Principles of Programming Languages 2012
Practical Session 9 – ML and Lazy Lists

Part 1 - ML

1. ML- Introduction

ML is a statically typed programming language that belongs to the group of Functional Languages like Scheme and LISP. The major difference between ML and Scheme is:

1. ML is statically typed language. The static typing obeys type correctness rules, the types of literals, values, expressions and functions in a program are calculated (inferred) by the Standard ML system. The inference is done at compile time. Scheme is a dynamically typed language.
2. ML enables the definition of new types.
3. ML has a general mechanism for parameter passing: PATTERN MATCHING.

1.1. Getting Started with ML

- SML is installed in the CS labs.

At Home:

- Install from course web page "Useful links", "download Standard ML" link.
- Open the SML Interpreter. Expressions can be evaluated there.
- Create a *.sml file and edit it with any text editor.
  - Recommended: use the free Notepad++ to (select: Language> Caml to highlight text).
- The file can be loaded to the interpreter.

At the prompt, which is a dash symbol -,

- You can type an expression or a declaration.
- You can load the file using the command:

use("file name including path");

Or (very recommended):

1. **Write a file-loading function:**

```
- val load = fn (file_name) => use("E:/PPL/ML/" ^ file_name ^ ".sml");
```

2. **Then at the prompt write:**

```
load("filename"); (without ".sml")
```

2. **Recursive Functions** (Optional)

Recursive functions can be defined in the global scope using standard naming (using the global environment for binding).

Example 1

Given b and n (n is natural, b>0) calculate b^n according to the following formula

$$b^n = \begin{cases} \left(b^2\right)^{n/2} & n/2 \in Z \\ b \cdot b^{n-1} & else \end{cases}$$

The definition In **Scheme**

```
(define exp (lambda (b n)

  (cond ((= n 1) b)

        ((even? n) (exp (* b b) (/ n 2)))

       (else (* b (exp b (- n 1)))))))
```

In ML: Let us try a function definition as before:

```
val exp = fn(b, n)=>
                    if n=1 then b
                    else if n mod 2 = 0
                            then exp (b*b, n div 2)
                            else b * exp(b, n-1);
```

stdIn:16.18-16.21 Error: unbound variable or constructor: exp
stdIn:17.38-17.41 Error: unbound variable or constructor: exp

What do you think is the cause of the error?
Answer:
In ML, the compiler (TYPE INFERENCE mechanism) checks the function body at STATIC (compile) time. It reaches the recursive call (the variable 'exp'), and it is not appeared in the type environment. Therefore, it creates an unbound variable error. The reason that this problem doesn't happen in Scheme is that Scheme doesn't read the function's body at STATIC time, and at RUN time, the function is already defined.

ML introduces the keyword '***rec***' for the declarations of recursive functions:

```
(*
Signature: exp(b,n)
Purpose: Calculates the power of a natural number
Type: Int*Int-> Int
*)
val rec exp = fn(b, n)=>
                    if n=1 then b
                    else if n mod 2 = 0
                            then exp (b*b, n div 2)
                            else b * exp(b, n-1);
```

Note that the expression fn$(x_1, \ldots, x_n)$=> is the procedure value constructor in ML and that its equivalent in scheme is lambda$(x_1, \ldots, x_n)$.

---

3. Clausal Function Definition Using Pattern Matching

Clausal function expressions exploit ML's general mechanism for parameter passing: PATTERN MATCHING. Pattern matching takes 2 arguments: A PATTERN and an EXPRESSION. It checks whether they are equal, to they both need to have an equality type.

PATTERN Definition:

1. A pattern is an ML expression that consists of:
   a. variables.
   b. constructors.
   c. wildcard character _.

   The constructors are:
   a. Numeric, character, Boolean and string constants.
   b. Pair and tuple constructors: Like (a,0), (a,0,_).
   b. List and user defined value constructors.
2. A variable may occur at most once in a pattern.
3. The function constructor '=>' cannot appear in a pattern
   (FUNCTION is not an equality type).

ML Patterns: *1*, *true*, (*1*, *false*), (*a*, *b*), [1,2,4].

Not every ML expression is a pattern. For example, the exp procedure defined above, +, =>, and (if 1 = a then true else false), are all ML expressions but are not patterns.

