UNIVERSITY OF CAMBRIDGE COMPUTER LABORATORY

First-Year Computer Science (Part IA)

ML Exercise Sheets

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

Exercises 1–4 are compulsory for all candidates and each is worth one tick. Exercise 5 is compulsory for CST candidates but not for candidates taking only Paper 1. Exercises 1*, 2*, 3* and 4* 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 *ticker*, who will comment on them.

Each exercise is pass/fail, and in the case of failure, can be resubmitted. Everybody should be able to pass every exercise eventually. Your code does not need to be complicated. Efficiency is unimportant unless the wording of the problem suggests otherwise. Please ensure that your submitted work includes your name, ID, and legible, neatly formatted ML code, along with examples of your code being tested. Please note how much time you spent, as this will help us ensure that the problems are of reasonable difficulty.

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

On the basis of past experience, as many students will find the exercises too easy as find them too hard. Likewise, as many students will 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!

The exercises are intended to be completed one-per-week to match the lectures. Each exercise should be ticked within two weeks; for instance, tick 1 is set in week 2 and should be ticked on or before the session in week 4. If you fail a tick, you must book a new ticking slot once you believe you have fixed the problems. Multiple ticks can be gathered in a single week if there are spare slots available, but please be considerate to your peers and do not block-book slots in advance.

Other (non-assessable) exercises are presented in the course notes. These should be attempted, especially by those who find the exercises in this document too easy.

Note: please report any errors in these problems to `lp15@cam.ac.uk`.

L.C. Paulson, *Course Lecturer*

Robert Harle, *Part IA Coordinator*

October 2014

Exercise 1: Recursive Functions

You may wish to limber up for this exercise by performing numerical calculations using ML. 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.

1. A mathematical formula is often expressed in the form of a *function*. For example, the formula for the area of a triangle, $xy/2$, can be written as the function definition

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

We can declare the analogous function in ML by typing

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

Enter this function definition and demonstrate it by calculating some areas of triangles. Remember to include decimal points! Write `area(3.0, 4.0)`, not `area(3, 4)`.

2. Everybody who writes computer programs should be aware of floating-point rounding errors. Ask ML to evaluate the following expression, and report its response.

```
1.0 - 0.9 - 0.1;
```

3. The two solutions to the equation $ax^2 + bx + c = 0$ are given by the *quadratic formula*,

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

The following ML code computes one of these roots. (Note that `Math.sqrt` refers to a function named `sqrt` in a library called `Math`.)

```
fun rootplus (a, b, c) = (~b + Math.sqrt (b*b-4.0*a*c)) / (2.0*a);
```

Write an ML function to compute the value of $ax^2 + bx + c$ given the four arguments a, b, c and x. (*Hint*: you are likely to need a type constraint to ensure that it returns a value of type `real`.) Use it to check the accuracy of the roots computed by the function `rootplus`, in the following cases: ($a = 1, b = 121, c = 11$); ($a = 1.22, b = 3.34, c = 2.28$).

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

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

4. Write an ML function `facr(n)` to compute $n!$ by recursion, and also a function `faci(n)` to compute $n!$ by iteration (in the sense described in Lecture 2). How do your functions behave when applied to a negative argument?

Exercise 1*: Recursive Functions Continued

Note that although the following problems will not count towards a 'tick', it is a good idea to attempt them before next week's exercise.

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 2: 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);
```

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

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

Recall that, although lists can contain structured data, 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];         [3, (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)` 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(xs,n)` returns the *n*th element of list `xs`, counting the head of the list as element zero.

Exercise 2*: Lists 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 every list in a list of lists:

```
- 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:

```
- concat ([[1],[2,3]],[[],[4,5,6]]);
> [[1],[2,3],[],[4,5,6]] : int list
```

But this is not essential.

Exercise 3: Functions as Arguments and Results; Integer Streams

Note: the second part of this exercise assumes knowledge of Lecture 9: Sequences.

In ML, functions can be given as input and can be returned as results from functions. As coded below, the function `twice` takes the argument `f` and returns an anonymous function that applies `f` twice to *its* argument.

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

Can you see 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 $s(x) = x + 1$ (which 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 $m \times n$ or 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.

