compose \( (f, g) = \lambda x \mapsto f (g x) \)

\[
\begin{align*}
\text{fun compose} \ (f, g) &= \lambda x \mapsto f \ (g x) \\
\text{fun sqrt_of_abs} \ i &= \text{Math.sqrt} (\text{Real.fromInt} \ (\text{abs} \ i)) \\
\text{fun sqrt_of_abs} \ i &= (\text{Math.sqrt} \ast \text{Real.fromInt} \ast \text{abs}) \ i \\
\text{val sqrt_of_abs} &= \text{Math.sqrt} \ast \text{Real.fromInt} \ast \text{abs}
\end{align*}
\]

(* 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 *)

(* definition of the pipeline operator *)

fun \( x \mapsto f = f \ x \)

fun sqrt_of_abs \( i = (\text{Math.sqrt} \ast \text{Real.fromInt} \ast \text{abs}) \ i \)

(* old way to get the effect of multiple arguments *)

fun sorted3_tupled \((x,y,z) = z \geq y \land y \geq x \)

val t1 = sorted3_tupled \((7,9,11)\)

(* new way: currying *)

val sorted3 = \( \lambda x \mapsto \lambda y \mapsto \lambda z \mapsto z \geq y \land y \geq x \)

val t2 = \( (\text{sorted3} \ 7 \ 9 \ 11) \ 11 \)

(* syntactic sugar for calling curried functions: optional parentheses *)

val t3 = \( \text{sorted3} \ 7 \ 9 \ 11 \)

(* syntactic sugar for defining curried functions: space between arguments *)

fun sorted3_nicer \( x \ y \ z = z \geq y \land y \geq x \)

val t4 = \( \text{sorted3_nicer} \ 7 \ 9 \ 11 \)

(* more calls that work: *)

val t5 = \( (\text{sorted3_nicer} \ 7 \ 9 \ 11) \ 11 \)

(* calls that do not work: cannot mix tupling and currying *)

(* val coz = \{(sorted3_tupled 7) \ 9 \ 11\} *)

(* val coz2 = \( \text{sorted3_tupled} \ 7 \ 9 \ 11 \) *)

(* val coz3 = \( \text{sorted3} \ (7,9,11) \) *)

(* val coz4 = \( \text{sorted3_nicer} \ (7,9,11) \) *)

(* a more useful example *)

fun fold \( f \ acc \ x s = (* means fun fold \( f = fn acc \Rightarrow fn x s \Rightarrow *\)\)

\[
\begin{align*}
\text{case} \ x s \ \text{of} \ &\ |
\end{align*}
\]

(* Note: foldl in the ML standard library is very similar, but the two arguments for the function \( f \) are in the opposite order. The order is, naturally, a matter of taste. *)

(* a call to curried fold: will improve with partial application next *)

fun sum \( xs = \text{fold} \ (\lambda \ (x,y) \mapsto x+y) \ 0 \ xs \)

(* 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
(* no problem *)
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 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 ⇒ i=j
    then print (*you pressed* ^ Int.toString i ^ "a")
    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

Winter 2018  
CSE 341: Programming Languages