# STANDARD ML

## SEQUENCES

# LAZY LISTS

- elements are not evaluated until their values are required
- **may** be infinite
- example: a sequence of all even integers `$0, 2, -2, 4, \ldots$`

# LAZY LISTS IN ML

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

fun head (Cons (x, _)) = x;

fun tail (Cons (_, xf)) = xf();
```

ML evaluates `E` in `Cons(x,E)`, so to obtain laziness we must write `Cons(x, fn()=>E)`

# EXAMPLES OF SEQUENCES

```
fun from k = Cons (k, fn() => from (k+1));

from 1;
```

```
tail it;
```

```
fun squares Nil = Nil
  | squares (Cons (x, xf)) =
        Cons (x*x, fn() => squares (xf()));
```

```
squares (from 1);
```

```
head (tail (tail (tail (tail it))));
```

# ELEMENTARY SEQUENCE PROCESSING

## `addq`

implement `addq` that takes two integer sequences and adds them element-wise

```
...
(*val addq = fn : int seq * int seq -> int seq*)
```

```
fun addq (Cons (x, xf), Cons (y, yf)) =
        Cons (x+y, fn() => addq (xf(), yf()))
  | addq _ = Nil;
```

## `appendq`

implement `appendq` that appends two sequences

```
...
(*val appendq = fn : 'a seq * 'a seq -> 'a seq*)
```

```
fun appendq (Nil, yq) = yq
  | appendq (Cons(x, xf), yq) =
        Cons (x, fn() => appendq (xf(), yq));
```

what would `appendq(xq,yq)` be if `xq` is infinite?

## `mapq`

implement `mapq` that applies a function on the elements of a sequence

```
...
(*val mapq = fn : ('a -> 'b) -> 'a seq -> 'b seq*)
```

```
fun mapq f Nil           = Nil
  | mapq f (Cons (x,xf)) =
        Cons (f(x), fn()=>mapq f (xf()));
```

## `filterq`

implement `filterq` that filters a sequence based on a predicate

```
...
(*val filterq = fn : ('a -> bool) -> 'a seq -> 'a seq*)
```

```
fun filterq pred Nil = Nil
  | filterq pred (Cons (x,xf)) =
        if pred x
        then Cons (x, fn()=>filterq pred (xf()))
        else filterq pred (xf());
```

## `interleaveq`

implement `interleaveq` that interleaves two sequences

e.g.: interleaving `1,2,3,...` and `11,12,13,...` returns: `1,11,2,12,3,13,4,...`

```
...
(*val interleaveq = fn : 'a seq * 'a seq -> 'a seq*)
```

```
fun interleaveq (Nil, yq)        = yq
  | interleaveq (Cons(x,xf),yq) =
        Cons (x, fn()=>interleaveq (yq, xf()));
```

`dropq` takes a sequence `s` and a positive number `n` and returns `s` without its first `n` elements

```
...
(*val dropq = fn: 'a seq -> int -> 'a seq*)
```

```
fun dropq seq 0 = seq
  | dropq Nil _ = Nil
  | dropq (Cons(x, xf)) n = dropq (xf()) (n - 1);
```

`seqToList` takes a sequence and returns a list of its elements

```
...
(*val seqToList = fn: 'a seq -> 'a list*)
```

```
fun seqToList Nil = []
  | seqToList (Cons(x, xf)) = x::(seqToList (xf()));
```

`listToSeq` takes a list and returns a sequence of its elements

```
...
(*val listToSeq = fn: 'a list -> 'a seq*)
```

```
fun listToSeq [] = Nil
  | listToSeq (x::xs) = Cons (x, fn () => listToSeq xs);
```

## סעיף א'

עליכם לכתוב את הפונקציה `fraction` המקבלת שני מספרים שלמים בסגנון currying, ומחזירה seq של רצף הספרות אחרי הנקודה העשרונית עבור תוצאת החלוקה של המספר הראשון ($m$) במספר השני ($n$). ניתן להניח בסעיף זה כי $m < n$ כלומר, שתוצאת החילוק קטנה מ-$1$. במידה ומתבצעת חלוקה ב $0$ יש לזרוק חריגה מתאימה.

***לדוגמא:***

```
val fraction = fn : int -> int -> int seq

- fraction 1 7;
val it = Cons(1, fn) : int seq
- tail it;
val it = Cons(4, fn) : int seq
- tail it;
val it = Cons(2, fn) : int seq
...
```

. . .

```
fun fraction m n =
  if m mod n = 0
  then Nil
  else Cons ((m * 10 div n mod 10), fn () => fraction (m*10) n);
```

## סעיף ב'

כעת כתבו את הפונקציה `lazy_divide` המקבלת שני מספרים שלמים בסגנון curried, ומחזירה tuple המכיל באינדקס הראשון את החלק השלם של החלוקה, ובאינדקס השני את רצף הספרות שלאחר הנקודה כ-seq. במידה ומתבצעת חלוקה ב-$0$ יש לזרוק חריגה מתאימה.

***לדוגמא:***

```
val lazy_divide = fn : int -> int -> int * int seq

- lazy_divide 22 7;
val it = (3, Cons(1, fn)) : int * int seq
- tail (#2 it);
val it = Cons(4, fn) : int seq
- tail it;
val it = Cons(2, fn) : int seq
```

...

```
fun lazy_divide m n = (m div n, fraction (m mod n) n);
```

```
datatype 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
datatype 'a option = NONE | SOME of 'a;
datatype 'a node =
  Node of 'a * (unit -> 'a node option) * (unit -> 'a node option);
type 'a lazy_tree = unit -> 'a node option;
```

```
fun take _ 0 = []
  | take Nil _ = []
  | take (Cons(x, xf)) n = x::(take (xf()) (n - 1));

Control.Print.printLength := 1000;
Control.Print.printDepth := 1000;
```

# define some trees

```
fun t1 () = NONE;
fun t2 0 () = SOME (Node (0, t1, t1))
  | t2 n () = SOME (Node (n, t2 (n div 2), t2 (n - 1)));
fun t3 () = SOME (Node (100, t2 8, t2 7));
```

# implement lazy bfs traversal of lazy trees

```
...
```

```
local
  fun aux [] [] = Nil
    | aux [] ts = aux (map (fn t => t ()) (List.rev ts)) []
    | aux (NONE::ns) ts = aux ns ts
    | aux ((SOME (Node (h, l, r)))::ns) ts = Cons(h, fn () => aux ns (r::l::ts))
in
  fun bfs t = aux [t ()] []
end;
```