ML Exercise Sheets
Part Ia CST and Mathematics with Computer Science

Here are the exercises for the ML Practical Classes. These Classes complement the Foundations of Computer Science Lectures given by L.C. Paulson.

Exercises 1, 2, 3, 4, 5 and 6 are compulsory and each of them is worth one tick. Exercises 3*, 5* and 6* are optional. Satisfactory solutions to starred exercises will be recorded on the Tick List but will not count as extra ticks. Each exercise is assessed on the basis of printouts of sessions and programs, after discussion with a demonstrator.

In addition to learning about the ML language and its implementation, it will be necessary to learn how to use the current generation of workstations. All these skills and full details of the assessment procedure will be explained at the first two Thursday-afternoon classes.

Another point which will be explained is the style to which the submitted printouts must conform. Thus, each printout should carry the name of the person who submitted it, the time it took to complete the exercise, any functions which an exercise requires to be written, and examples of such functions being tested.

No exercise requires elaborate written answers. Questions like 'What is the purpose of ...?' require a one- or two-line ML comment. Before awarding a tick, the demonstrator will go over the printout of the session with you, to see whether you can explain ML's responses, and understand what is going on.

On the basis of past experience, roughly the same number of people will find the exercises too easy as find them too hard. Likewise, roughly the same number of people are likely to find that the exercises take a very short time as find they take too much time. The average time spent on each weekly exercise is approximately three hours but times wildly different from this average may be expected!

Other (non-assessable) problems will be presented in the Lectures and these should be attempted, especially by those who find the exercises in this document too easy. Those who find they have time on their hands may also usefully study details of the local computing facilities. The Computing Service have a good deal of documentation available and also run classes, both introductory and more advanced, on many topics.

L.C. Paulson (Main Text)
October 1988

F.H. King (Minor Amendments)
October 1998

Exercise 1 — Introduction to ML

Full details of what constitutes a solution to this exercise will be explained at the first Thursday-afternoon class. There will be an associated handout which gives an exact specification of what is required. The purpose of the exercise is to learn how to log in to a workstation, invoke the ML application, print out a trivial example, and exit again.

Remember to type each line *exactly* as given and press the ENTER key at the end of the line.

After logging in, enter Windows and then enter ML.

ML will print various messages, then signal that it is ready by printing a line containing just a hyphen (called the *prompt character*):

```
- type ML text here ...
```

Now type each of the following lines, one at a time. Be *very* careful not to omit any quotation marks! Use small letters as shown — ML considers `b` to be different from `B`. Also, (`"`) is a single character, not a repeated (`'`). For aesthetic reasons and as explained in the associated handout, it is a requirement that the ENTER key should also be pressed *before* each line to ensure that the typed lines and responses appear in vertically-separated pairs.

```
val a = "very ";
a ^ "red";
val b = a^a;
b ^ "rich";
val c = b^b;
c ^ "ridiculous";
```

ML should respond to each line with, e.g., `> val a = "very " : string`. It should print a prompt character on the next line to say it is ready for a new command. If you make a typing error, ML will probably print an error message and then a prompt character. However, certain errors will cause ML to expect another line of input. ML signals this by printing an equals sign instead of a hyphen:

```
- val a = "very ;          A quotation mark was omitted ...
=                          ... so ML continues to read the string
```

If you get stuck like this, it may be possible to recover by typing the missing character.

Exercise 2 — Recursion vs Iteration, Part I

Before working any questions, try using ML as a desk calculator. The integer operators include

```
+    -    *    div    mod
```

The real operators include

```
+    -    *    /
```

Relations (for both integers and reals) include

```
=    <>    <    >    <=    >=
```

Note also the function `real` which converts an integer to a real number. The function `floor` converts a real number x to the largest integer i such that $i \leqslant x$.

1. Write an ML function `facr(n)` which determines $n!$ by recursion.

2. Write an ML function `faci(n)` which determines $n!$ by iteration (in the sense described in Lecture 2).

Exercise 2* — Recursion vs Iteration, Part II

Note that although the following Week 2 problems will not count towards a 'tick' it is a very good idea to attempt them before Week 3.

