UNIVERSITY OF CAMBRIDGE COMPUTER LABORATORY

First-Year Computer Science

ML Exercise Sheets

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 and 5 are compulsory for all candidates and each of them is worth one tick. Exercise 6 is compulsory for CST candidates but not for candidates who borrow Paper 1. Exercises 2*, 3*, 4* (yes, it's there!) and 5* 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 which are submitted to a demonstrator who will comment on them. You must keep marked exercises after they are returned to you, because every Part IA student is required to submit "a portfolio of assessed laboratory work" (ML, Java, and Hardware if applicable) in Easter Term.

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!

Due to a timetable change introduced in October 2010, these exercises now concern material that will often not have been lectured by the time they are given to students. To compensate for this situation, we have added explanations and simplified some of the problems. If however you find a problem impossible to understand, you may prefer to leave it for the following week. It is not essential to solve each exercise on the day that is given out. The final deadline for handing in exercises is in Lent term, and it will be announced to all students.

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, *Course Lecturer*

M.G. Kuhn, *Part IA Coordinator*

October 2011

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 (or RETURN) 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. Use small letters as shown — ML considers `x` to be different from `X`. 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 x = 0.1;
val y = x + x;
val z = y + y;
2.0 - z - z - y;
2.0 - x - x - x - x - x - x - x - x - x - x;
```

ML should respond to each line with, e.g., `> val x = 0.1 : real`. 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 y = x +                    ENTER was pressed too early ...
=                                   ... so  ML continues reading
```

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

Remark: This exercise is designed for Cambridge ML, which we use in our practical sessions. The output you see may vary for other versions of ML. If you are attempting this exercise with a different version of ML, or are merely feeling adventurous, you might want to enter the following additional line:

```
1.0 - x - x - x - x - x - x - x - x - x - x;
```

This exercise shows how ML can be used as a calculator. Variables (such as `x`, `y`, and `z`) can be defined and used. (If you are experienced with another programming language, note that these variables behave differently from those that you are familiar with!) You will see that machine arithmetic is not always exact. Can you explain why?

Exercise 2 — Recursive Functions

You may wish to limber up for this exercise by performing numerical calculations using ML, as in last week's exercise. If you write numerical constants that include decimal points, and use the familiar arithmetic operators (`+ - * /`), then ML will perform *floating point arithmetic* resembling that done by calculators. If you write integer constants (no decimal points), then ML will perform exact integer arithmetic; the operators `div` and `mod` yield integer quotient and remainder, respectively.

A mathematical formula is often expressed in the form of a *function.* For example, a mathematician might express the formula for the area of a triangle, $xy/2$ as the function definition

$$\text{area}(x, y) = xy/2.$$

We can define a similar function in ML by typing

```
fun area (x,y) = x*y/2.0;
```

and now we can calculate the areas of particular triangles by typing expressions like

```
area(3.5, 4.0);
```

1. Enter this function definition and demonstrate it by calculating some areas of triangles, as shown above. (Decimal points are essential. Can you see why `area(3,4);` doesn't work?)

Mathematical function definitions are often recursive. The well-known factorial function, $n!$, is defined by $0! = 1$ and (for $n > 0$)

$$n! = n \times (n - 1)!$$

Even this can easily be defined in ML:

```
fun fact n = if n=0 then 1 else n * fact(n-1);
```

It specifies a computation that, given n, tests whether $n = 0$ or not, and in the latter case, calls itself with the value $n - 1$.

2. Now consider the following recursive definition of the number 2 raised to a power:

$$2^0 = 1 \qquad\qquad 2^n = 2 \times 2^{n-1} \quad (\text{for } n > 0)$$

Write the analogous definition of this function in ML. Demonstrate your function with several different values of n.

Exercise 2* — Recursive Functions Continued

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.

Remark: The function `real` converts an integer to a real number. The function `floor` converts a real number x to the largest integer i such that $i \leqslant x$. These functions will be useful in the examples below, which involve both integer and real calculations.

1. Write an ML function `sumt(n)` to sum the n terms

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

for $n > 0$. 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 recursive definition (for $n > 0$)

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

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.

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, try some simple experiments with structured data. Start 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);
```

Lists in ML are written inside square brackets, e.g. `[1,2,3]` is a list consisting of elements 1, 2 and 3. The empty list, `[]`, has no elements. Lists are built ("constructed") using a *constructor* called *cons*. We write cons as `::` between its arguments, e.g. `1::[2,3]` constructs the list `[1,2,3]`. Try a few examples:

```
val x = [1,2,3];
val y = 4::5::[8,9];
val z = 4::5::x;
val q = "red"::"green"::[];
```

Note: All elements in a list must have the same *type*! Observe ML's response to each of the following. Remember that `1.0` is a `real` and 2 is an `int`.

```
["orange",1,2];         1.0::[2];
```

Lists are taken apart using functions `hd` and `tl`, respectively called *head* and *tail*. The function `hd` returns the first element of a non-empty list and the function `tl` returns a list consisting of all elements following the first element. Try these by typing in a few examples:

```
val x = hd [1,2,3];
val y = tl [1,2,3];
val z = hd (5::[]);
val q = tl [1];
```

The function `null` tests whether or not a given list is empty. Writing `null` *mylist* is more general and efficient (for a number of obscure reasons) than writing *mylist* `= []`.

We can define functions that operate over lists using `hd` and `tl`. Experiment with the following functions: `length(xs)` returns the length of list `xs`, `sum(xs)` returns the sum of all element in list `xs` and `take(xs,n)` returns a list consisting of the first `n` elements of list `xs`.

