

ML LECTURE 1 Homework questions

Write ML functions `area` and `perim` to respectively compute the area and perimeter of a square given its length `l`.

```
fun square_area l = l * l : real;  
fun square_perim l = 4.0 * l : real;
```

Write a recursive ML function `fib` to compute the n th Fibonacci number (this is the n th element in the sequence 1, 1, 2, 3, 5, 8, 13, ...).

```
fun fib n = if n = 0 orelse n = 1 then 1  
            else fib (n-1) + fib (n-2);
```

ML LECTURE 2 Homework questions

Write a function `length` to return the length of a list.

```
fun length [] = 0
  | length (x::xs) = 1 + length xs;
```

Write a function `upperList` to take a list of characters and to return the list of corresponding uppercase characters.

```
fun upperList [] = []
  | upperList (x::xs) = (toupper x)::(upperList xs);
```

Write a function `upperString` to take a string and to return the string of corresponding uppercase characters.

```
fun upperString s = implode (upperList (explode s));
```

ML LECTURE 3 Homework questions

Using a tuple to represent a complex number write functions to add, subtract and multiply two complex numbers.

```
fun complex_add ((r1,i1),(r2,i2)) = (r1+r2,i1+i2) : real*real;
fun complex_sub ((r1,i1),(r2,i2)) = (r1-r2,i1-i2) : real*real;
fun complex_mult ((r1,i1),(r2,i2)) = (r1*r2-i1*i2,r1*i2+r2*i1) : real*real;
```

What is the type(s) of +?

It has two types: `int*int -> int` and `real*real -> real`

Using `split` and `merge` write `mergeSort`.

```
(mergesort.ml *)
fun merge([],M) = M
  | merge(L,[]) = L
  | merge(L as x::xs, M as y::ys) =
    if x < y then x::merge(xs,M)
    else y::merge(L,ys);

fun split [] = ([],[])
  | split [a] = ([a],[])
  | split (a::b::cs) =
    let
      val (M,N) = split(cs)
    in
      (a::M, b::N)
    end;

fun mergeSort [] = []
  | mergeSort [a] = [a]
  | mergeSort L =
    let
      val (M,N) = split L;
      val M = mergeSort M;
      val N = mergeSort N;
    in
      merge (M,N)
    end;
```

Write a function to implement quicksort. Hint: simply use the first element of the list as the partition element.

```
(* quicksort *)
fun partition(x,[]) = ([],[])
  | partition(x,y::ys) =
    let
      val (lxs,gxs) = partition(x,ys)
    in
      if x > y then
        ((y::lxs,gxs))
      else
        ((lxs,y::gxs))
    end;

fun quickSort [] = []
  | quickSort [a] = [a]
  | quickSort (x::xs) =
    let
      val (lxs,gxs) = partition(x,xs)
    in
      quickSort(lxs)@(x::quickSort(gxs))
    end;
```

In a previous lecture we gave a definition of a function **reverse** that reversed a list. Unfortunately its time complexity is quadratic in the size of the input list.

See if you find a better definition of **reverse** which is linear in the size of the input list.

```
(* move elements from the first list to the
   second and then return this list *)
fun rev1 ([], ys) = ys
  | rev1 ((x::xs), ys) = rev1 (xs,(x::ys));
fun reverse xs = rev1 (xs,[]);
```

ML LECTURE 4 Homework questions

Add appropriate exception handling to `max`.

```
exception Empty;

(* find the maximum element in a list *)
fun max [] = raise Empty
  | max [x] = x
  | max (x::xs) =
    let val xmax = max xs in
      if x > xmax then x else xmax
    end;
```

Write a function to print out a list of real numbers.

```
fun printListReals [] = ()
  | printListReals (x::xs) = (
    print(Real.toString(x));
    print("\n");
    printListReals xs
  );
```

Write a polymorphic function to find the largest item in a list. If you can't do this explain why you are not stupid.

Trick question: not possible, since any function to find the largest element cannot be polymorphic since it must depend on the comparison function used to compare elements in the list.

ML LECTURE 5 Homework questions

Modify the `falafelRoll` definition so that it keeps track of the number of falafel balls in the roll.

```
datatype falafelRoll =  
  Pita |  
  Falafel of int * falafelRoll |  
  Tabouli of falafelRoll |  
  Pickles of falafelRoll |  
  Hommus of falafelRoll |  
  Chilli of falafelRoll;
```

Now write a function which returns the total number of falafel balls in the roll.

```
fun numFalafel Pita = 0  
  | numFalafel (Falafel(n,r)) = n+numFalafel(r)  
  | numFalafel (Tabouli(r)) = numFalafel(r)  
  | numFalafel (Pickles(r)) = numFalafel(r)  
  | numFalafel (Hommus(r)) = numFalafel(r)  
  | numFalafel (Chilli(r)) = numFalafel(r);
```

