fun compose \(f, g\) = \(\lambda x \Rightarrow f (g x)\)

fun sqrt_of_abs i = Math.sqrt(Real.fromInt (abs i))

val sum = fold (\(x, y\) \(\Rightarrow x + y\)) 0

fun exists predicate xs = 
   (xs \(\Rightarrow\) predicate x)

fun range i j = 
   if i > j 
   then []
   else i :: range (i+1) j

fun pairWithOne xs = List.map (fn x => (x,1)) xs

fun curry f x y = f (x,y)

val sum = fold (\(\lambda (x,y) \Rightarrow x + y\)) 0

fun is_nonnegative inferior x = sorted3 0 0 x

fun sum inferior xs = fold (\(\lambda (x,y) \Rightarrow x + y\)) 0 xs

fun is_nonnegative inferior x = sorted3 0 0 x

fun incrementAll = List.map (\(\lambda x \Rightarrow x + 1\))

fun removeZeros = List.filter (\(\lambda x \Rightarrow x \neq 0\))

fun sorted3_tupled (x,y,z) = z

val is_nonnegative = sorted3 0 0

Winter 2018
CSE 341: Programming Languages
val xs = countup 7

(* callbacks *)

(* these two bindings would be internal (private) to the library *)
val chs : (int -> unit) list ref = ref []
fun onEvent i =
  let fun loop fs =
    case fs of
      [] ⇒ ()
    | f::fs' ⇒ (f i; loop fs')
  in
    loop (!cbs)
  end

(* clients call only this function (public interface to the library) *)
fun onKeyEvent f = cbs := f::(!cbs)

(* some clients where closures are essential  notice different environments use bindings of different types *)
val timesPressed = ref 0
val _ = onKeyEvent (λ _ ⇒ timesPressed := (!timesPressed) + 1)

fun printIfPressed i =
  onKeyEvent (λ j ⇒
    if i=j
    then
      print (*you pressed" ^ Int.toString i ^ ")
      else ()
  )

(*val _ = printIfPressed 4
val _ = printIfPressed 11
val _ = printIfPressed 23
val _ = printIfPressed 4*)

(*setADT via closures ***********)

(* a set of ints with three operations *)
(* this interface is immutable -- insert returns a new set -- but we could
  also have implemented a mutable version using ML’s references *)
(* Note: a 1-constructor datatype is an SML trick for recursive types *)
datatype set = S of { insert : int -> set,
  member : int -> bool,
  size   : unit -> int }

(* implementation of sets: this is the fancy stuff, but clients using
this abstraction do not need to understand it *)
val empty_set =
  let
    fun make_set xs = (* xs is a "private field in result *)
      let (* contains a "private method in result *)
        fun contains i = List.exists (λ j ⇒ i=j) xs
      in
        S { insert = λ i ⇒ if contains i
          then make_set xs
          else make_set (i::xs),
        member = contains,
        size   = λ () ⇒ length xs
      }
    in
      make_set []
  end

(* example client *)
fun use_sets () =
  let
    val S s1 = empty_set
    val S s2 = (#insert s1) 34
    val S s3 = (#insert s2) 34
    val S s4 = #insert s3 19
  in
    if (#member s4) 42
    then 99
    else if (#member s4) 19
      then 17 + (#size s3) ()
    else 0
  end