```
(* Exercise 1 *)
(* Write a function il2rl of type int list -> real list that takes a list of integers and returns a list of the same numbers converted to type real. *)
fun il2rl lst = map Real.fromInt lst;

(* Exercise 2 *)
(* Write a function ordlist of type char list -> int list that takes a list of characters and returns the list of the integer codes of those characters. *)
fun ordlist lst = map Char.ord lst;

(* Exercise 3 *)
(* Write a function squarelist of type int list -> int list that takes a list of integers and returns the list of the squares of those integers. *)
fun squarelist lst = map (fn x => x * x) lst;

(* Exercise 4 *)
(* Write a function multpairs of type (int * int) list -> int that takes a pairs of integers and returns a list of the products of each pair. *)
fun multpairs lst = map (fn (a, b) => a * b) lst;

(* Exercise 5 *)
(* Write a function inclist of type int list -> int -> int list that takes a list of integers and an integer increment, and returns the same list of integers but with the inte
fun inclist lst n = map (fn x => x + n) lst;

(* Exercise 6 *)
(* Write a function sqsum of type int list -> int that takes a list of integers and returns the sum of the squares of those integers. *)
fun sqsum lst = foldr (op +) 0 (map (fn x => x * x) lst);

(* Exercise 7 *)
(* Write a function bor of type bool list -> bool that takes a list of boolean values and returns the logical OR of all of them. If the list is empty, your function should ret
fun bor lst = foldr (fn (a, b) => a orelse b) false lst;

(* Exercise 8 *)
(* Write a function band of type bool list -> bool that takes a list of boolean values and returns the logical AND of all of them. If the list is empty, your function should r
fun band lst = foldr (fn (a, b) => a andalso b) true lst;

(* Exercise 9 *)
(* Write a function bxor of type bool list -> bool that takes a list of boolean values and returns the logical exclusive OR of all of them. If the list is empty, your function
fun bxor lst = foldr (fn (a, b) => if a = b then false else true) false lst;

(* Exercise 10 *)
(* Write a function dupList of type 'a list -> 'a list whose output list is the same as the input list, but with each element of the input list repeated twice in a row. *)
fun dupList lst = foldr (fn (a, b) => a::a::b) [] lst;

```

```
(* Exercise 11 *)
(* Write a function mylength of type 'a list -> int that returns the length of a list. *)
fun mylength lst = foldr (fn (a, b) => b + 1) 0 lst;

(* Exercise 12 *)
(* Write a function il2absrl of type int list -> real list that takes a list of integers and returns a list containing the absolute values of those integers, converted to rea
fun il2absrl lst = map Real.fromInt (map Int.abs lst);

(* Exercise 13 *)
(* Write a function truecount of type bool list -> int that takes a list of boolean values and returns the number of trues in the list. *)
fun truecount lst = foldr (fn (a, b) => if a then b + 1 else b) 0 lst;

(* Exercise 14 *)
(* Write a function maxpairs of type (int * int) list -> int list that takes a list of pairs of integers and returns the list of the max elements from each pair. *)
fun maxpairs lst = map (fn (a, b) => if a > b then a else b) lst;

(* Exercise 15 *)
(* Write a function implode that works just like the predefined implode. *)
fun implode lst = foldr (fn (a, b) => (Char.toString a) ^ b) "" lst;

(* Exercise 16 *)
(* Write a function lconcat of type 'a list list -> 'a list that takes a list of lists as input and returns the list formed by appending the input lists together in order. *)
fun lconcat lst = foldr (fn (a, b) => a @ b) [] lst;

(* Exercise 17 *)
(* Write a function max of type int list -> int that returns the largest element of a list of integers. *)
fun max lst = foldr (fn (a, b) => Int.max (a, b)) (hd lst) lst;

(* Exercise 18 *)
(* Write a function min of type int list -> int that returns the smallest element of a list of integers. *)
fun min lst = foldr (fn (a, b) => Int.min (a, b)) (hd lst) lst;

(* Exercise 19 *)
(* Write a function member of type ''a * ''a list -> bool so that member(e,L) is true if and only if e is an element of list L. *)
fun member (n, lst) = foldr (fn (a, b) => if b orelse a = n then true else false) false lst;

(* Exercise 20 *)
(* Write a function append of type 'a list -> 'a list -> 'a list that takes two lists and returns the result of appending the second one onto the end of the first. *)
fun append lst1 lst2 = foldr (fn (a, b) => a::b) lst2 lst1;

(* Exercise 21 *)
(* Define a function less of type int * int list -> int list so that less(e,L) is a list of all the integers in L that are less than e. *)
fun less (n, lst) = foldr (fn (a, b) => if a < n then a::b else b) [] lst;

(* Exercise 22 *)
(* Write a function evens of type int list -> int list that takes a list of integers and returns the list of all the even elements from the original list (in the original ord
fun evens lst = foldr (fn (a, b) => if a mod 2 = 0 then a::b else b) [] lst;

(* Exercise 23 *)
(* Write a function convert of type ('a * 'b) list -> 'a list * 'b list, that converts a list of pairs into a pair of lists, preserving the order of the elements. *)
fun convert lst = foldr (fn ((a, b), (c, d)) => (a::c, b::d)) ([], []) lst;

(* Exercise 24 *)
(* Define a function mymap with the same type and behavior as map, but without using map. *)
fun mymap f lst = foldr (fn (a, b) => f a::b) [] lst;

(* Exercise 25 *)
(* Represent a polynomial using a list of its (real) coefficients, starting with the constant coefficient and going only as high as necessary. Write a function eval of type r
fun eval [] x = 0.0
|   eval (head::tail) x:real = head + eval (map (fn y => x * y) tail) x;

(* Exercise 26 *)
(* Define a function mymap2 with the same type and behavior as map. *)
fun mymap2 _ [] = []
|   mymap2 f (head::tail) = f head::mymap2 f tail;

(* Exercise 27 *)
(* Define a function myfoldr with the same type and behavior as foldr. *)
fun myfoldr _ x [] = x
|   myfoldr f x [head] = f (head, x)
|   myfoldr f x (head::tail) = f (head, (myfoldr f x tail));

(* Exercise 28 *)
(* Define a function myfoldl with the same type and behavior as foldl. *)
fun myfoldl _ x [] = x
|   myfoldl f x (head::tail) = myfoldl f (f (head, x)) tail;
```

