Chapter 14: Input/Output

1. Textual Input/Output
    a. The text I/O primitives are based on the notions of an input stream and output stream, which are values of type instream and outstream
    b. Input stream is unbounded sequence of characters arising from some source (disk file, interactive user, or another program, etc.)
    c. Input stream can be thought of as a buffer containing zero or more characters that have already been read from the source, together with a means of requesting more input from the source should the program require it
    d. Output stream is an unbounded sequence of characters leading to some sink (data file, etc.)
    e. Any sink for characters can be attached to an output stream
    f. It can be thought of as a buffer containing zero or more characters that have been produced by the program but have yet to be flushed to the sink
    g. Each program comes with one input stream (stdIn) and output stream (stdOut)
    h. These are ordinarily connected to user's keyboard and screen, and are used for performing simple text I/O in a program
    i. Output stream stdErr is also pre-defined and is used for error reporting that is connected to the user's screen
    j. Textual input and output are performed on streams using variety of primitives
    k. Simplest are inputLine and print
        i. To read a line of input from a stream, use the function inputLine of type instream -> string (yields that line as string whose last character is line terminator)
        ii. If the source is exhausted, return empty string
        iii. To write a line to stdOut, use the function print of type string -> unit
        iv. To write to a specific stream, use the function output of type outstream * string -> unit, which writes the given string to the specified output stream

Chapter 15: Lazy Data Structures

a. In ML, all variables are bound by value, which has several consequences
    i. Right-hand side of val binding is evaluated before the binding is effected. If the right-hand side has no value, the val binding does not take effect
    ii. In a function application the argument is evaluated before being passed to the function by binding that value to the parameter of the function. If the argument does not have a value, then neither does the application
    iii. The arguments to a value constructors are evaluated before the constructed value is created

b. ML → eager language since it evaluates bindings of variables, regardless of whether that variable is every needed to complete execution
c. Bind variables by name, meaning that binding of a variable is unevaluated expression, known as computation or suspension or thunk
    i. The right-hand side of val binding is not evaluated before binding is effected. The variable is bound to a computation, not a value
    ii. In a function application the argument is passed to the function in unevaluated form by binding it directly to the parameter of the function
    iii. The arguments to value constructor are left unevaluatd when the constructed value is created
d. Lazy language → adopt the by-name discipline (bindings of variables are only evaluated when their values are required by primitive operation)
e. Laziness is based on refinement of by-name principle (variables are bound to unevaluated computations and are evaluated only as often as the value of that variable's binding is required to complete the computation)
f. EX) x + x → x is evaluated twice
g. By-need principle (binding of variable is evaluated at most once and re-evaluation of the same computation is avoided by memoization. Once computation is evaluated, its values are saved for future references)
h. ML has lazy data types that allow us to treat unevaluated computations as values of such type (we can use it when it's appropriate, and ignore it when it is not)

1. Lazy Data Types
    a. SML provides general mechanism for introducing lazy data types by simply attaching keyword lazy to ordinary datatype declaration
    b. EX) datatype lazy 'a stream = Cons of 'a * 'a stream
    c. This type has no base case (when it is lazy, such case is not very useful)
    d. By doing so, it specifies that the values of type typ stream are computations of values of the form Cons (val, val') → computation is not evaluated until we examine it but is revealed as consisting of an element val together with another of the same type when we examine it
    e. Values of type typ stream are created using val rec lazy declaration
    val rec lazy ones = Cons(1, ones)
    f. Keyword lazy indicates that we are binding ones to computation rather than value
    Keyword rec indicates that the computation is recursive
    g. Pattern matching forces evaluation of a computation to the extent required by the pattern
2. Lazy Function Definitions
    a. Combination of lazy function definitions and decomposition by pattern matching are core mechanisms required to support lazy evaluation

b. The tail function fun stl (Cons (_, s)) = s can be thought of as the stream created by stl should also be suspended until its value is required

c. Therefore, we state
fun lazy lstl (Cons(_, s)) = s → does not immediately perform pattern matching on its argument but rather sets up a stream computation that, when forced, forces argument and extracts tail of the stream

d. Behavior of two forms of tail function

```
val rec lazy s = (print "."; Cons (1, s))
val _ = stl s     (* prints "." *)
val _ = stl s     (* silent *)
val rec lazy s = (print "."; Cons (1, s));
val _ = lstl s    (* silent *)
val _ = stl s     (* prints "." *)
```

e. stl evaluates its argument when applied, "." is printed when is first called but not if it is called again

f. lstl only sets up computation, its argument is not evaluated when it is called, but only when its result is evaluated

3. Programming with Streams

a. Function smap that applies function to every element of a stream, giving another stream

b. fun smap f = (type ('a -> 'b) -> 'a stream -> 'b stream)

```
        let
                fun lazy loop (Cons(x,s)) =
                        Cons(f x, loop s)
        in
                loop
        end
```

c. This loop is lazy function to ensure that it merely sets up stream computation, rather than evaluating its argument when it is called

d. If lazy is dropped, application of smap to a function and a stream would immediately force computation of the head element of stream

e. Use of smap: infinite stream of natural numbers

```
val one_plus = smap (fn n => n+1)
val rec lazy nats = Cons (0, one_plus nats)
```

f. fun sfilter pred (function that filters out all elements of a stream that do not satisfy predicate)
(type) ('a -> bool) -> 'a stream -> 'a stream)

```
fun sfilter pred =
    let
        fun lazy loop (Cons (x, s)) =
            if pred x then
               Cons (x, loop s)
            else
               loop s
    in
        loop
    end
```

g. Use sfilter to define function sieve that, when applied to a stream of numbers, retains only numbers that are not divisible by preceding number in the stream

```
fun m mod n = m - n * (m div n)
fun divides m n = n mod m = 0
fun lazy sieve (Cons (x, s)) =
    Cons (x, sieve (sfilter (not o (divides x)) s))
```

h. Define infinite stream of primes by applying sieve to the natural numbers greater than or equal to 2
val nats2 = stl(stl nats)
val primes = sieve nats2

i. Inspect values of stream

```
fun take 0 _ = nil
  | take n (Cons (x, s)) = x :: take (n-1) s
```

j. Effects of memoization

```
val rec lazy s = Cons ((print "."; 1), s)
val Cons (h, _) = s;
(* prints ".", binds h to 1 *)
val Cons (h, _) = s;
(* silent, binds h to 1 *)
```