```
(* CS 251: Delayed evaluation examples *)

fun fact n =
    if n=0 then 1 else n * (fact (n-1))

fun iffy x y z =
    if x then y else z

(* What goes wrong? *)
fun facty n =
    iffy (n=0) 1 (n * (facty (n-2)))

(* y and z are thunks -- call to evaluate *)
fun ifok x y z =
    if x then y () else z ()

fun factok n =
    ifok (n=0) (fn () => 1) (fn () => n * (factok (n-1)))

(* Thunking can help or hurt performance.
   This is a silly addition function that purposely runs slowly for
   demonstration purposes *)
fun slowadd x y =
    let fun slowid a b =
            if b=0 then a else slowid a (b-1)
    in
        (slowid x 50000000) + y
    end

(* mult x ythunk
   multiplies x and result of ythunk, calling ythunk x times,
   assumes x >= 0 *)
fun mult 0 ythunk = 0
  | mult 1 ythunk = ythunk ()
  | mult x ythunk = (ythunk ()) + (mult (x-1) ythunk)

(* these calls: great for 0, okay for 1, bad for > 1
val x = mult 0 (fn () => slowadd 3 4)
val y = mult 1 (fn () => slowadd 3 4)
val z = mult 2 (fn () => slowadd 3 4)
*)

(* these calls: okay for all
val x = mult 0 let val x = slowadd 3 4 in (fn () => x) end
val y = mult 1 let val x = slowadd 3 4 in (fn () => x) end
val z = mult 2 let val x = slowadd 3 4 in (fn () => x) end
*)


(* Explicit laziness with promises. *)
signature PROMISE =
sig
    (* Type of promises to produce an 'a. *)
    type 'a t
    (* Make a promise for a thunk. *)
    val delay : (unit -> 'a) -> 'a t
    (* If promise not yet forced, call thunk and save.
       Return saved thunk result. *)
    val force : 'a t -> 'a
end

structure Promise :> PROMISE =
struct

(* Before a promise has been forced, it is just a thunk.  After it has
   been forced, it is a value. *)
datatype 'a promise = Thunk of unit -> 'a
                    | Value of 'a

(* Hide limited mutation inside ADT. *)
type 'a t = 'a promise ref

(* Wrap the thunk to make a promise. *)
fun delay th = ref (Thunk th)

(* If the promise is already a value, return it.
   Otherwise, call the thunk and save and return its result. *)
fun force p =
  case !p of
      Value v => v
    | Thunk th => let val v = th ()
                      val _ = p := Value v
                  in v end
end

(* these calls: great for 0, okay for 1, okay for > 1 *)
val x = mult 0 let val p = Promise.delay (fn () => slowadd 3 4)
               in (fn () => Promise.force p) end
val y = mult 1 let val p = Promise.delay (fn () => slowadd 3 4)
               in (fn () => Promise.force p) end
val z = mult 2 let val p = Promise.delay (fn () => slowadd 3 4)
               in (fn () => Promise.force p) end
```

```
(* CS 251: Stream examples *)

exception Unimplemented

signature STREAM =
sig
    (* A stream is a thunk that, when called, produces a pair of
       element and remaining stream. *)
    datatype 'a scons = Scons of 'a * (unit -> 'a scons)
    type 'a t = unit -> 'a scons
    type 'a stream = 'a t

    (* Make a new stream where the first element is the given element,
       and each element's successor is determined by applying the
       given function to the current element. Calling this function
       does not expand any stream. *)
    val smake   : ('a -> 'a) -> 'a -> 'a stream

    (* Take the first n elements from a stream, returning a pair of:
         - a list of those elements, in stream order; and
         - the rest of the stream (as a stream).
       Calling this function expands exactly n elements of the given
       stream. *)
    val stake   : 'a stream -> int -> 'a list * 'a stream

    (* Make a new stream where each element is the result of applying
       the given function to the corresponding element of the given
       stream.  Calling this function does not expand any stream. *)
    val smap    : ('a -> 'b) -> 'a stream -> 'b stream

    (* Make a new stream containing only those elemesnts of the given
       stream (in order) for which the given function returns true.