1. Write two versions, the first recursive and the second iterative, of an ML function `sumt(n)` to sum the n terms

$$1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{n-1}}$$

where n may be 1, 2, 3, etc. When $n = 2$ the sum is $\frac{1}{2^0} + \frac{1}{2^1}$, namely 1.5.

Observe that each term can be cheaply computed from its predecessor. A fancy treatment of this is to consider the slightly more general function

$$f(x, n) = x + \frac{x}{2} + \frac{x}{4} + \cdots + \frac{x}{2^{n-1}}$$

This function satisfies the recurrence (for $n = 0, 1, 2, \ldots$)

$$f(x, n+1) = x + f(x/2, n)$$

2. Write an ML function `eapprox(n)` to sum the n terms in the approximation

$$e \approx 1 + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{(n-1)!}$$

Again, each term can be cheaply computed. Is your function recursive or iterative?

3. Write an ML function `exp(z,n)` to compute exponentials:

$$e^z \approx 1 + \frac{z}{1!} + \frac{z^2}{2!} + \cdots + \frac{z^{n-1}}{(n-1)!}$$

Exercise 3 — Structured Data: Pairs and Lists

Before working the questions, you are advised to try some simple experiments with structured data. Enter ML and define the following selector functions:

```
fun fst (x,y) = x;
fun snd (x,y) = y;
```

To experiment with them, type declarations like

```
val p = ("red",3);
val q = (p, "blue");
val r = (q, p);
val s = ((23,"grey"), r);
```

and then type things like

```
fst (fst q);          fst (fst p);          fst(snd s);
```

Structures can contain functions. Try some examples like these:

```
val u = (fst,snd);
fst(u)(1,2);
```

Note: Triples are not pairs! Compare ML's response to each of the following:

```
fst((1,2),3);         fst(1,2,3);
```

1. Recall that the function

```
fun hd (x::_) = x;
```

returns the first element of a list. Getting at the last element is harder. Write a recursive function `last` to return the last element of a list. Comment, using O-notation, on the time and space complexity of `last(xs)`, comparing it with `hd(rev xs)`.

2. Now do the same thing for `tl`: write a recursive function `butLast` to *remove* the last element of a list. For example, `butLast[1,2,3,4]` should return `[1,2,3]`. Note that `butLast(xs)` must be equivalent to `rev(tl(rev xs))` so `butLast[1]` should return `[]`. Compare the time and space complexity of `butLast` with `rev(tl(rev xs))`.

3. Write a function `nth` such that `nth(x,n)` returns the nth element of list `x`, counting the head of the list as element zero.

Exercise 3* — List of Lists

Write a function `choose(k,xs)` that returns all `k`-element lists that can be drawn from `xs`, ignoring the order of list elements. If n is the length of `xs`, then (provided $k \leqslant n$) the result should be an $\binom{n}{k}$-element list. Here are some sample inputs and outputs

```
- choose (3, [1,2]);
> [] : (int list) list

- choose (3, [1,2,3]);
> [[1,2,3]] : (int list) list

- choose (3, [1,2,3,4,5]);
> [[1,2,3],[1,2,4],[1,2,5],[1,3,4],[1,3,5],[1,4,5],[2,3,4],
                           [2,3,5],[2,4,5],[3,4,5]] : (int list) list
```

Exercise 4 — Route Finding

This problem is concerned with finding routes through a system of one-way streets. Suppose we have an ML list of pairs

$$[(x_1, y_1), \ldots, (x_n, y_n)]$$

where each (x_i, y_i) means that there is a route from x_i to y_i. (This need *not* mean there is a route from y_i to x_i!) The exercises on this sheet lead up to a programme for producing the list of all pairs (x, y) such that there is a route from x to y involving one or more steps from the input list. (So do not include (x, x) unless there is a non-trivial route from x back to itself.)

1. Write a function `startpoints(pairs,z)` that produces the list of all x such that (x, z) is in the list `pairs`.

2. Write a function `endpoints(z,pairs)` that produces the list of all y such that (z, y) is in the list `pairs`.

3. Write a function `allpairs(xs,ys)` that produces the list of all (x, y) for x in the list `xs` and y in the list `ys`.