Write a datatype which represents hamburgers.

```
datatype hamburger =  
  Bun |  
  Hamburger of hamburger |  
  Cheese of hamburger |  
  Pickles of hamburger |  
  Lettuce of hamburger |  
  Tomatoe of hamburger;
```

Define a function which checks that a hamburger is without cheese.

```
fun noCheese Bun = true  
  | noCheese (Hamburger(h)) = noCheese h  
  | noCheese (Cheese(h)) = false  
  | noCheese (Pickles(h)) = noCheese h  
  | noCheese (Lettuce(h)) = noCheese h  
  | noCheese (Tomatoe(h)) = noCheese h;
```

Write a function to traverse the elements in a labelled binary tree in-order and return the result in a list.

```
fun traverse Empty = []  
  | traverse (Node(lt,lbl,rt)) = (traverse lt)@(lbl::(traverse rt));
```

ML LECTURE 6 Homework questions

Consider the following definitions for a function to concatenate three lists

```
fun concat3a (xs,ys,zs) = xs@ys@zs;  
fun concat3b xs ys zs = xs@ys@zs;  
fun concat3c xs (ys,zs) = xs@ys@zs;
```

What are the types for each and what is the difference between them?

```
concat3a: 'a list * 'a list * 'a list -> 'a list  
concat3b: 'a list -> 'a list -> 'a list -> 'a list  
concat3c: 'a list -> 'a list * 'a list -> 'a list
```

`concat3a` takes a triple of lists and returns a list.

`concat3b` takes a list and returns a function which takes a list and returns a function which takes a list and returns a list. It could be defined by

```
fun concat3b xs = (fn ys => (fn zs => xs@ys@zs));
```

`concat3c` takes a list and returns a function which takes a pair of lists and returns a list. It could be defined by

```
fun concat3c xs = (fn (ys,zs) => xs@ys@zs);
```

Using `simpleMap`, `filter` and `reduce` write a function `nnsqsum` which sums the squares of the non-negative numbers in a list. Now write a version which is recursive and does not use any higher-order predicates.

```
fun nnsqsum L = reduce (op +)  
  (simpleMap (fn x => x*x)  
   (filter (fn x => x >= 0.0) L))  
  
fun nnsqsum [] = 0.0  
  | nnsqsum (x::xs) = if x >= 0.0 then (x*x)+(nnsqsum xs)  
                     else (nnsqsum xs);
```

The variance of a list of reals is a measure of the “spread” from the mean. More precisely, the variance of $[a_1, \dots, a_n]$ is

$$\frac{(\sum_{i=1}^n a_i^2)}{n} - \left(\frac{\sum_{i=1}^n a_i}{n}\right)^2.$$

Write a function `variance` which uses `reduce` and `simpleMap` to compute the variance of a list of reals.

You can use `len` to compute the length of a list. Now write version of `variance` which does not use higher order programming. Again you can use `len`.

```
fun square x = x*x:real;

fun variance xs =
  let
    val n = real (len xs)
  in
    reduce (op +, simpleMap (square,xs))/n -
    square (reduce (op +,xs)/n)
  end;

fun sumsquares [] = 0.0
  | sumsquares (x::xs) = (square x) + (sumsquares xs);
fun sum [] = 0.0
  | sum (x::xs) = x + (sum xs);
fun variance xs =
  let
    val n = real (len xs)
  in
    (sumsquares xs)/n - (square (sum xs)/n);
```

Using ML’s higher order built-ins write a function which takes a list of strings and concatenates them all.

```
fun concatenate strings = foldr (op ^) "" strings;
```

Using ML’s higher order built-ins write a function which converts a list of integers into the corresponding list of reals.

```
fun toReals ints = map real ints;
```

0.1 Extended Homework

Your job is to write a data type and functions in Standard ML for representing and manipulating Boolean expressions:

- An datatype definition for `BoolExp` a data type representing Boolean expressions. This should have different data constructors for the different kinds of Boolean expressions: *true*, *false*, *not*, *and*, *or* and variables.

```
datatype BoolExp =
  True |
  False |
  Var of string |
  Not of BoolExp |
  And of BoolExp * BoolExp |
  Or of BoolExp * BoolExp;
```

- Functions for constructing a Boolean expression:

```
const: bool -> BoolExp
var: string -> BoolExp
lnot: BoolExp -> BoolExp
land: BoolExp * BoolExp -> BoolExp
lor: BoolExp * BoolExp -> BoolExp
```

The function `bool` takes a Boolean constant *true* or *false* and returns the corresponding Boolean expression, `var` takes the name of a variable and returns a Boolean expression variable with that name, while `lnot`, `land` and `lor` combine Boolean expressions with the operators *not*, *and* and *or* respectively.

```
fun const true = True
  | const false = False;
fun var v = Var(v);
fun lnot b1 = Not(b1);
fun land (b1,b2) = And(b1,b2);
fun lor (b1,b2) = Or(b1,b2);
```