Example 2

Consider again the *exp* procedure defined in example 1.The **exp** function is defined using the patterns:

```
      (b, 1)

      (b, n)
```

```
Then,
val rec exp =
     fn (b,1) => b
     | (b,n) => if n mod 2 = 0
                    then exp (b*b, n div 2)
                    else b* exp(b*b, n - 1);
```

When it called with arguments (2,8) -- the arguments expressions match the 2nd pattern. The clausal definition is evaluated in **sequence**: The first clause whose pattern matches the given argument is the one to execute. The rest are ignored (like '**cond**' evaluation in Scheme).

Notice that the "rec" keyword was used (to denote this is a definition of a recursive function). It is important to understand why it should be written here, while in Scheme we didn't need to specify it. It is needed because in ML (unlike in Scheme) the body of the defined function is operated on during define time in order to do type inference (which is not done in Scheme).

4. High Order Functions (optional)

The COMPOSITION of ***f*** after ***g*** is defined to be the function x --> f (g(x)).

The function ***repeated*** computes the nth repeated application of a given function 'f' of one argument. That is, f(f(f ...(f(x))...) n times.

Example 3

```
val rec repeat =
      fn(0,f,x) => x
       |(n,f,x) => repeat((n-1), f, f(x));

val repeat = fn : int * ('a -> 'a) * 'a -> 'a
```

Example:

```
- repeat(4,(fn x => x * x), 2);
val it = 65536 : int
```

Example 4 (**Optional**):

The ***repeated*** function can be partially evaluated, by giving it only one argument. The evaluation creates a single argument function. This process of turning a multi-argument function into a single argument one is called Currying (after the logician Curry):

```
(*
Signature: c_repeat(n, f)
Purpose: create the function that on a given input x returns
f(f(f(….(f(x)))…) , where f is repeated n times.
Type:  Number *  [T1*T1] -> [T1*T1]
Precondition: n>=0
*)
val rec c_repeat =
      fn(n,f) =>
            let val rec helper=
                  fn(0, c)  => (fn(x) => c(x))
                    |(n, c) => helper((n - 1), fn(x)=> f(c(x)))
            in
                  helper(n, f)
            end;

val c_repeat = fn : int * ('a -> 'a) -> 'a -> 'a
```

Note:

1. ML inferred the types of c_repeat and repeat automatically.
2. The type constructor -> is right associative, which means that the type:
   `int * ('a -> 'a) -> 'a -> 'a`
   may also be written as:
   `(int * ('a -> 'a) -> ('a -> 'a))`

---

5. DATA TYPES

Problems that require data beyond numbers or Booleans require the extension of the type system with new datatypes. The introduction of a new data type consists of:

1. Type constructors.
2. Value constructors.

The type constructors introduce a name(s) for the new type, possibly with parameters, while the value constructors introduce labeled values.

5.1 ATOMIC Types

The simplest ADTs are those having no components to select. They are also called ENUMERATION TYPES. The ADT is just a set of values and operations.

Example 5

```
(*
   Type Constructor: direction
   Value Constructor: North, South, East, West
   Purpose: A datatype declaration that creates a new type to represent
            the compass directions.


*)
datatype direction = North | South | East | West;

val move =
  fn((x,y),North) => (x,y+1) ;
  |((x,y),South) => (x,y-1)
  |((x,y),East) => (x+1,y)
  |((x,y),West) => (x-1,y);

val move = fn : (int * int) * direction -> int * int
```

Example

```
- move((4,5), North);
val it = (4,6) : int * int
```

5.2 Composite Types

A composite type is a data type whose values are created by value constructors that take as parameters values of other types.

Example 6 – Complex numbers

We develop a system that performs arithmetic operations on complex numbers. We present two plausible representations for complex numbers as ordered pairs: rectangular form (real part and imaginary part) and polar form (magnitude and angle):

Rectangular Representation: Complex numbers can be viewed as points in a 2 dimensional plan, where the axes correspond to the real and imaginary parts. They can be represented as PAIRS of the coordinates. We call this representation RECTANGULAR

Polar Representation: They also can be represented by the magnitude of the vector from. The origin to the point, and its angle with the x axis. We call this POLAR.