       Calling this function does not expand any stream. *)
    val sfilter : ('a -> bool) -> 'a stream -> 'a stream

    (* Make a new stream where each element is a pair of elements, one
       from each of the given lists, cycling through the elements of
       each list in order.  Calling this function does not expand any
       stream. *)
    val scycle  : 'a list -> 'b list -> ('a * 'b) stream
end

structure Stream :> STREAM =
struct
(* Stream representation. *)
datatype 'a scons = Scons of 'a * (unit -> 'a scons)
type 'a t = unit -> 'a scons
type 'a stream = 'a t

(* Make a new stream where the first element is init, and each
   element's successor is determined by applying succ to the current
   element.  Calling this function does not expand any stream. *)
fun smake succ init =
  let fun f x = Scons (x, fn () => f (succ x))
  in fn () => f init end

(* Take the first n elements from a stream, returning a pair of:
     - a list of those elements, in stream order; and
     - the rest of the stream (as a stream).
   Calling this function expands exactly n elements of the given
   stream. *)
fun stake stream n = raise Unimplemented

(* Make a new stream where each element is the result of applying f to
   the corresponding element of stream.  Calling this function does
   not expand any stream. *)
fun smap f stream = raise Unimplemented

(* Make a new stream containing only those elemesnts of stream (in
   order) for which f returns true.  Calling this function does not
   expand any stream. *)
fun sfilter f stream = raise Unimplemented

(* Make a new stream where each element is a pair of elements, one
   from each of the lists xs and ys, cycling through the elements of
   each list in order.  Calling this function does not expand any
   stream. *)
fun scycle xs ys = raise Unimplemented
end

open Stream

(* A stream of ones. *)
fun ones () = Scons (1,ones)
(* Alternatively *)
val rec ones = fn x => Scons (1, ones)

(* A stream of the natural numbers from 0. *)
val nats =
    let fun f x = Scons (x, fn () => f (x+1))
    in fn () => f 0 end

(* A stream of powers of two from 1. *)
val powers2 =
    let fun f x = Scons (x, fn () => f (x * 2))
    in fn () => f 1 end

(* Build streams using smake *)
val nats = smake (fn x => x + 1) 0
val powers2 = smake (fn x => x * 2) 2

(* Count the stream elements until f returns true on one of them. *)
fun firstindex f stream =
    let fun help stream ans =
            let val Scons (v,s) = stream ()
            in
                if f v then ans else help s (ans + 1)
            end
    in
        help stream 0
    end

val four = firstindex (fn x => x=16) powers2
```

```
(* Memoization *)

(* O(2^n) via naturally recursive algorithm. *)
fun fibexp 0 = 1
  | fibexp 1 = 1
  | fibexp n = fibexp (n-2) + fibexp (n-1)

(* O(n) via double-accumulator tail-recursive algorithm. *)
fun fibn 0 = 1
  | fibn 1 = 1
  | fibn x =
    let fun f (acc1, acc2, y) =
            if y=x
            then acc1 + acc2
            else f (acc1 + acc2, acc1, y + 1)
    in f (1,1,3) end

(* Association lists. *)
fun assoc x [] = NONE
  | assoc x ((k,v)::rest) =
    if k=x then SOME v else assoc x rest

(* Memoize any function, but INEFFICIENTLY -- only top-level calls
   (not recursive calls) use the memo table.  Recursive calls do
   not. *)
fun memotop f =
  let
      (* Mutable reference to memo table, hidden in closure.  We will
         ignore the fact that association list lookup is an O(|list|)
         operation.  We should replace association lists with hash
         tables or other structures with faster lookups, but our focus
         is not on the data structure.