4. Call a list of pairs *complete* if whenever (x, z) and (z, y) are in the list, then (x, y) is also in the list. Write a function `addnew((x,y),pairs)`, where the list `pairs` is expected to be complete. The result should be a new list containing (x, y), the elements of `pairs`, and just enough additional pairs to make the result list complete.
Hint: use `startpoints`, `endpoints`, and `allpairs`.

5. Write a function `routes(pairs)` that finds all possible routes from one point to another via the `pairs`. The result of the function should be a complete list of pairs. Hint: use `addnew`.

Test it on things like $[(1, 2), (2, 1)]$: it must not loop! Test it on bigger examples. How efficient is your programme?

Mathematicians may like to know that this problem is really concerned with the transitive closure of a relation. A list of pairs represents the finite relation R that holds just between those pairs (x_i, y_i) in the list. The *transitive closure* of R is another relation R^+, and $R^+(x, y)$ holds just when there is a chain from x to y along R:

$$x = x_0,\ R(x_0, x_1),\ \ldots,\ R(x_{n-1}, x_n),\ R(x_n, y), \quad n \geqslant 0$$

Thus $R(x, y)$ implies $R^+(x, y)$ and $(R \circ R^+)(x, y)$ implies $R^+(x, y)$, but we do not necessarily have $R^+(x, x)$.

Note: it is *not* necessary to estimate the cost of the `routes` function to secure a 'tick' but a reasoned derivation of the cost expressed in big-O notation will count as Exercise 4*.

Exercise 5 — Functions as Arguments and Results

1. Consider the family of functions $f_0, f_1, f_2, \ldots$, where $f_0(n) = n$, $f_1(n) = 1 + n$, $f_2(n) = 2 + n$, ... In general, for integers m and n,

$$f_m(n) = m + n$$

Write an ML function `plus` such that `plus(m)` returns f_m (as an ML function). Either use `let fun` to declare a local function, or use `fn`. In fact, ML offers a special syntax for defining functions like `plus`:

```
fun plus m n : int = m+n;
```

These are sometimes called *curried functions*.

Evaluate the definitions

```
val succ = plus 1;
fun add(m,n) : int = m+n;
```

Test all these functions. What is the relationship between `plus` and `add`?

2. If f is a function and $n \geqslant 0$ is an integer then the function f^n is defined as follows:

$$f^n(x) = \underbrace{f(f(\cdots f(}_{n \text{ times}} x)\cdots))$$

In particular, $f^0(x) = x$.

Given that s is the function such that $s(x) = x + 1$ (i.e. it simply adds 1 to its argument), we can express the sum of two non-negative integers m and n as $m+n = s^n(m)$ (i.e. 1 is added to m but n times over).

Express the product $m \times n$ and power m^n similarly. Hint: consider what has to be done to what n times over to obtain $m \times n$ and what has to be done to what n times over to obtain m^n. Note that the functions which are equivalent to $s(x)$ may have to depend upon m.

3. Write an ML function `nfold` such that `nfold(f,n)` returns the function f^n. Use `nfold` and the results of the previous exercise to write functions to compute sums, products, and powers.

Exercise 5* — Generic Operations

This exercise concerns a universal type we can define in ML to cope with generic data. This type comprises numbers, booleans, and strings. It also permits pairing, which represents all forms of structured data. Finally, the constructor `Named` lets us easily attach a 'type label' (a string) to any piece of data.