Second Part:

```
(* Exercise 2 *)
(* Define a function member of type ''a * ''a list -> bool so that member(e,L) is true if and only if e is an element of the list L. *)
fun member (n, []) = false
|   member (n, head::tail) =
        if n = head then true
        else member (n, tail);

(* Exercise 3 *)
(* Define a function less of type int * int list -> int list so that less(e,L) is a list of all the integers in L that are less than e. *)
fun less (n, []) = []
|   less (n, head::tail) =
        if head < n then head::less (n, tail)
        else less (n, tail);

(* Exercise 4 *)
(* Define a function repeats of type ''a list -> bool so that repeats(L) is true if and only if the list L has two equal elements next to each other. *)
fun repeats [] = false
|   repeats [a] = false
|   repeats (a::b::tail) =
        if a = b then true else repeats (b::tail);

(* Exercise 5 *)
(* Represent a polynomial using a list of its (real) coefficients, starting with the constant coefficient and going only as high as necessary. Write a function eval of type re
fun eval ([], _) = 0.0
|   eval (head::tail, x) =
        let
            fun evalHelper ([], _) = 0.0
            |   evalHelper (head::tail, x) =
                    x * head + evalHelper (tail, x * x);
        in
            head + evalHelper (tail, x)
        end;
```

```
(* In the following exercises, implement sets as lists. *)
(* Exercise 8 *)
(* Write a function to test whether an element is a member of a set. *)
fun memberSet ([], _) = false
|   memberSet (head::tail, element) =
         if head = element then true
         else memberSet (tail, element);
```

Week 4 Assignment > quicksort.sml

```
fun partition (pivot, nil) = (nil, nil)
|   partition (pivot, target::rest) =
        let
            val (small, big) = partition(pivot,rest)
        in
            if target < pivot
                then (target::small, big)
            else (small, target::big)
        end

fun quicksort nil = nil
|   quicksort [a] = [a]
|   quicksort (first::rest) =
        let
            val (lessthan, greater) = partition(first,rest)
        in
            quicksort(lessthan) @ [first] @ quicksort(greater)
        end

```

Week 4 Assignment > intersectConstruct2.sml

```
fun isMember ([], _) = false
|   isMember (head::tail, element) =
        if head = element then
            true
        else isMember (tail, element)

fun intersection([],y) = []
|   intersection(first::rest, y) =
        if isMember(y,first) then
            first :: intersection(rest,y)
        else intersection(rest,y)
```

Union Constructor (incomplete)

Week 4 Assignment > unionConstructor2.sml

```
fun isMember ([], _) = false
|   isMember (head::tail, element) =
        if head = element then
            true
        else isMember (tail, element)

fun unionConstructor ([], y) = y
|   unionConstructor (first::rest, y) =
        if isMember(y, first) then
            union(y, rest)
        else first::union(rest, y)

```

```
(* Exercise 11 *)
(* Write a function to construct the powerset of any set. *)
fun consPowerset [] = [[]]
|   consPowerset (head::tail) =
    let
        fun powersetHelper ([], element) = [[]]
        |   powersetHelper ([]::tail, element) = [[element]]@powersetHelper (tail, element)
        |   powersetHelper (head::tail, element) = [head,head@[element]]@powersetHelper (tail, element);
    in
        powersetHelper (consPowerset tail, head)
    end;
```