CSCC24 2023 Summer – Assignment 2
Due: Sunday July 2, 11:59PM
This assignment is worth 10% of the course grade.

In this assignment, we investigate some properties of lazy and/or self-referencing data structures.

This assignment is mainly theory and calculations. Please hand in your answer in a text file A2.txt

Question 1: iterate

In lectures we have seen that the library function `iterate` is handy for generating an infinite list of the form $[x, f(x), f(f(x)), \ldots]$ to help with search problems. We now investigate its memory cost under different use cases.

For concreteness and focus, we work a special case, the following function `r`:

```
r x = x : r (x+2)
```

1(a): The nth element [6 marks]

The following function gives the nth item (base-0 indexing) of a list, assuming it's long enough. (It is basically `(!!)` in the library.)

```
get 0 (x:_) = x
get n (_:xs) = get (n-1) xs
```

Show the lazy evaluation steps of `get 3 (r 10)` until you get the numeric answer.

1(b) [1 mark]

In general, how much space does it take to evaluate `get n (r 10)`? You can give a big-Θ answer.

1(c): Search [6 marks]

You see that `r` (and `iterate`) can be too lazy in its elements. We usually don't mind it because the most common use cases are not asking for the nth element, but rather searching for an element by a criterion.

Below is a toy example (but it gets the point across) that searches for a particular number.

```
find k (x:xs) | k==x = True
              | otherwise = find k xs
```

Show the lazy evaluation steps of `find 16 (r 10)` until you get True.

1(d) [1 mark]

In general, how much space does it take to evaluate `find k (r 10)`? You can give a big-Θ answer. We assume that k can be found.

Question 2: Memory from Feedback Loop

This is basically an exam question last year, but without Functor and Applicative, focusing on the feedback loop; plus, you possess a powerful tool that students last year didn't have: the method of successive approximations! (The one about ⊥.)

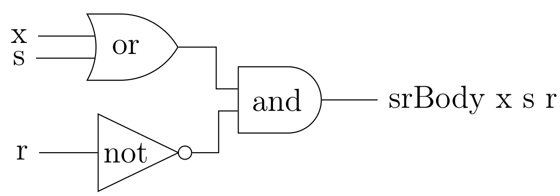
We will use the built-in list type for its nicer syntax, instead of the custom-made type in the exam question. Hence, we use `[Bool]` as infinite lists for inputs and outputs of digital circuits under discrete time.

2(a) [2 marks]

Implement

```
srBody :: [Bool] -> [Bool] -> [Bool] -> [Bool]
srBody x s r = ...
```

to model this circuit (no delay or feedback loop for now):

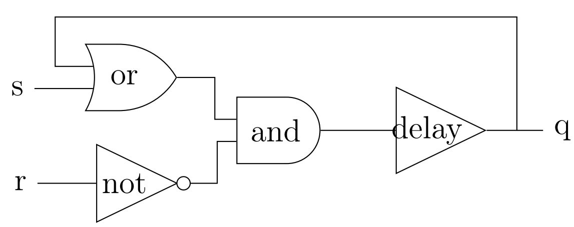


We assume that the inputs are infinite lists; you do not need a base case for the empty list.

Although this part is marked by a TA, starter code (SRLatch.hs) with test cases is provided to help clarify what you need to do. However, you stil have to copy your solution to A2.txt.

SR AND-OR Latch

The SR AND-OR latch (Wikipedia entry) is the feedback circuit shown below; in a discrete-time model such as our Haskell code, an extra delay is also needed[1]:



Soon we will discover its functionality:

- Whenever r (short for "reset") becomes 1 for a moment, q becomes 0, and stays that way even after r goes back to 0.
- Whenever s (short for "set") becomes 1 for a moment (and r stays 0), q becomes 1, and stays that way even after s goes back to 0.

So it is 1 bit of memory, and you write 0 or 1 by sending a pulse to r or s.

In the remainder of this question, I use "0" and "1" instead of False and True to make things look nicer. You may do the same in your answers.

[1]continuous-time models and real gates also have tiny delays

2(b) [2 marks]

First we see why the model needs a delay—by omitting it and seeing what happens. With no delay, the model becomes (with sample input)

```
myS = 0 : 1 : 0 : 0 : 0 : 0 : 0 : ... stays 0 forever
myR = 0 : 0 : 0 : 0 : 1 : 0 : 0 : ... stays 0 forever
bad = srBody bad myS myR
```

Use the method of successive approximations to explain why $bad = \bot$.

2(c) [8 marks]

If the model includes a delay, it becomes (with sample input)

```
myS = 0 : 1 : 0 : 0 : 0 : 0 : 0 : 0 : ... stays 0 forever
myR = 0 : 0 : 0 : 0 : 1 : 0 : 0 : 0 : ... stays 0 forever
q   = 0 : srBody q myS myR
```

Calculate the approximation q_8, which should be enough to illustrate how q behaves.

Since this is doing math, the steps you show are for the purpose of "show your work". You can also write like "`1 | 0 & ~0`" to keep things short.

End of questions.