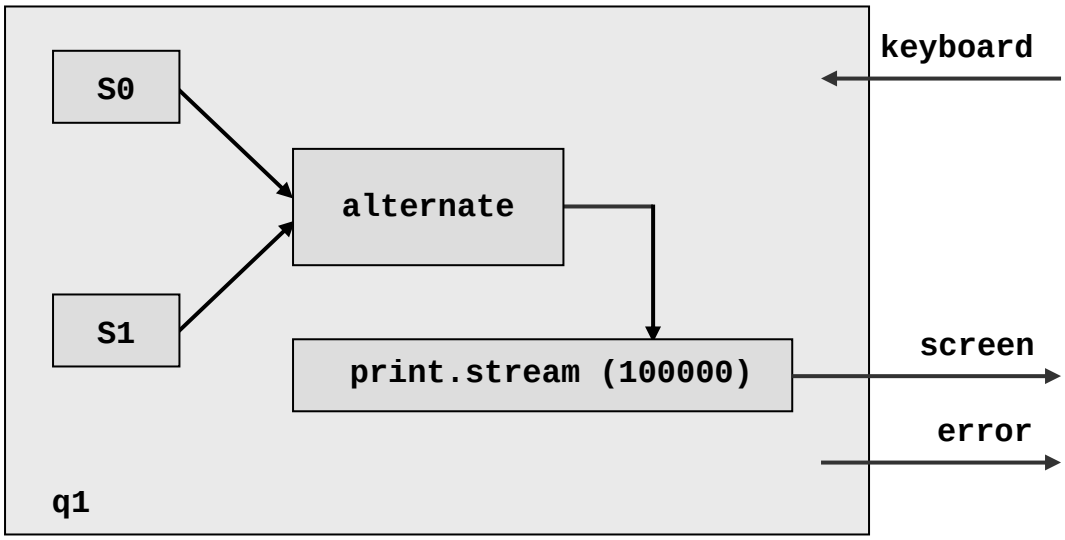
Choose your own name for the channel connecting **S0** and **print.stream**. Compile and run this process. Then, change it to test **S1**.

/Continued …

alternate is a process with two (**INT**) input channels and one (**INT**) output channel:

Again, the given body of **alternate** is just a **SKIP**. Change this to the following behaviour. It assumes an infinite stream of numbers offered to its input channels. **alternate** first takes a number from its **in0?** channel and outputs it on **out!**. Then, it takes a number from its **in1?** channel and outputs it on **out!**. It repeats these two operations for ever.

Test this by modifying your **q1** process to the following circuit:



which should display a column of all the numbers (0, 1, 2, 3, 4, 5, 6, 7, …).

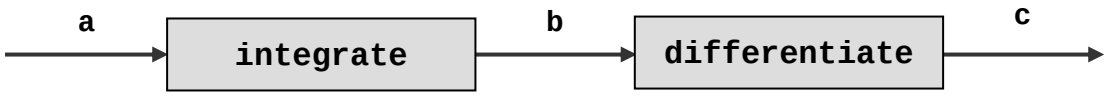
Next – *without changing any of the processes* **S0**, **S1**, **alternate** *or* **print.stream** – modify your **q1** system so that the column of numbers printed excludes all multiples of 5. [*Hint:* define, implement and insert a suitable **filter** process somewhere in the **q1** network.]

Next, modify **print.stream** so that it takes an extra (**VAL INT**) parameter that specifies the number of columns of output produced. For example, if given the argument 3, it would tabulate its first 3 inputs on a single line, then the next 3, then the next 3, etc.

Finally, modify your **q1** process to use your modified **print.stream**.

Exercise 2:

In the same style as the **integrate** *process studied in the course,* design and implement a **differentiate** process that undoes the effect of the former – i.e. if we build the pipleline:



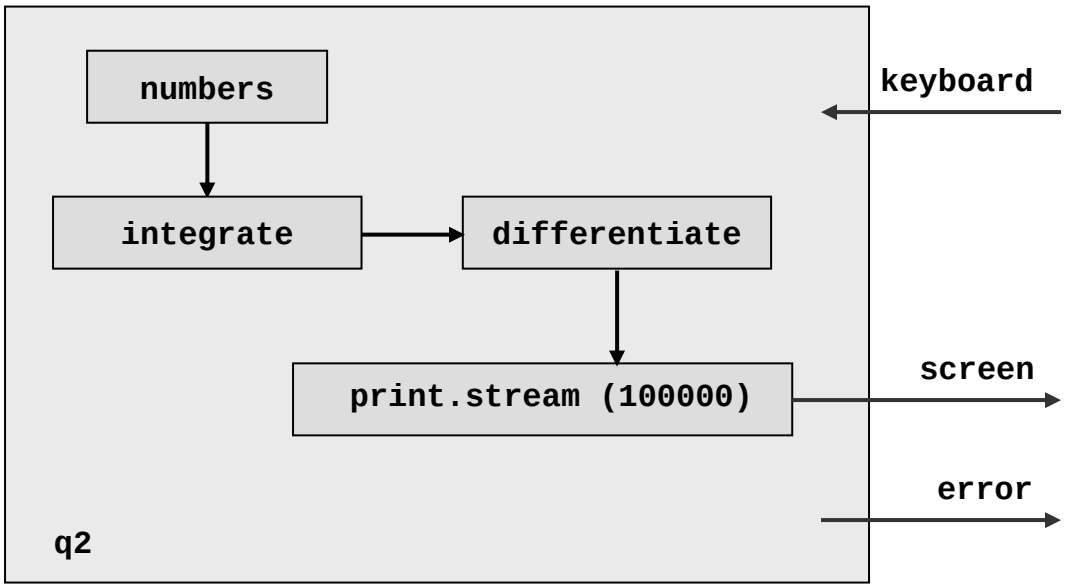
the stream of numbers emerging from channel **c** will be the same as that which flowed in on channel **a**.

Hint1: **integrate** was built to do running sums – **differentiate** needs to do running differences.

Hint2: **differentiate** can be built as a network of three processes. You will certainly need a **minus** process, modified from the **plus** process given in the course slides. You need *two* more from the set: **delta**, **tail** and **prefix**. [*Note:* **differentiate** can also be built as a fairly simple *serial* process – but, for this exercise, that is banned! The purpose of the exercise is to exercise you in parallel design.]

Note: all the processes given in the *'Legoland'* slides are in the **course.lib** – you do not need to retype them. Your starter file imports them into your program in its first two lines. You *will* need to introduce the **minus** process since that is *not* in **course.lib**.

Test your **differentiate** process by building the pipeline:



which should, of course, produce the screen output: 0, 1, 2, 3, 4, 5, …

Warning: make sure the initial 0 is produced!