1. Enter the following declaration and experiment with building various values of type `univ`.

```
datatype univ = Int of int
              | Real of real
              | Bool of bool
              | String of string
              | Pair of univ*univ
              | Named of string*univ;
```

2. Type in the function `uofl`:

```
fun uofl [] = String"nil"
  | uofl (u::us) = Pair(u, uofl us);
```

To find out what it does, execute things like

```
uofl [ Int 3, Real 1.4, String "what" ];
```

Write and test the function `lofu`, an inverse of `uofl` in the sense that `lofu(uofl(l))` should always equal l. *Hint*: fill in the `???` in the following:

```
fun lofu (String"nil") = ???
  | lofu (Pair(u, v)) = ???
```

3. What does this function do? Could you imagine writing a similar function that worked on ordinary ML values?

```
fun ints (Int n) = [n]
  | ints (Pair(u, v)) = ints u @ ints v
  | ints (Named(s,u)) = ints u
  | ints u = [];
```

Write a function similar to `ints` to find the sum of all `Real` numbers contained in a `univ`.

Exercise 6 — Integer Streams, Part I

1. Here is a definition of integer streams. Calling `makeints(n)` makes the stream of all integers starting with `n`; note its use of `fn()=>` to create a function. Calling `tail(s)` applies this function to the dummy value `()`.

```
datatype stream = Item of int * (unit->stream);
fun cons (x,xs) = Item(x, xs);
fun head (Item(i,xf)) = i;
fun tail (Item(i,xf)) = xf();
fun makeints n = cons(n, fn()=> makeints(n+1));
```

The ML identifier `it` always holds the value of the last expression typed at top level (that is, not contained in a `val` declaration). Test `makeints` by typing

```
makeints 4;
tail it;
tail it;
```

By typing `tail it;` again and again, you can see successive items in the stream.

2. Calling the function `maps f xs`, where `xs` is the stream $x_1, x_2, x_3, \ldots$, returns the stream $f(x_1), f(x_2), f(x_3), \ldots$:

```
fun maps f xs = cons(f (head xs), fn()=> maps f (tail xs));
```

Note the very typical use of `fn()=>`, `head`, and `tail`.

3. Write a function `nth(s,n)` to return the `n`th element of sequence `s`. For example, `nth(makeints 1,100)` should return 100. Use `maps` and `makeints` to make the stream of positive squares (1, 4, 9, ...) and find its 49th element.

4. Write the function `filters f xs` to return the stream of all x_i in `xs` such that $f(x_i)$ is true. Use it to make the stream of positive integers that are not divisible by 2 or 3.

Exercise 6* — Integer Streams, Part II

1. Write a function `map2 f xs ys`, similar to `maps`, to take streams $x_1, x_2, x_3, \ldots$ and $y_1, y_2, y_3, \ldots$ and return the stream $f(x_1)(y_1), f(x_2)(y_2), f(x_3)(y_3), \ldots$

2. The Fibonacci Numbers are defined as follows: $F_1 = 1$, $F_2 = 1$, and $F_{n+2} = F_n + F_{n+1}$. So new elements of the sequence are defined in terms of two previous elements. If ML lists were streams then we could define the steam of Fibonacci Numbers (in pseudo-ML) as follows:

```
val fibs = 1 :: 1 :: map2 plus fibs (tail fibs);
```

Here `plus m n = m+n`, and two copies of `fibs` recursively appear in the definition of this stream. But this code is not legal; we have to use `cons`. We also have to force `fibs` into a function, since in ML only functions can be recursive. So the following is legal:

```
fun fibs() =
    cons(1, fn()=>
         cons(1, fn()=>   map2  plus  (fibs())  (tail(fibs())) ));
```

Use this code to compute the fifteenth Fibonacci Number.

3. Write a function `merge(xs,ys)` that takes two *increasing* streams, $x_0 < x_1 < x_2 < \ldots$ and $y_0 < y_1 < y_2 < \ldots$, and returns the increasing stream containing all the x's and y's. Since the input streams are increasing, you need to compare their heads, take the smaller one, and *recursively* merge whatever remains. Make certain there are no repeated elements in the output stream.

4. Construct in ML the increasing stream containing all numbers of the form $2^i \times 3^j$ for integers i, $j \geqslant 0$. *Hint*: The first element is 1, and each new element can be obtained by multiplying some previous element by 2 or 3. The code is similar to `fibs`, and calls `merge`.

5. Construct the increasing stream of all numbers of the form $2^i \times 3^j \times 5^k$ for integers i, j, $k \geqslant 0$. What is the sixtieth element of this stream?