Here is a definition of streams (infinite lists) that cannot terminate. Note that the function `from` (which creates the stream of integers starting from a specified value) can be declared exactly the same as with the type `'a seq` presented in the lectures.

```
datatype 'a stream = Cons of 'a * (unit -> 'a stream);
fun from k = Cons(k, fn()=> from(k+1));
```

3. Write a function `nth(s,n)` to return the `n`th element of `s`. For example, `nth(from 1, 100)` should return 100.

4. Make the stream of positive squares (1, 4, 9, ...) and find its 49th element.

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

Exercise 3*: Integer Streams Continued

1. 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.

2. 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.

3. 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`.

4. 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 4: A Tiny Graphics Package

Portable pixmap format (PPM) is an extremely simple image file format, where an image of a specified width w and height h is given by a text file containing h lines, each w pixels across.[1] Each pixel is specified by three integers according to the RGB colour model. Each integer ranges from 0 to 255. Here are a few representative examples:[2]

0	0	0	black	255	0	0	red	0	255	0	green
255	255	0	yellow	0	0	255	blue	0	255	255	cyan
255	0	255	magenta	128	128	128	grey	255	255	255	white

The task is to represent such images using ML arrays and to implement some operations on them. You must provide the following types (and the first two are given to you):

```
type color = int * int * int       (* RGB colour components, 0..255  *)
type xy    = int * int             (* points (x, y) and sizes (w, h) *)
type image
```

1. Implement the following functions:

```
val image     : xy -> color -> image
val size      : image -> xy
val drawPixel : image -> color -> xy -> unit
```

The function `image`, given a dimension $w \times h$, should create an ML array consisting of h elements to represent the rows, where each row is itself an ML array consisting out of w elements of type `color`. Each pixel should be given the supplied colour. Remember that arrays are mutable objects, so each row needs to be a distinct array; you may find the function `Array.tabulate` useful for this.[3]

The function `size` should return the dimensions of the given image as a pair (w, h).

The function `drawPixel`, given a specified `image`, sets the pixel at the given (x, y) position to the specified `color`.

2. Implement a function to write an image to a file. Base it on the following skeleton, which opens the file and writes two lines that must appear before the lines of pixels.

```
fun toPPM image filename =
  let val oc = TextIO.openOut filename
      val (w,h) = size image
  in
     TextIO.output(oc, "P3\n" ^ Int.toString w ^ " " ^ Int.toString h ^
                       "\n255\n");
     ... code to output image rows, one per line ...
     TextIO.closeOut oc
  end;
```

You may find `Array.app` helpful; it applies a given function to every element of an array. Also, here is a function to format integers. Each line of pixels should consist of integers separated by spaces and no other characters.

```
fun format4 i = StringCvt.padLeft #" " 4 (Int.toString i);
```

3. Code a function to draw horizontal or vertical lines of a given colour. Create a simple image. You can view your image, or convert it to another format, using these UNIX commands:

```
display example.PPM
convert example.PPM example.png
```

[1] http://en.wikipedia.org/wiki/Netpbm_format

[2] http://en.wikipedia.org/wiki/Web_colors

[3] http://www.standardml.org/Basis/array.html

Exercise 4*: Line Drawing

Implement a function to draw a line from one pixel position to another.

```
val drawLine : image -> color -> xy -> xy -> unit
```

Use Bresenham's line algorithm,[4] which can be expressed by the following pseudo-code:

```
function line(x0, y0, x1, y1)
  dx := abs(x1-x0)
  dy := abs(y1-y0)
  if x0 < x1 then sx := 1 else sx := -1
  if y0 < y1 then sy := 1 else sy := -1
  err := dx-dy
  loop
    setPixel(x0,y0)
    if x0 = x1 and y0 = y1 exit loop
    e2 := 2*err
    if e2 > -dy then
      err := err - dy
      x0 := x0 + sx
    end if
    if e2 < dx then
      err := err + dx
      y0 := y0 + sy
    end if
  end loop
```

You may be tempted to translate this into ML making heavy use of reference types. It's actually simpler to implement the loop as a recursive function taking three arguments `x`, `y` and `err`. The integer absolute value is available as the function `Int.abs`.

[4] `http://en.wikipedia.org/wiki/Bresenham's_line_algorithm`

Exercise 5: Mandelbrot Set (Vacation Task)

This exercise is compulsory for CST candidates. If you expect to be taking CST Paper 2 in June, then you must complete it.