       *)
      val mem = ref []
  in
      fn x =>
         case assoc x (!mem) of
             SOME y => y
           | NONE => let val y = f x
                         val _ = mem := ((x,y)::(!mem))
                     in y end
  end
val fibtop = memotop fibexp

(* Memoized fib. *)
val fibm =
    let
        (* Reference to a memo table, available in closure for fib,
           but invisible elsewhere. *)
        val memo = ref []
        fun fib x =
          case assoc x (!memo) of
              SOME y => y
            | NONE => let val y = (case x of
                                     0 => 1
                                   | 1 => 1
                                   | n => fib (n-2) + fib (n-1))
                          val _ = memo := ((x,y)::(!memo))
                      in y end
    in fib end


(* OPTIONAL beyond here, but really cool!

   The above fibm implementation is quite efficient (assume we replace
   association lists with a hash table), but it is a bit ugly.  It
   mixes up what we are computing with how we are doing it
   efficiently.

   With a relatively non-intrusive change to the function we define,
   we can apply memoization orthogonally to fib itself and then
   compose them.  We call this form "open recursion."  It has close
   ties to the way that method dispatch is defined in object-oriented
   languages.

   fibopen takes 2 arguments instead of 1.  Its second argument is the
   usual n.  Its first argument is a function to call in order to make
   recursive calls.  This adds a little extra baggage, but much less
   than in fibm, and, with a well-chosen names, it is fairly clear.
*)
fun fibopen fib 0 = 1
  | fibopen fib 1 = 1
  | fibopen fib n = fib (n-2) + fib (n-1)

(* fix takes a function in open recursive form and makes a closed
   recursive function from it.  It implements recursion via a
   fixpoint.

   Does it look familiar?  Think back to recursion in the lambda
   calculus.  It is tempting to rewrite it to remove the x argument
   and its use (remove the function wrapping) and take advantage of
   currying and partial application, but something unfortunate
   happens.  What?  Why?  (Hint: it works fine in Haskell.)
 *)
fun fix f x = f (fix f) x

(* fibfix implements fib in O(2^n), equivalent to the naturally
   recursive implementation. *)
val fibfix = fix fibopen

(* Make a memoizer function in open recursive form. *)
fun make_memo () =
  let val mem = ref []
      (* In open recursive form: *)
      fun memf f x =
        case assoc x (!mem) of
            SOME v => v
          | NONE => let val v = f x
                        val _ = mem := ((x,v)::(!mem))
                    in v end
  in memf end

(* Memoized fib implementation equivalent to fibm. *)
val fibmemo = fix (make_memo () o fibopen)

(* Or, by reimplementing fix within the memoizer: *)
fun memoize f = (* diff: f as arg to memo construction *)
  let val mem = ref []
      fun memf x = (* diff: capture f in closure *)
        case assoc x (!mem) of
```

```
            SOME v => v
          | NONE => let val v = f memf x (* diff: explicitly fix *)
                        val _ = mem := ((x,v)::(!mem))
                    in v end
  in f memf end

val fibmemo' = memoize fibopen


(* In fact, this form supports arbitrary "shim" functions between
   recursive levels.  Neither has to know about the other at
   definition time.  They are combined later via application. *)
fun log name atos rtos f =
  let fun wrap indent x =
        let val _ = print (indent ^ name ^ " " ^ atos x ^ "\n")
            val v = f (wrap ("  " ^ indent)) x
            val _ = print (indent ^ "=> " ^ rtos v ^ "\n")
        in v end
  in wrap "" end

val fiblog = log "fib" Int.toString Int.toString fibopen

(* We need a bit more machinery to make log fully composable like
   fibopen and the memoizers created by make_memo.  It is possible,
   interesting and even pretty clean, but we will stop here.  If you
   are curious about this come chat!  Check out:
   https://www.cs.utexas.edu/~wcook/Drafts/2006/MemoMixins.pdf.
   Sections 1 - 2.2 should be accessible to a motivated 251-level
   reader.  Reading beyond will require some extra background.  As
   always, come chat if you are curious.  Related topics could make a
   great final project...
 *)
```