- A function `BoolExpToString: BoolExp -> string` for returning a string describing a Boolean expression.

```
fun BoolExpToString True = "true"
  | BoolExpToString False = "false"
  | BoolExpToString (Var v) = v
```

```

| BoolExpToString (Not b) = "not("^BoolExpToString b^")"
| BoolExpToString (And(b1,b2)) =
  "("^BoolExpToString b1^") and (^BoolExpToString b2^")"
| BoolExpToString (Or(b1,b2)) =
  "("^BoolExpToString b1^") or (^BoolExpToString b2^)";

```

- A function `evalBoolExp`: `(string -> bool) -> BoolExp -> bool` for evaluating a Boolean expression with a valuation.

```

fun evalBoolExp e True = true
| evalBoolExp e False = false
| evalBoolExp e (Var v) = e(v)
| evalBoolExp e (Not (b)) = (not (evalBoolExp e b))
| evalBoolExp e (And(b1,b2)) = (evalBoolExp e b1) andalso (evalBoolExp e b2)
| evalBoolExp e (Or(b1,b2)) = (evalBoolExp e b1) orelse (evalBoolExp e b2);

```

- A function `vars`: `BoolExp -> string list` for returning a list of the variables occurring in a Boolean expression. Each variable should occur exactly once in the list.

```

fun merge([],M) = M
| merge(L,[]) = L
| merge(L as x::xs, M as y::ys) =
  if (x:string) < y then x::merge(xs,M)
  else if x > y then y::merge(L,ys)
  else x::merge(xs,ys);

```

```

fun vars True = []
| vars False = []
| vars (Var v) = [v]
| vars (Not (b)) = (vars b)
| vars (And(b1,b2)) = merge ((vars b1),(vars b2))
| vars (Or(b1,b2)) = merge ((vars b1),(vars b2));

```

- A function `satisfiable`: `BoolExp -> bool` which determines if a Boolean expression is satisfiable.

```

fun test [] e B = (evalBoolExp e B)
| test (v::vs) e B = (test vs (fn s => if v=s then false else (e s)) B)
  orelse (test vs e B);
fun satisfiable B =
  test (vars B) (fn x => true) B;

```

ML LECTURE 7 Homework questions

Define a signature for a generic `Mapping` structure. The mapping is a list of pairs `(d,r)` where `d` is the domain type and `r` the range type. For any domain value there is at most one pair in the list with the value as the first component in the pair. The structure should provide 3 functions:

- `create` to produce the empty list;
- `lookup` to find the range value associated with a given domain value, it should throw the `NotFound` exception if there is no associated value;
- `insert` which takes a domain and range element `d` and `r` and makes `r` the unique range value associated with `d`.

See pg 263 Ullman ML text.

Modify your answer to the extended homework so that it makes use of `local` function definitions and makes the Boolean expression data type an abstract data type.

```
abstype BoolExp =
  ...
with
fun BoolExpToString True = ...;

fun const true = True
  | const false = False;
fun var v = Var(v);
fun lnot b1 = Not(b1);
fun land (b1,b2) = And(b1,b2);
fun lor (b1,b2) = Or(b1,b2);

fun evalBoolExp e True = true
  | evalBoolExp e False = false
  | evalBoolExp e (Var v) = e(v)
  | evalBoolExp e (Not (b)) = (not (evalBoolExp e b))
  | evalBoolExp e (And(b1,b2)) = (evalBoolExp e b1) andalso (evalBoolExp e b2)
  | evalBoolExp e (Or(b1,b2)) = (evalBoolExp e b1) orelse (evalBoolExp e b2);

local
fun merge([],M) = M
```

```

| merge(L, []) = L
| merge(L as x::xs, M as y::ys) =
  if (x:string) < y then x::merge(xs,M)
  else if x > y then y::merge(L,ys)
  else x::merge(xs,ys);
in
fun vars True = []
| vars False = []
| vars (Var v) = [v]
| vars (Not (b)) = (vars b)
| vars (And(b1,b2)) = merge ((vars b1),(vars b2))
| vars (Or(b1,b2)) = merge ((vars b1),(vars b2));
end;

local
fun test [] e B = (evalBoolExp e B)
| test (v::vs) e B = (test vs (fn s => if v=s then false else (e s)) B)
  orelse (test vs e B);
in
fun satisfiable B =
  test (vars B) (fn x => true) B;
end
end;

```

ML LECTURE 8 Homework questions

Can you use `while` loops without using destructive update? For instance consider

```
val i = 0;
  while i < 5 do (
    print(Int.toString(i));
    print(" ");
    val i = i + 1;
  );
```

No

Name an application for which you would prefer to use ML to C.

Automatic theorem proving, symbolic processing in general, compiler writing...