There are 6 problems in this set, each worth as marked, for a total of 100 points. The harder problems are marked with a single * (average difficulty) or two ** (higher-than-average difficulty). For the easy points, start with the unmarked problems.

Potentially infinite lists are usually called “streams”. This is what Scheme programmers call them, as in the Abelson-Sussman book, for example. But a stream in ML also refers to an input/output channel. So, we will call them “sequences” instead, throughout this assignment. More precisely, a sequence is a countable list of elements all of the same type; “countable” means the elements can be put in a 1-1 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). In Problems 1 to 4 – but not Problems 5 and 6 – the type of sequences is given by the following declaration:

```
datatype 'a seq = Nil | Cons of 'a * (unit -> 'a seq);
```

which means that a sequence is either empty (in which case it is represented by “Nil”) or non-empty (in which case it is represented by the constructor “Cons” applied to a pair of the form “(x, xf)” where “x” is the head and “xf” is a function to compute the tail).

**Problem 1** *(15 points)* Write the SML code for the following 3 functions on sequences:

1. `null`
2. `drop`
3. `toList`

Declare “null” and “drop” in analogy with the versions of these functions on lists, which you can find in the library structure List. The function “toList” should convert a finite sequence into a list, with the same entries and in the same order.

* **Problem 2** *(15 points)* Define SML functions `repeatEach` and `addAdjacent`:

```
repeatEach: 'a seq * int -> 'a seq
addAdjacent: int seq -> int seq which behave as follows:
```

1. Given a sequence `xq` whose entries are `x_0, x_1, x_2, ...` and a positive integer `k`, `repeatEach (xq, k)` returns a sequence whose entries are:

```
x_0, ..., x_0, x_1, ..., x_0, x_1, ..., x_2, ..., x_2, ...

k times  k times  k times
```

2. Given an infinite sequence `xq` of integers `n_0, n_1, n_2, n_3, ...`, `addAdjacent (xq)` returns an integer sequence whose entries are:

```
n_0 + n_1, n_2 + n_3, n_4 + n_5, ...
```

1
**Problem 3** *(15 points)* The function \texttt{allOnes} takes no input and is declared with the following code:

\begin{verbatim}
fun allOnes () = Cons (1, fn () => allOnes ());
\end{verbatim}

The infinite sequence of all 1's is returned by evaluating “\texttt{allOnes ()}”.

1. Consider the following declaration:

\begin{verbatim}
fun foo (xq : int seq) = Cons (1, fn () => add (foo xq, foo xq));
\end{verbatim}

What is the sequence returned by \texttt{foo (allOnes ())}? Describe the elements in the sequence precisely, preferably with a mathematical formula. Two lines will suffice.

2. Write a function \texttt{mult} of type \texttt{int seq * int seq -> int seq}, analogous to the function \texttt{add}, which multiplies the corresponding elements of its two input sequences.

* 3. Complete the following declaration

\begin{verbatim}
fun facts (xq : int seq) = Cons (1, fn () => mult (<?>, <?>));
\end{verbatim}

so that the evaluation of \texttt{facts (from 0)} returns the sequence whose \texttt{n}-th entry (starting from 0) is the factorial of \texttt{n}.

**Problem 4** *(15 points)* Consider the definition of the datatype \texttt{intTree} in Handout 34, which represents potentially infinite binary trees where internal nodes are labelled with integers.

1. Write a datatype declaration for \texttt{'a tree}, which is the polymorphic version of \texttt{intTree} representing potentially infinite binary trees where labels are items of type \texttt{'a}.

* 2. Write a function \texttt{fromTreeToSeq: 'a tree -> 'a seq}, which builds a sequence consisting of all the labels in a given binary tree. The order of the labels in the output sequence must be the result of a breadth-first traversal of the input tree (left-to-right, top-to-bottom).

* **Problem 5** *(20 points)* Exercise 5.25, page 194, in [P].

** Problem 6 ** *(20 points)* Exercise 5.26, page 194, in [P].
(* Useful SML code for PSet 11 -- most of it from Handout 34 *)

(* A datatype of sequences: *)
datatype 'a seq = Nil | Cons of 'a * (unit -> 'a seq);

(* The head, tail, and cons functions for sequences: *)
fun hd Nil = raise Empty
  | hd (Cons(x,xf)) = x;
fun tl Nil = raise Empty
  | tl (Cons(x,xf)) = xf();
fun cons (x,xq) = Cons(x, fn () => xq);

(* Converting a list to a sequence: *)
fun fromList l = List.foldr cons Nil l;

(* The increasing sequence of integers starting from k: *)
fun from k = Cons (k, fn () => from (k+1));

(* The sequence of all 1's is produced with "allOnes()": *)
fun allOnes () = Cons (1, fn () => allOnes ());

(* Calling "take(xq,n)" returns the first n elements of xq as a list: *)
fun take (xq, 0)  = []
  | take (Nil, n) = raise Subscript
  | take (Cons(x,xf), n) = x :: take (xf (), n-1);

(* Appending two sequences: *)
fun append (Nil, yq) = yq
  | append (Cons(x,xf),yq) = Cons(x, fn () => append(xf (),yq));

(* Interleaving two sequences: *)
fun interleave (Nil,yq) = yq
  | interleave (Cons(x,xf),yq) = Cons(x, fn () => interleave(yq, xf()));

(* The "map" function for sequences is: *)
fun map f Nil = Nil
  | map f (Cons (x,xf)) = Cons(f x, fn () => map f (xf ()));

(* The "filter" function for sequences is: *)
fun filter pred Nil = Nil
  | filter pred (Cons (x,xf)) =
      if pred x then Cons(x, fn () => filter pred (xf ()))
      else filter pred (xf ());

(* The function "iterates" generates an infinite sequence of the form *)
(* x, f(x), f(f(x)), f(f(f(x))), ...: *)
fun iterates f x = Cons(x, fn () => iterates f (f x));

(* The functions "squares" and "add" are examples of arithmetical functions *)
(* on sequences of type "int seq": *)
fun squares Nil : int seq = Nil
  | squares (Cons(x,xf)) = Cons(x*x, fn () => squares (xf ()));
fun add (Cons(x,xf), Cons(y,yf)) = Cons(x+y, fn () => add (xf (), yf ()))
  | add _ : int seq = Nil;