The two representations are interesting because they can conveniently express different operations. The Rectangular representation is convenient for addition and subtraction, while the Polar one is convenient for multiplication and division:

```
real_part(z1 + z2) = real_part(z1) + real_part(z2)
imaginary_part(z1 + z2) = imaginary_part(z1) + imaginary_part(z2)

magnitude(z1 * z2) = magnitude(z1) * magnitude(z2)
angle(z1 * z2) = angle(z1) + angle(z2)
```

In ML, using the type constructors and value constructors (that act like type tags in Scheme), the problem is simple to solve:

```
(* Type constructor: complex
   Value constructors: Rec, Complex.
   Data values of this type have the form:
   Rec(3.0, 4.5), Polar(-3.5, 40.0)
*)

datatype complex = Rec of real * real | Polar of real * real;


(*  AUXILIARY FUNCTION: square *)
- val square = fn x : real => x * x;
val square = fn : real -> real




- val real =
      fn (Rec(x,y) ) => x
      |  (Polar(r,a)) => r * Math.cos(a);
val real = fn : complex -> real

- val imaginary =
      fn (Rec(x,y) ) => y
      |  (Polar(r,a)) => r * Math.sin(a);
val imaginary = fn : complex -> real

- val radius =
      fn (Rec(x,y) ) => Math.sqrt( square(x) + square(y) )
      |  (Polar(r,a)) => r;
val radius = fn : complex -> real


- val angle =
      fn (Rec(x,y) ) => Math.atan( y / x )
      |  (Polar(r,a)) => a;
val angle = fn : complex -> real


- val add_complex =
```

```
      fn (Rec(x, y), Rec(x', y')) => ( Rec( x + x', y + y') )
      |  (Rec(x,y), z) => ( Rec( x + real(z), y + imaginary(z) ) )
      |  (z, Rec(x, y)) => ( Rec( real(z) + x, imaginary(z) + y) )
      |  (z,z') => (Rec( real(z) + real(z'), imaginary(z) + imaginary(z') ) );
val add_complex = fn : complex * complex -> complex
```

```
val sub_complex =
      fn (Rec(x, y), Rec(x', y')) => ( Rec( x - x', y - y') )
      |  (Rec(x,y), z) => ( Rec( x - real(z), y + imaginary(z) ) )
      |  (z, Rec(x, y)) => ( Rec( real(z) - x, imaginary(z) - y) )
      |  (z,z') => (Rec( real(z) - real(z'), imaginary(z) - imaginary(z') ) );
val sub_complex = fn : complex * complex -> complex
```

```
val mul_complex =
    fn (Polar(r, a), Polar(r', a')) => (Polar(r * r', a + a'))
      |  (Polar(r,a), z) => (Polar( r * radius(z), a + angle(z) ) )
      |  (z, Polar(r,a)) => (Polar( radius(z) * r, angle(z) + a ) )
      |  (z, z') => (Polar( radius(z) * radius(z'), angle(z) + angle(z') ) );
val mul_complex = fn : complex * complex -> complex
```

```
(* Pre -condition: r' != 0  *)
val div_complex =
    fn (Polar(r, a), Polar(r', a')) => (Polar(r / r', a - a'))
      | (Polar(r, a), z) => (Polar(r / radius(z), a - angle(z)))
      | (z, Polar(r', a')) => (Polar(radius(z) / r', angle(z) - a'))
      | (z, z') => (Polar(radius(z) / radius(z'), angle(z) - angle(z')));
val div_complex = fn : complex * complex -> complex
```

Example 7 (Optional)

Design a set of procedures for manipulating circles on x-y plane. Suppose we want to do geometric calculations with circles .

Definition of composite data type Circle:

```
(* auxiliary function: square *)

val square = fn(x:real) => x * x
```

```
(*
   Type constructor: circle
   Value constructors: Circle.
   Example: Circle(2.0,4.0,5.0)
*)

datatype circle = Circle of real*real*real;
```

```
(*
  Signature: circ_eq(Circle(x1, y1, r1), Circle(x2, y2, r2))
  Purpose: check if two circles are equale.
  Type: circle * circle -> Boolean
*)
val circ_eq =
  fn (Circle(x1, y1, r1), Circle(x2, y2, r2)) =>
          Real.==(x1, x2) andalso Real.==(y1, y2) andalso Real.==(r1,
r2);
```

```
(*
 Signature: move_circle(Circle(x, y, r), x', y')
 Purpose:  Move circ center point by x on x-axis and by y on y-axis
 Type: circle * real * real -> circle
*)
val move_circle =
           fn(Circle(x, y, r), x', y') =>
              Circle(x + x', y + y' ,r);
```

```
(*
 Signature: scale_circle(Circle(x, y, r), scale)
 Purpose: scale circ by scale
 Type: circle * real -> circle
*)
val scale_circle =
           fn(Circle(x, y, r), scale) =>
              Circle(x, y,r * scale);
```

Note

The expression Circle(_, _, r) in the function ***area_circle*** binds the variable *r* to the circle's radius.

