Computer Science 320 (Spring, 2020)
Principles of Programming Languages

Programming Assignment 3: Streams

Out: Friday, February 21, 2020
Due: Tuesday, March 3, 2020, by 11:59 pm

Potentially infinite lists are usually called "streams," or "sequences" when also referring to input/output channels. We will call them "streams" throughout this assignment. More precisely, a stream is a countable list of elements all of the same type; "countable" means the elements can be put in a one-one correspondence with an initial segment of the natural numbers $0, 1, 2, ..., n$ (in which case the sequence is finite) or with the whole set of natural numbers $0, 1, 2, ...$ (in which case the sequence is infinite). *Since a stream may be finite, your functions will need to take this possibility into account.* An appropriate definition of the polymorphic datatype `stream` is the following:

```
type 'a stream = Nil | Cons of 'a * (unit -> 'a stream)
```

which you will have to use in every part of this assignment.

Remark: Depending on the purpose, streams are sometimes limited to infinite lists, *i.e.*, the standard (finite) lists are excluded from streams. This more restricted definition will be presented in lecture.

1 Questions to be solved

There are 12 separate questions in this assignment. Your answers should be written in OCaml.

(A) Define a function `read :  'a stream -> 'a -> ('a * 'a strem)` which takes a potentially infinite stream as its first argument, and a default element as its second argument. If the stream is non-empty, `read` should return the element at the front of the stream paired with the rest of the stream. If it is empty, `read` should return the default element paired with a representation of the empty stream.

(B) Define a function `skip : 'a stream -> ('a -> Bool) -> 'a stream` which takes a potentially infinite stream along with a function of type `'a -> Bool` and returns a stream which skips all elements for which the function returns `false`. "Skipping" here means returning the next element in the stream for which the function returns `true`.

(C) Define a function `mergeS : 'a stream -> 'a stream -> 'a stream` which takes two potentially infinite streams and merges them into a single stream, taking elements alternately. For example, if `nats` is the infinite stream of all natural numbers $0, 1, 2, \ldots$, and `nines` is the infinite stream $9, 9, 9, \ldots$ (the numeral 9 is repeated infinitely many times), then `(mergeS nats nines)` should return the stream $0, 9, 1, 9, 2, 9, \ldots$.

(D) Define a function `twoseq : 'a stream -> 'a stream -> 'a stream` which takes two streams and returns a stream which has every element in the first stream, and if it is finite (it might not be), once the elements in the first stream are exhausted, also has every element in the second stream.

(E) Define a function `dupk : 'a -> int -> 'a stream -> 'a stream` which takes an element of some type `'a`, an integer k, and a stream of type `'a stream`. The function should place k copies of the element at the beginning of the stream.

(F) Define a function `repeatk : int -> 'a stream -> 'a stream` which takes a potentially infinite stream $x_1, x_2, x_3, \ldots$ and returns a stream of the form:

$$\underbrace{x_1, \ldots, x_1}_{k\text{times}}, \underbrace{x_2, \ldots, x_2}_{k\text{times}}, \underbrace{x_3, \ldots, x_3}_{k\text{times}}, \ldots$$

In other words, the function should return a stream where every element is duplicated k times. Points can be deducted for inefficient solutions.

(G) Define a function `addAdjacent : int stream -> int stream` which takes a stream of integers $n_1, n_2, n_3, n_4, \ldots$ and returns an integer stream of the form

$$n_1 + n_2,\ n_3 + n_4,\ n_5 + n_6,\ \ldots$$

In other words, the new stream should consist of sums of adjacent elements in the old stream. If the input stream is finite of odd length, the last element in the stream should be thrown away.

(H) Define a function `addAdjacentk : int -> 'a stream -> 'a stream` which is a generalization of the function from part (G). Given a stream $n_1, n_2, n_3, n_4, \ldots$, it should return a stream:

$$n_1 + \ldots + n_k,\ n_{k+1} + \ldots + n_{k+k},\ \ldots$$

Note that if the length of the list is not a multiple of k, the remainder of the elements should be thrown away.

(I) Define a function `binOpSeq : ('a -> 'b -> 'c) -> 'a stream -> 'b stream -> 'c stream` which takes a function `f` of two arguments along with two streams $x_1, x_2, \ldots$

and $y_1, y_2, \ldots$, and returns a stream consisting of the results of applying that function to each corresponding pair of elements:

$$(\texttt{f}\, x_1\, y_1),\ (\texttt{f}\, x_2\, y_2),\ (\texttt{f}\, x_3\, y_3),\ \ldots$$

The function should stop returning results as soon as one of the two input streams runs out of arguments to supply to the function.

(J) Define functions `addSeq` and `mulSeq`, both of type `int stream -> int stream -> int stream`, which add or multiply the corresponding elements of two streams and return the stream of results. *For credit, you must use the function* ***`binOpSeq`*** *defined in part (I).*

(K) Define a function `zipS : 'a stream -> 'b stream -> ('a * 'b) stream` which takes two streams $x_1, x_2, \ldots$ and $y_1, y_2, \ldots$, and returns a stream consisting of the pairs of corresponding elements in the two streams:

$$(x_1,\, y_1),\ (x_2,\, y_2),\ (x_3,\, y_3),\ \ldots$$

The function should stop returning results as soon as one of the two input streams runs out of arguments to supply to `zipS`. *For credit, you must use the function* ***`binOpSeq`*** *defined in part (I).*

(L) Define a function `unzipS : ('a*'b) stream -> ('a stream * 'b stream)` which takes a stream of pairs:

$$(x_1,\, y_1),\ (x_2,\, y_2),\ (x_3,\, y_3),\ \ldots$$

and returns a pair of the two input streams:

$$((x_1,\, x_2,\, \ldots),\ (y_1,\, y_2,\, \ldots))$$

For this part, you can get credit without using the function ***`binOpSeq`*** *defined in part (I).*

2 Submission guidelines

Late submissions will not be accepted. You can use GradeScope to confirm that your program adheres to the specification outlined. Only your last GradeScope submission will be graded. This means you can submit as many times as you want. If you have any questions please ask well before the due date.

2.1 GradeScope submission instructions

When you log into GradeScope and pick the CS 320 course you will see an assignment called *streams*. The total of points for this assignment is 50. Click *Submit* and choose your source file. **The solution you submitted must be in an .ml file named** `streams.ml`.

You can submit as many times as you wish before the deadline.

2.2 Additional resources

For information on how to run OCaml please see the following references:

- `https://caml.inria.fr/pub/docs/manual-ocaml/stdlib.html`
- `https://caml.inria.fr/pub/docs/manual-ocaml/`

You will find resources for these languages on the Piazza course website under the Resources page.