


## Extra problems, week 5

[Subscribe for email updates.](#)

 PINNED

 No tags yet. [+ Add Tag](#)

Sort replies by: [Oldest first](#) [Newest first](#) [Most popular](#)

Charilaos Skiadas COMMUNITY TA · 15 days ago 

As always, these go far and beyond what is required/expected from the class, and should only be pursued after you have finished all other requirements for the week.

### More fun with streams

Some of this functionality might appear in racket/stream and friends. I would suggest NOT using any of that functionality, and instead implementing these "from scratch". You should be able to avoid mutation. (Might have to slightly rename some of these functions, to avoid shadowing library functions)

1. Create a function `until` that takes in a function `f` and a stream `s`, and applies the function `f` to the values of `s` in succession until `f` evaluates to `#f`. This should be fairly easy.
2. Create a function `map` that takes in a function `f` and a stream `s`, and returns a new stream whose values are the result of applying `f` to the values produced by the stream `s`.
3. Create a function `zip` that takes in two streams `s1` and `s2` and returns a stream that produces the pairs that result from the other two streams (so the first value for the result stream will be the pair of the first value of `s1` and the first value of `s2`).
4. Let us define a "terminating" stream to be one that at some point returns `#f`, and that is considered the "end" of that stream. Write a function `replay` that takes a (possibly terminating) stream `s`, and produces a stream that "replays" `s`, in the sense that when `s` produces an `#f`, then it "rewinds" to the beginning and starts again. You probably have two ways to go about it. One will correctly reproduce any side1.effects that calling the original stream had, the other does not.
5. Define a function `interleave` that takes a list of streams and produces a new stream that takes one element from each stream in sequence. So it will first produce the first value of the first stream, then the first value of the second stream and so on, and it will go back to the first stream when it reaches the end of the list. Try to do this without ever adding an element to the end of a list.
6. Define a function `pack` that takes an integer `n` and a stream `s`, and it returns a stream that produces the same values as `s` but packed in lists of `n` elements. So the first value of the new stream will be the list consisting of the first `n` values of `s`, the second value of the new stream will be the list consisting of the next `n` values and so on.
7. Define a function `cumulative` that takes a function `f(acc, v)` and a stream `s`, and it produces a new stream that accumulates the values as they move along. So the first value `w1` of the new stream will be the first value of `s`, say `v1`, the second value `w2` of the new stream will be `f(w1,`

`v2`) where `v2` was the second value of `s`, the third value `w3` will be `f(w2, v3)` and so on. Essentially doing a fold. For example doing `cumulative` with addition as the function and the stream `ones` should produce the sequence of natural numbers.

8. Write a function `filter` that takes in a predicate function `p` and a stream `s`, and produces a new stream that contains only those values that do not evaluate to `#f` under `p`.
9. (Probably fairly challenging) Write a function `repeats?` that takes a stream `s` and a number `n`, and determines if the stream ends up repeating with a period of no more than `n`, that repetition starting at a point no more than `n` far into the stream. For instance `0 1 2 3 4 5 3 4 5 3 4 5 3 4 5 ...` would be repeating with period 3 and starting at the 4th spot, so it should be found by a call like `(repeats? s 4)`. The function must either return the repeating sequence as a list, or return `#f` if it has determined such a repetition does not start within `n`. Your function must terminate.
10. (This is also fairly challenging) Write a function `partition`, that given a stream `s` and a function `p` produces a pair of streams `s1` and `s2`. The stream `s2` contains those values from `s` that evaluate to `#f` under `p`, while the stream `s1` contains all the others. You must not end up traversing `s` more than once. You can assume that the streams are only called once, in the sense that once `s1` is called and produces `(v, s1-prime)`, then `s1` may not be called again (but of course `s1-prime` might, and it will produce the next value, and then *it* can't be called again but the stream it generated might, and so on). However, the values from `s1` and `s2` maybe be asked in any order (for instance we might generate the first 3 values of `s1` before asking for the first 2 values of `s2` before going back to `s1` and so on). You will likely need to use some mutable structures for this.
11. Write a function `iterate` that takes a stream and returns a function of no arguments, whose successive applications produce one value from the stream at a time. So if `s` is a stream and `(define f (iterate s))`, then the first `(f)` call returns the first value of the stream, the second `(f)` call returns the second value of the stream and so on.
12. Write a version of `cumulative` that uses a function `f(w,v)` and an initial value `w0`, and works similar to foldl.
13. Write a function `from_f` that takes a function `f(n)` and returns the stream whose values are `(f 1), (f 2), (f 3), ...`.

For more Racket problems, consider porting some of the extra problems from the previous weeks.

## Delayed Evaluation in SML

Yes, I must admit to liking SML very much as a language, so you'll see me going back to it even as we have moved on to other languages.