Example 7

Version 1

```
val area_circle =
     fn(Circle(_,_, r) =>
           let
               val pi = 3.14
           in
               pi * square(r)
           end;
```

Version 2

```
val area_circle =
let
    val pi = 3.14
in
    fn(Circle(_,_, r)) =>
                    pi*square(r)
end;
```

What is the difference between these two versions?

6. Polymorphic and Recursive Data Types

Recursive type (Optional)

Recursive types create infinite sets of values. The value constructors of a recursive datatype accept parameters of the defined datatype.

Example 8

```
- datatype binTree = Null | Leaf | Node of binTree * binTree;
datatype binTree = Leaf | Node of binTree * binTree | Null
```

Running Example:

```
- Node(Node(Leaf,Leaf),Node(Leaf,Null));
val it = Node (Node (Leaf,Leaf),Node (Leaf,Null)) : binTree
```

Example 9 (Optional)

```
(*
  Signature: tree_size
  Purpose: Calculate the size (number of nodes) in a binary tree
  Type: binTree -> int
  Example: tree_size(Node(Empty,0,Node(Empty,1,Empty))) returns 2.
*)

val rec tree_size =
     fn Null => 0
     |Leaf   => 1
     | Node(lft, rht) => (1 + tree_size(lft) + tree_size(rht));
```

Running Example

```
 -tree_size(Node(Node(Leaf,Leaf),Node(Leaf,Node(Leaf,Null))));
val it = 8 : int
```

Recursive Polymorphic datatype

Adding polymorphic typing to a recursive datatype we get a type which is both recursive an polymorphic.

Example (optional): Sequence operations*:* *Accumulate_left* and *Accumulate_right* functions

Recall the function ***accumulate*** presented in Chapter 3, which combines the list elements using an operator *f* and initial value *e*. The function *Accumulate_right* is equivalent to the function *accumulat*e. It combines the list elements from right to left. We introduce a second version *Accumulate_Left* which combines the list elements from left to right.

Here is a reminder to both rules (as an example with + as the operation and 0 as the initial value)

Accumulate left
(((0+a) + b) + c)+d

Accumulate Right
a + (b + (c + (d + 0)))

```
- val rec accumulate_left  =
      fn(f, e, []) => e
       |(f,e,(head::tail)) => accumulate_left (f, f(head, e), tail);
val accumulate_left = fn : ('a * 'b -> 'b) * 'b * 'a list -> 'b
```

Running Examples:

```
- accumulate_left ((fn(x,y) => x - y), 0, [1, 2 ,3 , 4]);
val it = 2 : int
```

```
- accumulate_left ((fn(x,y) => x @ y), [], [[1,2,3], [4,5,6],
[7,8,9]]);
val it = [7,8,9,4,5,6,1,2,3] : int list
```

Note: @ is the *append* operator

Example 10: Tree

Trees form recursive polymorphic datatypes. In scheme, trees are represented as heterogeneous lists. However, heterogeneous list cannot be represented in ML using the list value constructor ::. The reason is that in list, once the type variable 'a is instantiated by a ML type value, the type of the list is determined. For example, trying to translate the following scheme list to an equivalence ML version gives an error

In Scheme:
```
(list 1 (list 2 3))
```

In ML
```
- [1, [2,3]];
stdIn:39.1-39.11 Error: operator and operand don't agree [literal]
  operator domain: int * int list
  operand:         int * int list list
  in expression:
    1 :: (2 :: 3 :: nil) :: nil
```

The solution is using datatypes.

We present here a second representation for a binary tree:
A binary tree has ***branch nodes***, each with a label and two ***subtrees***:

```
datatype 'a tree = Lf | Node of 'a * 'a tree * 'a tree;
```

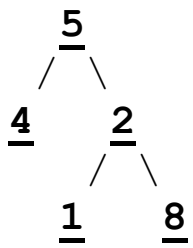
Here are some sample binary trees:

```
- val t1 = Lf;
val t1 = Lf : 'a tree

- val t2 = Node("abc", Lf, Lf);
val t2 = Node ("abc",Lf,Lf) : string tree

- val t3 = Node("+", Node("-", Lf, Lf), Lf);
val t3 = Node ("+",Node ("-",Lf,Lf),Lf) : string tree
```

Question: How would we represent the following tree? :



Answer (of course, white spaces are optional):

```
val binnum  =  Node (5, Node(4,Lf, Lf),
                        Node(2, Node(1, Lf, Lf),
                                Node(8, Lf, Lf)));
```

Example 11: The Function *treefold*

The function *treefold* for a binary tree is analogous to *accumulate_right*.
Given a tree, *treefold* replaces each leaf by some value *e* and each branch by the application of a 3-argument function *f*.
For example the expression:

`treefold((fn(a,b,c) =>  a + b + c), 0,` `binnum`**`)`**

returns 27

```
(* Type constructor: 'a tree
   Value constructors: Lf, Node.
   Data values of this type have the form:
 Node (5,Node (4,Lf,Node (7,Lf,Lf)),Node (2,Node (1,Lf,Lf),Node
(8,Lf,Lf)))
 *)
datatype 'a tree = Lf
                 | Node of 'a * 'a tree  * 'a tree;
```

```
(*
Signature: treefold(tree)
Purpose: Given a tree, treefold replaces each leaf by some value e and
each branch by the application of a 3-argument function f.
Type:
Example:
val binnum = Node (5,Node (4,Lf,Node (7,Lf,Lf)),Node (2,Node
(1,Lf,Lf),Node (8,Lf,Lf)))
treefold((fn(a,b,c) => a + b +c), 0, binnum) => (5 + (4 + 0 + ( 7 + 0 +
0)) + ( 2 + (1 + 0 + 0) + ( 8 + 0 + 0)))= 27
 *)
val rec treefold =
       fn(f, e, Lf) => e
        | (f, e, Node(node, left, right)) =>
             f(node, treefold(f , e, left), treefold(f, e, right));
```

```
datatype 'a tree = Node of 'a * 'a tree * 'a tree | Lf
val treefold = fn : ('a * 'b * 'b -> 'b) * 'b * 'a tree -> 'b
```

Part 2 – Lazy Lists

- Lazy lists, called ***sequences*** in ML, are lists whose elements are not explicitly computed.
    - We will use such lists to represent infinite sequences.
- When working with **eager** operational semantics, the "regular" list implementation computes all list elements before constructing the list. This is natural since list constructors are functions, and **due to applicative order**, arguments are evaluated **before** calling the function (e.g. the function 'list' in Scheme).
    - Therefore, lazy lists must be **defined as a new datatype**.
- Lazy lists are a special feature of **functional** programming, since their implementation is typically based upon creating procedures at run time.
- Although lazy lists are possibly infinite, we take care to construct lazy lists such that it is possible to reach **every finite location** in the list in finite time.
- An important advantage of lazy lists is that we only compute the **part of the sequence the we require**, without producing the entire sequence.

1. Basic Definitions

Example 1: Lazy list definition.

Recall that 'unit' is the type of the empty set (or the void type), e.g:.

```
  -fn () => 1;

val it = fn : unit -> int
```

We will define the type constructor **`seq`** for creating lazy-lists :

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

('a is the type parameter)

Values:

```
- Nil;   (* This is the empty lazy-list *)
val it = Nil : 'a seq;
```

Let's try:

```
-Cons(1, Nil);
```

ERROR – type mismatch…
The second argument should be a function (with no arguments) !

The correct way:

```
- val seq1 = Cons(1, fn()=>Nil);  (* this sequence contains Nil at it's tail *)
val seq1 = Cons (1,fn) : int seq   1::Nil
- val seq2 = Cons(2, fn()=>seq1); (* this sequence contains seq1 at it's tail *)
val seq2 = Cons (2,fn) : int seq   2::1::Nil
```

Note that these sequences lazy, since it is possible to evaluate only part of them (e.g. only '2' in seq2), but still not infinite.

Example 2: Head\Tail of a Sequence (Optional)

We will need to use exceptions to indicate that these functions are applied to empty sequences. (Remark: we will only raise the exceptions, but not "catch" them)

Adding exceptions:

```
- exception Empty;
exception Empty
- raise Empty;
uncaught exception Empty raised
```

Head definition: Selecting the head is very similar to non-lazy lists:

```
(* signature: head(seq)
   Purpose: get the first element of lazy sequence seq
   Type: 'a seq -> 'a
*)
- val head =
    fn Cons(h, _) => h
     | Nil => raise Empty;
val head = fn : 'a seq -> 'a
```

Tail definition: To select the tail's contents (and not just the empty function "wrap"), we need to apply it:

```
(* signature: tail(seq)
   Purpose: get the rest of the elements of lazy sequence seq
   Type: 'a seq -> 'a seq
*)
-   val tail =
   fn  Cons(_, tl) => tl()
                (* Here we apply the tail with no arguments *)
    |  Nil => raise Empty;
val tail = fn : 'a seq -> 'a seq
```

Examples (using seq1, seq2 from the previous section):

```
- head(seq1);
Val it = 1 : int
- tail(seq1);
val it = Nil : 'a seq
 - head(seq2);
Val it = 2 : int
- tail(seq2);  (* Note that this gives seq1's value *)
val it = Cons (1,fn) : int seq
```

