(* Programming Languages, Dan Grossman, CSE341 *)

(* Lecture 9: Function-Closure Idioms *)

fun compose (f,g) = λ x ⇒ f (g x)

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

val is_nonnegative = sorted3 0 0

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

fun backup1 (f,g) =
  x
  λ
  ⇒
  case
  f x
  of
  NONE ⇒ g x | SOME y ⇒ y

fun backup2 (f,g) =
  x
  λ
  ⇒
  handle _ ⇒ g x

(* tells the parser !> is a function that appears between its two arguments *)

infix !> (* operator more commonly written |>, but that confuses the current version of SML Mode for Emacs, leading to bad editing and formatting *)

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

fun other_curry1 f = λ x ⇒ λ y ⇒ f y x

fun other_curry2 f x y = f y x

(* example *)

(* tupled but we wish it were curried *)

(* if a curried function is applied to "too few" arguments, that just returns a closure, which is often useful -- a powerful idiom (no new semantics) *)

Winter 2018 1/2
CSE 341: Programming Languages
val countup = curry range 1
val xs = countup 7

(* callbacks *)

(* these two bindings would be internal (private) to the library *)
val cbs : (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

(* 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 ^ "\n")
  else ())

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

(*************** likely optional below here: ADT 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
        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 }
      end
    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