fun map f [] = []
    | map f (x::xr) = f x :: map f xr;

SML: anonymous functions

fun double x = 2 * x;

map double [4, 5, 89];

map (fn x => 2 * x) [4, 5, 89];
SML: the list iterator \texttt{foldr}

An application \texttt{foldr f e xs} replaces ‘::’ by \texttt{f} and \texttt{[]} by \texttt{e} in list \texttt{xs}:

```ml
fun foldr f e [] = e
  | foldr f e (x::xr) = f(x, foldr f e xr);

fun len xs = foldr (fn (_, res) => 1+res) 0 xs;
fun sum xs = foldr (fn (x, res) => x+res) 0 xs;
fun prod xs = foldr (fn (x, res) => x*res) 1 xs;
fun map g xs = foldr (fn (x, res) => g x :: res) [] xs;
```

Using \texttt{foldr} and \texttt{map} to compute \texttt{freevars} for arbitrary primitives

```ml
fun freevars e : string list =
  case e of
    CstI i => []
  | Var x => [x]
  | Let(x, erhs, ebody) =>
    union (freevars erhs, minus (freevars ebody, [x]))
  | Prim(ope, es) => foldr union [] (map freevars es)
```
Lists are an example of an inductive type. The existence of `foldr` captures induction. More on the board.
SML: the tree iterator tfold replaces Br by f and Lf by e

```ml
fun tfold f e Lf = e
    | tfold f e (Br(v, t1, t2)) = f(v, tfold f e t1, tfold f e t2)

fun sumtree t = tfold (fn (v, r1, r2) => v + r1 + r2) 0 t
```
fun reverse xs = foldr (fn (x, ys) => ys @ [x]) [] xs;
> val 'b reverse = fn : 'b list -> 'b list
- reverse [1,2,3];
> val it = [3, 2, 1] : int list

fun foldl (g: 'a -> 'b -> 'a) (acc: 'a) [] = acc
| foldl g acc (x::xs) = foldl g (g acc x) xs;

fun reverse' xs = foldl (fn ys => fn x => x::ys) [] xs;
> val 'b reverse' = fn : 'b list -> 'b list
- reverse' [1,2,3];
> val it = [3, 2, 1] : int list
Exercise

Use `foldr` to write function `filter`:

\[ \text{`a list} \rightarrow (\text{`a} \rightarrow \text{bool}) \rightarrow \text{`a list.} \]
Consider the following data type of terms:

```plaintext
datatype term =  
    Var of int  
  | Fun of string * term list;

exception Unify;
```

You should define a type

```plaintext
subst
```

for representing substitutions on terms and a function

```plaintext
apply_subst : subst -> term -> term
```

for applying a substitution to a term. Finally, you should write a function

```plaintext
unify : term * term -> subst
```

which, given two terms returns the most general unifier for the two terms; if no unifier exists you should raise the exception `Unify`. 

Lars Birkedal (PLS, ITU)  
Introduction to ML, III  
Spring 2009
Example, given the code below

```ml
val v0 = Var 0
val v1 = Var 1
val v2 = Var 2
val k = Fun("k", [])
val l = Fun("l", [])

val t1 = Fun("f", [v0, v1, k])
val t2 = Fun("f", [l, v2, k])

val S = unify(t1, t2)

val res = (apply_subst S t1) = (apply_subst S t2)
```

res should be bound to true.