Example 3: The First n Elements (Optional)

A useful function for examining sequences:

```
(*Signature:  take(seq,n)
  Purpose:  produce a list containing the first n elements of
            seq.
  Type:  'a seq * int -> 'a list
  Precondition: n >=0
 *)
- val rec take =
       fn (seq, 0) => [ ]
        | (Nil, n) => raise Subscript
        | (Cons(h,t), n) => h::take( t(), n-1);
val take = fn : 'a seq * int -> 'a list
```

We use this function extensively below, whenever we wish to examine the contents of a sequence we construct.

2. Basic Infinite Sequences

Example 4: Ones Sequence

Let's create the infinite sequence of ones: 1,1,1,1,1,…

Note 1: *Note that we don't need an argument - This is the first inifinte sequence we create which doesn't take an argument!*

```
(* Signature:  ones()
     Purpose:  produce a lazy sequence in which each element is the
number 1.
     Type:  unit -> int seq
*)
-val rec ones =
   fn () =>
     Cons (1, ones);
val ones = fn : unit -> int seq

take(ones(), 10);
val it = [1,1,1,1,1,1,1,1,1,1] : int list
```

Note 2: Note that 'ones' must be a procedure, or else we wouldn't be able to define it recursively.

3. **Processing Infinite Sequences**

Example 5: Adding sequences (Optional)

Suppose we wish to add to sequences. This could be done by:

1) Add the two heads to get the current head.
2) Continue recursively with both tails.

```
(* Signature:  add_seqs(seq1, seq2)
   Purpose:  return a seq which contains  elements resulting from
             the addition of same-location elements in seq1, seq2
  REMARK: In this function we rely on the pattern matcher to select the sequence's
             Elements (line 2). If we do so, we do not need to call the selectors
             head() and tail()  (as we did when defining the procedure 'take' in page 2)
             However, we need to apply  t (line 3) to get the tail of a sequence.
             This is, of course, only a matter of convention.
*)
1 - val rec add_seqs =
2   fn ( Cons(h, t), Cons (h', t') ) =>
3      Cons ( h+h', fn()=>add_seqs( t(), t'() ))
4    | (_, _) => Nil;


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

Question: ***why is the fn() in line 3 needed? What happens if we remove it (and just call add_seqs recursively) ?***

Example:

```
-add_seqs (ones(), ones());
```

val it = Cons (2,fn) : int seq

```
-take (add_seqs(ones(), ones()), 3);
```

val it = [2,2,2] : int list

Example 6: Even Integer (Optional)

To create an infinite sequence evens_from of even integers starting from n, we simply Cons n to evens_from of n+2.

Version 1:

```
(* Signature:  evens_from(n)
   Purpose:  produce a seq of even numbers.
   Type:  int -> int seq
   Precondition: n is even
 *)
- val rec evens_from =
      fn (n) => Cons(n, fn()=>evens_from(n+2));
val evens_from = fn : int -> int seq

 - val evens_from_4 = evens_from(4);
val evens_from_4 = Cons (4,fn) : int seq

-  take (evens_from_4, 3);
val it = [4,6,8] : int list
```

what would happen if we don't taking the precondition into account?

How to make an arithmetic sequence?

Example - Calculation steps:

```
take (evens_from_4, 3)
take (Cons (4,fn()=>evens_from(6)), 3)
4::take(evens_from(6), 2)
4:: take(Cons (6,fn()=>evens_from(8), 2)
4::6::take(evens_from(8), 1)
4::6::take(Cons (8,fn()=>evens_from(10), 1)
4::6::8::take(evens_from(10), 0)
4::6::8::[]
[4,6,8]
```

Note, that we may apply 'take' here with the second argument as large as we want, since this is an infinite sequence.

Version 2:

```
 (*  Pre-condition:  n is even *)
- val rec evens_from  =
   fn (n) => add_seqs(integers_from (n div 2),
                      integers_from (n div 2) );
val evens_from = fn : int -> int seq
```

`integers_from`: The infinite sequence of integers from K (shown in class)

4. Infinite-Sequence Operations

Example 7: Fibonacci Numbers

Version 1: (optional)

To define a sequence of (all) Fibonacci numbers, we use an auxiliary function fib:

```
- val rec fib =
    fn 0 => 0
     | 1 => 1
     | n => fib(n-1) + fib(n-2);
val  fib = fn : int -> int
```