In this instance we will introduce streams in SML, but first we will talk about a model of delayed evaluation. Let's work out a simple signature first:

```
signature LAZY =
sig
  type 'a lazy
  val delay: (unit -> 'a) -> 'a lazy
  val force: 'a lazy -> 'a
end
```

We will now implement a structure that represents this:

```
structure Lazy :> LAZY =  
struct  
  type 'a thunk = unit -> 'a  
  datatype 'a promise = VALUE of 'a | THUNK of 'a thunk  
  type 'a lazy = 'a promise ref  
  
  fun delay f      = ...  
  fun force aRef = ...  
end
```

As you see, we describe a lazy evaluation as a reference to a "promise" value, which can be either an actual value or the unevaluated thunk. You are now asked to implement the two functions `delay` and `force`. All `delay` needs to do is set up a reference, it should be extremely easy. `force` will need to examine what is stored in the reference. If it is the thunk, then it will now need to compute it, change the reference to reflect the value, then return that value. If it is already a value, then its job is pretty much done.

Of course the `thunk` type above isn't strictly necessary, as we could have easily just write `THUNK of unit -> 'a` instead.

Use this structure with something like:

```
val a = Lazy.delay (fn () => (print "hi!\n"; 2));  
Lazy.force a;  
Lazy.force a;
```

We will now move on to discuss streams. We will use the lazy structure internally to hide some implementation details (like looking ahead to the next element). We again hide things behind a signature. Unlike the streams introduced in class, our streams can reach an "end of file" state. This is indicated by throwing an exception when called.

EDIT: This is a revision of an earlier form of this document, where I simplified the description of streams.

```
signature STREAM =  
sig  
  exception EndOfStream  
  type 'a stream  
  val next: 'a stream -> 'a * 'a stream  
  val safe_next : 'a stream -> ('a * 'a stream) option  
  ...  
end
```

We will add a rich set of functions later, but let us start with that. Here's how the structure would look like:

```

structure Stream :> STREAM =
struct
  exception EndOfStream
  datatype 'a stream = Stream of unit -> 'a * 'a stream

  fun next (Stream th) = ...
  fun safe_next s = ...

```

As in our Racket examples, we define a stream as a thunk whose evaluation produces a value and another stream. We had to make this into a datatype, rather than just a type, as it has to refer to itself and there is no other way around that. Note how, since there is only one form in the datatype, we can insert it directly into the definition of `next`. We did not do the same for `safe_next`, as it would most likely call `next` to do its thing.

For now there are two functions on streams (we have no way of creating streams yet, hold your horses). `next` attempts to read the next value out of the stream, producing a pair of a value and a stream, raising an exception if that would have reached the end of stream. `safe_next` returns an option, and is guaranteed not to raise an exception.

We now need a way to generate streams. One way would be to simply expose the `Stream` constructor, and allow people to manually create streams. We will instead offer a more restrictive interface. Users have to instead provide a function `gen: 'b -> ('a * 'b) option` and a seed value of type `'b`. Then the function `gen` will be called on that seed value to generate a value of type `'a` and the next seed value. For each successive value of the stream that we want to compute, we will feed the current value of the seed to `gen`, to obtain a new value for the stream and a new seed. returning an option value of `NONE` indicates that the stream has finished, and should enter an `EndOfStream` state.

So let us add this function to our streams, along with 2 other generators.

You will need to add the following to your signature:

```

val generate: ('b -> ('a * 'b) option) -> 'b -> 'a stream
val generate_inf: ('b -> 'a * 'b) -> 'b -> 'a stream
val generate_simple: ('a -> 'a) -> 'a -> 'a stream

```

Then you will need to implement these. A key step in the creation of a stream, whichever way you go about it, is a something of the form: `Stream (fn () => ...)`.

Now we're off to a good start! You could for instance generate the stream that produces all natural numbers via `Stream.generate (fn n => SOME (n, n + 1)) 1`. The two latter generate functions are an "infinite" version `generate` that never terminates, so doesn't need the option part, and a version of `generate` that simply uses the current value as the seed for the next value, and so can use a simpler generator.

Each of these functions can be written in around 4 lines of code.

We now proceed to add a function `take` that we can use for testing our streams:

```
val take: int -> 'a stream -> 'a list
val safe_take: int -> 'a stream -> 'a list
```

The function `take` simply takes a number of values from the beginning of the stream and creates a list from them. It raises an exception if you ask it to take more than what's there. Its safe variant instead returns what it has so far, if it has reached the end of stream.