```
fun length(xs) = if null xs then 0 else 1 + length(tl xs);
fun sum(xs) = if null xs then 0 else (hd xs) + sum(tl xs);
fun take(xs,n) = if n = 0 then [] else (hd xs) :: take(tl xs, n-1);
```

Experiment with these definitions on a few inputs:

```
val l = length [1,2,3];
val s = sum [5,6,7];
val xs = take ([7,6,5,4,3], 3);
```

Note: Lists can contain structured data, e.g. pairs of numbers `[(1,2),(4,5),(8,9)]` and functions can be defined for these similarly:

```
fun prod(i,j) = i * j : int;
fun sum_prod(xs) = if null xs then 0 else prod (hd xs) + sum_prod(tl xs);
```

Try this function on sample inputs, e.g. enter the following into ML:

```
val p = sum_prod [(1,2),(3,3)];
```

Do the following tasks to complete this Exercise:

1. The function `hd` 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. For example, on input `[1,2,3]`, your function `last` should return `3`.

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)` should return `[]` if the list `xs` has length 1.

3. Write a function `nth` such that `nth(xs,n)` returns the nth element of list `xs`, 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
```

Note: It might be useful to first define two auxiliary functions: a function which adds a specific element to all list in a list:

```
- allcons (6, [[1,2,3],[2],[]]);
> [[6,1,2,3],[6,2],[6]] : (int list) list
```

and a function which concatenates two lists (of lists) together:

```
- append ([[1],[2,3]],[[],[4,5,6]])
> [[1],[2,3],[],[4,5,6]] : int 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 program 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`. For example, `startpoints ([(1,2), (2,3), (2,1)], 2)` should yield `[1]`.

2. Write a function `endpoints(z,pairs)` that produces the list of all y such that (z, y) is in the list `pairs`. For example, `endpoints ([(1,2), (2,3), (2,1)], 2)` should yield `[3,1]`.

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. (The empty list is trivially complete.) Write a function `addnew((x,y),pairs)`, where you may assume the list `pairs` 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: This function should not use recursion, but should concatenate lists generated with the help of the functions `startpoints`, `endpoints`, and `allpairs`. Note that the new segment (x, y) can create new paths in three different ways: as a first or last step of a path or somewhere inside.

5. Write a function `routes(pairs)` that produces a list of all (x, y) such that there is a route from x to y via the `pairs` (which you should not assume to be complete). The result of the function should be a complete list of pairs. *Hint*: let `addnew` do all the work.

   For example, `routes [(1,2), (2,3), (2,1)]` should yield `[(1,2), (2,2), (1,3), (1,1), (2,3), (2,1)]`.

Mathematicians may like to know that this problem is 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.

A reasoned derivation of the the cost of the `routes` function, expressed in big-O notation, will count as Exercise 4*.

Exercise 5 — Functions as Arguments and Results; Integer Streams

In ML, functions can be given as input and can be returned as results from functions. For example, the function `twice` takes a function `f` as its first argument and applies `f` twice to the second argument, `x`.

```
fun twice (f,x) = f (f x);
```

The `fn` construct lets us express a function in-place; for example, `fn i=>i+5` denotes the function that returns `i+5` when applied to `i`. Try using `twice` with that function:

```
twice (fn i=>i+5, 3);
```

If we code `twice` as shown below, then instead of taking a pair of arguments, it takes simply the argument `f` and returns a function that applies `f` twice to *its* argument.

```
fun twice f = (fn x => f (f x));
```

Compare the type of this version of `twice` with that of the previous version. Can you explain why the following expression evaluates to 11?

```
twice (twice (twice (fn i=>i+1))) 3;
```

Do the following tasks to complete this Exercise:

1. 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 repeated n times over to obtain $m \times n$ and what has to repeated n times over to obtain m^n. Note that the functions that are analogous to $s(x)$ may have to depend upon m.

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

3. Here is a definition of integer streams (infinite lists). Calling `makeints(n)` makes the stream of all integers starting with `n`; note its use of `fn()=>exp` to create a function which expects `()` as input before evaluating expression `exp`, i.e. `fn()=>` is used to delay evaluation of `exp`. Calling `tail(s)` applies this function to the dummy value `()`, which prompts evaluation to happen.

```
datatype stream = Cons of int * (unit -> stream);
fun tail (Cons(i,xf)) = xf();
fun makeints n = Cons(n, fn()=> makeints(n+1));
```

Test `makeints` by typing

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

Demonstrate that typing `tail it;` again and again reveals successive items in the stream. (Recall that `it` always holds the value of the last top-level expression.)

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

Exercise 5* — Integer Streams Continued

Before undertaking this exercise, you may wish to wait until the corresponding lectures have been delivered.

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?

Exercise 6 — Vacation Task

The concluding Tick 6 exercise is the subject of a separate handout which will be issued later in the term. As noted on the front page, this exercise is compulsory for CST candidates but not for candidates who borrow Paper 1, though such candidates are welcome to try it.