```
 >val fibs_from =
           fn 0 => Cons(0, fn()=>fibs_from (1))
            | 1 => Cons(1, fn()=>fibs_from (2))
            | n => Cons(fib(n), fn()=>fibs_from(n+1))
 val fibs_from = fn : int -> int seq

>val fibs = fibs_from 0;
val it = Cons (0,fn) : int seq

> take(fibs, 12);
val it = [0,1,1,2,3,5,8,13,21,34,55,89] : int list
```

Note that the above definition is highly inefficient – we newly compute fib(n) for every n without using the information from previous ns.

Version 2 (not-optional):

A better version (yielding the same sequence):

```
(* Signature:  fibs()
   Purpose:  produce a seq of fib numbers.
   Type:  unit -> int seq
*)
- val fibs =
   let
     val rec fibs_help =
       fn(n, next) => Cons(n, (fn()=>fibs_help(next, n+next)) )
     in
       fibs_help(0, 1)
     end;
```

Example - Calculation steps:

```
take (fibs, 4)
take (Cons (0,fn()=>fibs_help(1,1)), 4)
0::take(fibs_help(1,1), 3)
0:: take(Cons (1,fn()=>fibs_help(1,2), 3)
0::1::take(fibs_help(1,2), 2)
0::1::take(Cons (1,fn()=>fibs_help(2,3), 2)
```

```
0::1::1::take(fibs_help(2,3), 1)
0::1::1::take(Cons (2,fn()=>fibs_help(3,5), 1)
0::1::1::2::take(fibs_help(3,5), 0)
0::1::1::2::[]
[0,1,1,2]

- take(fibs, 12);
```
val it = [0,1,1,2,3,5,8,13,21,34,55,89] : int list

Note that we do not use the function fib, or any other sequence operations. All the information we need is stored in the parameters, and computed iteratively.

Fibs sequence may be seen as an example of mapping over a sequence. We present a third version in the next section.

5. Infinite-Sequence Operations

Let's examine the map_seq procedure:

```
(* Signature:  map_seq(proc,seq)
   Purpose:  produce a seq in which each element is proc(k) where k is the corresponding element in seq.
   Type:  ('a -> 'b) * 'a seq -> 'b seq
   Example: map_seq ( fn(x)=>2*x, ints_from(1))
*)
- val rec map_seq =
    fn (proc, Cons(h,tl)) =>Cons( proc(h),
                                  fn()=>map_seq(proc, tl()));
```
val map_seq = fn : ('a -> 'b) * 'a seq -> 'b seq

Example 7: Fibs- Version 3 (optional)

```
- val fibs=map_seq( fib, ints_from(1) );
```

Example 8: Scaling a sequence (optional).

Scaling a sequence means multiplying all of its elements by a given factor.

```
(* Signature:  scale_seq(seq,factor)
   Purpose:  produce a seq in which each element is factor*k, where k is the an element in seq.
   Type:  int seq * int -> int seq
   Example: scale_seq(ints_from(1), 10)
*)

- val scale_seq =
     fn (seq, factor) => map_seq ( fn(x)=>factor*x, seq);
val scale_seq = fn : int seq * int -> int seq
```

```
- take( scale_seq(ints_from(1), 10), 10);
val it = [10,20,30,40,50,60,70,80,90,100] : int list
```

Example 9: Nested Sequence (not optional)

```
(* Signature: nested_seq(seq)
   Purpose: produce a seq in which each element is seq. the value in input seq initials the starting value of
            the corresponding result sequence.
   Type: int seq -> int seq seq
   Example:  take(nested_seq(ints_from(1)),3) => [Cons (1,fn),Cons (2,fn),Cons (3,fn)] : int
             seq list
*)
- val nested_seq =
     fn(seq) => map_seq ( fn(x)=>ints_from(x), seq);
val nested_seq = fn : 'a seq -> 'a seq seq
```

Illustration of result: [1,2,3,4,…] ➔ [[1,2,3,4,…],[2,3,4,5,…],[…],…]

For example, by using the function list_ref

```
(*TYPE: 'a list * int --> 'a *)
val rec list_ref =
    fn ( [], _) => raise Empty
     | ( a::li, 0) => a
     | ( a::li, n) => list_ref( li, n-1);
```

We can type:

```
val nest1 = nested_seq(ints_from(1));

val list2 = take( nest1 ,2);
val second_element = list_ref( list2, 1);

take(second_element, 5);
val it = [2,3,4,5,6] : int list
```

Example 10: The append function(Optional)

Regular lists append is defined by:

```
 -val rec append=
     fn ([], lst)=>
                lst
                | (h :: lst1, lst2) => h :: append(lst1,
                                                   lst2);

val append = fn : 'a list * 'a list -> 'a list
```

Trying to write an analogous seq_append yields:

```
 -val rec seq_append=
     fn (Nil, seq)          => seq
     | (Cons(h, tl), seq <= (
          Cons(h, (fn() => seq_append( tl(), seq))) ;
val seq_append = fn : 'a seq * 'a seq -> 'a seq
```

However, observing the elements of the appended list, we see that all elements of the first sequence come before the second sequence. What if the first list is already infinite? There is no way to reach the second list. So, this version DOES NOT satisfies the natural property of sequence functions: Every finite part of the output sequence depends on at most a finite part of the input.

Solution: Interleaving

When dealing with possibly infinite lists, append is replaced by an interleaving function that interleaves the elements of sequences in a way that guarantees that every element of the sequences is reached within finite time:

How do we combine 2 (or more) infinite sequences? By interleaving: taking one element from each at a time.

E.g. – if we want to combine:
1,1,1,1,… And 2,2,2,2,…
We wish to get: 1,2,1,2,…

```
(* Signature:  interleave(seq1,seq2)
   Purpose:  produce a seq that combines elements from seq1 and seq2.
   Type:  'a seq* 'a seq -> 'a seq
*)
- val rec interleave =
    fn (Nil, seq) => seq
     | (Cons(h, tl), seq) =>
         Cons(h, (fn()=>interleave(seq, tl() ) ) );
val interleave = fn : 'a seq * 'a seq -> 'a seq
```

Note how the argument 'seq' ,which was the first argument in the original call, is used as the second argument in the recursive call.

Assume twos is the sequence: 2,2,2,2,2…

```
take (interleave(ones,twos), 3)
take (Cons (1,fn()=>interleave(twos, ones)), 3)
```

```
1::take(interleave(twos, ones), 2)
1:: take(Cons (2,fn()=>interleave(ones, twos)), 2)
1::2::take(interleave(ones, twos), 1)
1::2::take(Cons (1,fn()=>interleave(twos, ones)), 1)
1::2::1::take(interleave(twos, ones), 0)
1::2::1::[]
[1,2,1]
```

EXAMPLE 11 (not optional):

Repeated

This useful function is used for generating sequences of the form: [x, f(x), f(f(x)), …. , f^n(x), …]

```
(* Signature:  repeated_seq(f,x)
   Purpose:  produce the seq x,f(x),f(f(x)),…f^n(x),…
   Type:  ('a -> 'a) * 'a -> 'a seq
*)
 - val rec repeated_seq =
      fn (f, x) => Cons(x, fn()=>repeated_seq(f, f(x)));
val repeated_seq = fn : ('a -> 'a) * 'a -> 'a seq
```

EXAMPLE 12:

The geometric series:

$a_0, a_0q, a_0q^2, \ldots, a_0q^n, \ldots$

```
- val geom_series =
     fn (a0, q) => repeated_seq (fn(x)=>q*x, a0);
val geom_series = fn : int * int -> int seq

- take(geom_series(10, 2), 5);
val it = [10,20,40,80,160] : int list
```

Explanation: the function used to obtaining a_{i+1} from a_i is simply a function which multiplies its argument by q (which is given as a parameter). The first element of the series is a0 (also a parameter), so it's the head of the sequence.

EXAMPLE 13: (optional)

Recall the definition of square roots with high order procedures from chapter 2. The idea was to generate a sequence of better guesses for the square root of *x* by applying over and over again the procedure that improves guesses:

```
(define (sqrt-improve guess x)
  (average guess (/ x guess)))
```

Here, we do the same, only we use an infinite sequence and the function *iterates*.

First, we define curried sqrt_improve (since *iterates* takes a function of one argument)

```
- val c_sqrt_improve =
    fn (x) =>
      fn (guess) => 0.5*(guess + x/guess);
val c_sqrt_improve = fn : real -> real -> real
```

Recall that x is the number for which we are seeking the square root – it would be given as an argument to sqrt_seq. Then, we apply c_sqrt_improve to x, and receive the single-parameter function we need (observe the 3rd line of code below):

```
- val sqrt_seq =

    fn (x, guess) =>

      repeated_seq(c_sqrt_improve(x), guess);


val sqrt_seq = fn : real * real -> real seq

- take(sqrt_seq(2.0, 5.0), 5);
val it = [5.0,2.7,1.72037037037,1.44145536818,1.41447098137] : real list
```