Here's an example use:

```
Stream.take 10 (Stream.generate_simple (fn n => n + 1) 1)
```

Here's some more functions to implement (add these declarations to the signature). They should be fairly self-explanatory (and many are motivated by the functions from the Racket problems above).

```
val seq: int -> int -> int stream (* from / step *)
val const: 'a -> 'a stream
val fromList: 'a list -> 'a stream (* EOS once list ends *)
val replay: 'a stream -> 'a stream (* once stream hits EOS, "restart" it *)
val cycleList: 'a list -> 'a stream (* start afresh once list ends *)
val map: ('a -> 'b) -> 'a stream -> 'b stream
(* until moves along until a v with (p v) true. May not terminate. *)
val until: ('a -> bool) -> 'a stream -> 'a * 'a stream
val filter: ('a -> bool) -> 'a stream -> 'a stream
val zip: 'a stream * 'b stream -> ('a * 'b) stream (* EOS when one of the streams does *)
val pack: int -> 'a stream -> 'a list stream (* Will discard shorter list on EOS *)
val interleave: 'a stream list -> 'a stream (* If any EOS, discard it and keep going *)
val cumulative: ('b * 'a -> 'b) -> 'b -> 'a stream -> 'b stream
val from_f: (int -> 'a) -> 'a stream (* f 1, f 2, f 3, ... *)
val iterate: 'a stream -> unit -> 'a
```

↑ 1 ↓ · flag

[Charilaos Skiadas](#) COMMUNITY TA · 14 days ago 🔒

Edit: Simplified a bit the definition of streams.

↑ 0 ↓ · flag

[+ Comment](#)

[Pierre Barbier de Reuille](#) · 14 days ago 🔒

Thanks for the exercises, they are very nice and interesting. I just have a couple of comments:

1. For the racket version of `cumulative`, why not use a signature equivalent to the SML version? By that, I mean with a function looking like the function for fold.
2. I feel that, by using lazy evaluation, you are kind of cheating for the streams. In particular, if the stream has side effects, with your implementation the side effects will be visible only on the first call to the stream. Although this might be desirable, this should be a decision made by the designer of the stream, not the stream library.

Otherwise, I found that another fun and useful exercise on the racket streams is the following:

Implement a `shadow-stream` function that takes a stream and return a stream. The returned stream should behave exactly like the original one, although if the stream is evaluated again, none of the side-effect should be executed again. This would be an example output, assuming `nat-stream` is the stream of natural numbers:

```
> (define print-nat-stream (map (lambda (x) (begin (print x) x)) nat-stream))
> (define shadowed-stream (shadow-stream print-nat-stream))
> (stream-for-n-steps shadowed-stream 10)
12345678910'(1 2 3 4 5 6 7 8 9 10)
> (stream-for-n-steps shadowed-stream 10)
'(1 2 3 4 5 6 7 8 9 10)
```

↑ 2 ↓ · flag

[Charilaos Skiadas](#) COMMUNITY TA · 13 days ago 🔗

Yeah not too sure why I made the two cumulatives different.

I changed the SML streams yesterday to not use lazy evaluation. It was complicating things unnecessarily. You probably were working with the earlier version.

↑ 0 ↓ · flag

[Pierre Barbier de Reuille](#) · 13 days ago 🔗

Indeed, I think it's much better this way!

↑ 0 ↓ · flag

---

[+ Comment](#)

[Charilaos Skiadas](#) COMMUNITY TA · 13 days ago 🔗

Added a couple more problems.

↑ 0 ↓ · flag

---

[+ Comment](#)

 **HANG HANG** · 12 days ago 

For 9. repeats?

I can't figure out why there is an algorithm to decide whether a black box stream will repeat at some point or not.

If the algorithm decide a stream  $s$  will repeat after examining the first  $K$  values, what if the stream stop repeating after the  $K$ th value?

↑ 0 ↓ · flag

**Charilaos Skiadas** COMMUNITY TA · 12 days ago 

Hm I suppose that's a fair point. I'm sort of envisioning a situation where the stream's state is determined by the last 3-4 values or something like that. But you are correct, it's not very clear what the problem asks you to do. Think of it as an open-ended question I suppose ;)

↑ 0 ↓ · flag

**Pierre Barbier de Reuille** · 12 days ago 

What I did is the following:

I suppose the stream repeats if the matching patterns can be completed after the element  $n$ . For example, if the stream returns `1 2 3 1 2 3 1 2 4`, if  $n$  is 4, then it matches (the pattern is `1 2 3` and `1 2 3 1 2 3` goes beyond the position 4), but if  $n$  is 7 or more, then it won't match as it will see the `4` as to finish the sequence after `1 2 3 1 2 3 1` it will need to see the `2 4`.





Otherwise, indeed, there is no way to actually assess if the stream is really repeating.

↑ 0 ↓ · flag

[+ Comment](#)

New post

To ensure a positive and productive discussion, please read our [forum posting policies](#) before posting.

<b>B</b>	<i>I</i>			 Link	<code>	 Pic	Math		Edit: Rich ▼	Preview
<div></div>										

☐ Make this post anonymous to other students

☒ Subscribe to this thread at the same time

Add post