(* lexical scope examples *)
(* first example *)

```sml
val x = 1
(* x ==> 1 *)

fun f y = x + y
(* f ==> func which increments arg by 1 *)

val x = 2
val y = 3
val z = f (x + y)
(* second example *)

val x = 1 (* irrelevant *)
fun f y =
  let
    val x = y + 1
  in
    λ z ⇒
    x + y + z (* (y + 1) + y + z = 2y + 1 + z *)
  end
val x = 3
val g = f 4 (* always adds 9 to its argument. always *)
val y = 5
val z = g 6
(* why lexical scope *)

fun f1 y =
  let
    val x = y + 1
  in
    λ z ⇒
    x + y + z
  end
fun f2 y =
  let
    val q = y + 1
  in
    λ z ⇒
    q + y + z
  end
val x = 17 (* irrelevant *)
val a1 = (f1 7) 4
val a2 = (f2 7) 4
(* f3 and f4 are always the same, no matter what argument is passed in *)

fun f3 g =
  let
    val x = 3 (* irrelevant *)
  in
    g 2
  end
fun f4 g =
  g 2
val x = 17
val a3 = f3 (λ y ⇒ x + y)
val a4 = f3 (λ y ⇒ 17 + y)
(* under dynamic scope, the call "g 6" below would try to add a string
(from looking up x) and would have an unbound variable (looking up y),
even though f1 type-checked with type int -> (int -> int) *)
val x = "hi"
val g = f1 7
val z = g 4
(* Being able to pass closures that have free variables (private data)
makes higher-order functions /much/ more useful *)
fun filter (f,xs) =
case xs
```

(* third example *)

```sml
fun f g =
  let
    val x = 3
  in
    g 2
  end
val x = 4
fun h y = x + y (* always adds 4 to its argument. always *)
val z = f h (* 6 *)
```

(* f1 and f2 are always the same, no matter where the result is used *)

```sml
fun f1 y =
  let
    val x = y + 1
  in
    λ z ⇒
    x + y + z
  end
fun f2 y =
  let
    val q = y + 1
  in
    λ z ⇒
    q + y + z
  end
val x = 17 (* irrelevant *)
val a1 = (f1 7) 4
val a2 = (f2 7) 4
```

(* f3 and f4 are always the same, no matter where the result is used *)

```sml
fun f3 g =
  let
    val x = 3 (* irrelevant *)
  in
    g 2
  end
fun f4 g =
  g 2
val x = 17
val a3 = f3 (λ y ⇒ x + y)
val a4 = f3 (λ y ⇒ 17 + y)
(* under dynamic scope, the call "g 6" below would try to add a string
(from looking up x) and would have an unbound variable (looking up y),
even though f1 type-checked with type int -> (int -> int) *)
val x = "hi"
val g = f1 7
val z = g 4
(* Being able to pass closures that have free variables (private data)
makes higher-order functions /much/ more useful *)
fun filter (f,xs) =
case xs
```
fun filter(f,xs') =  
  if f x  
  then x::(filter(f,xs'))  
  else filter(f,xs')

(* curried < *)
fun greaterThanX x = λ y ⇒ y > x
(* filter negative elements out of xs *)
fun noNegatives xs = filter(greaterThanX -1, xs)
(* keep all list elements greater than n *)
fun allGreater (xs,n) = filter(λ x ⇒ x > n, xs)
fun allShorterThan1 (xs,s) =  
  filter(λ x ⇒ String.size x < (print "!"); String.size s), xs)
(* function bodies are evaluated when function is called, in an environment where function was defined -- with expressions already bound to values, i.e., results of computations *)
fun allShorterThan2 (xs,s) =  
  let
    val i = (print "!"); String.size s
  in
    filter(λ x ⇒ String.size x < i, xs)
  end

(* Another hall-of-fame higher-order function *)
(* note this is "fold left" if order matters can also do "fold right" *)
fun fold (f,acc,xs) =  
  case xs of
    [] ⇒ acc
    | x::xs' ⇒ fold(f,acc,x),xs')

(* examples not using private data *)
fun f5 xs = fold((λ (x,y) ⇒ x+y), 0, xs)
fun f5' xs = fold(op+, 0, xs)
fun f5'' xs = fold(op-, 0, xs)

fun f6 xs = fold((λ (x,y) ⇒ x ∧ y ≥ 0), true, xs)

(* examples using private data *)
fun f7 (xs,lo,hi) =  
  fold((λ (x,y) ⇒ x + (if y ≥ lo ∧ y ≤ hi then 1 else 0)), 0, xs)
fun f8 (xs,s) =  
  let
    val i = String.size s
  in
    fold((λ (x,y) ⇒ x ∧ String.size y < i), true, xs)
  end
fun f9 (g,xs) =  
  fold((λ(x,y) ⇒ x ∧ g y), true, xs)