The aim of this exercise is to draw images containing sections of the Mandelbrot set. It builds upon the previous exercise, in which you implemented a simple library for drawing PPM images.

The submission deadline is 5pm on the first Wednesday of Lent Term. All students will have their ticking session on the following day during the first Java practical of term. The Java practicals will run on every Thursday afternoon from 2pm to 4pm or from 4pm to 6pm; full details will be sent out at the beginning of next term.

Your submission should be placed in the Blue Rack appropriate for your usual Ticker exactly as for earlier ML ticks. Any outstanding ML submissions should be handed in at the same time.

1 Background Information

The Wikipedia article on the Mandelbrot set[5] is a useful guide, although it goes into more depth than is required for this exercise.

You can use Poly/ML, Moscow ML or SML/NJ for this exercise. To use Moscow ML, you must include the following line at the top of your program:

```
load "Math";
```

The function `load` (of type `string -> unit`) takes the name of an external library and makes it available to the Moscow ML runtime.

Various ML systems are already installed on the PWF machines, and you can easily install ML on a personal machine.[6]

2 Exercise

The instructions below will guide you through to the completion of this exercise. You should implement your program in a single source file to which you will add as you progress through the exercise.

1. Write a function `drawAll`, which applies a function to every pixel in an `image`.

```
val drawAll : (xy -> color) -> image -> unit
```

The function passed to `drawAll` will be referred to as the *colouring function* and should have the type `xy -> color`.

This function should take a pair of integers (the coordinates of the current pixel) and return a triple of integers (the RGB value to colour this pixel). Your function `drawAll` should thus take a colouring function and colour each pixel of the image according to the output of the colouring function. Why does the `drawAll` function return `unit` rather than a new image?

2. Add the following function to your program:

```
fun gradient (x,y) =
    (((x div 30) * 30) mod 256, 0, ((y div 30) * 30) mod 256);
```

[5] `http://en.wikipedia.org/wiki/Mandelbrot_set`

[6] `http://www.cl.cam.ac.uk/teaching/current/FoundsCS/usingml.html`

Write a function `gradImage: unit->unit` which uses `drawAll` and creates an image on disk (`gradient.ppm`) of dimensions 640x480 pixels with each pixel value set according to the gradient function.

3. Add the following function to your source code. This function checks to see if the point (x, y) lies within the Mandelbrot set. The argument `maxIter` indicates how many attempts should be made before assuming that the point is in the set.

```
fun mandelbrot maxIter (x,y) =
  let fun solve (a,b) c =
      if c = maxIter then 1.0
      else
        if (a*a + b*b <= 4.0) then
          solve (a*a - b*b + x,2.0*a*b + y) (c+1)
      else (real c)/(real maxIter)
  in
    solve (x,y) 0
  end;
```

The `mandelbrot` function returns the amount of work done as the fraction $c/maxIter$, so 0.0 represents zero iterations and 1.0 represents the maximum number of iterations.

4. Add the following function to your source code. This function selects an RGB colour based on the number of iterations returned by the mandelbrot function.

```
fun chooseColour n =
  let
    val r = round ((Math.cos n) * 255.0)
    val g = round ((Math.cos n) * 255.0)
    val b = round ((Math.sin n) * 255.0)
  in
    (r,g,b)
  end;
```

The implementations of `sin` and `cos` are provided by the `Math` library.

5. The `mandelbrot` function does not operate on pixel values. Instead it operates on the number plane. Our image will cover only a portion of the mandelbrot set and so we will need to convert between pixel values and real numbers. Write a function `rescale` to do this:

```
val rescale : xy -> real*real*real -> xy -> real*real
```

The call `rescale (w,h) (cx,cy,s) (x,y)` should return a tuple (p, q) which is the real number position on the number plane which corresponds to the pixel value (x, y). The tuple (w, h) gives the size of the image in pixels, and the tuple (cx, cy, s) specifies the central point of the image and the size of the window of interest (Figure 1).

The point p is given as follows (you should derive an equation for q yourself)

$$p = s\left(\frac{x}{w} - \frac{1}{2}\right) + c_x$$

6. Write a function `compute` which combines your `rescale` function with `mandelbrot` and `chooseColour` in order to compute the Mandelbrot set of a particular region and save the result to a file on disk.

```
val compute : real*real*real -> unit
```

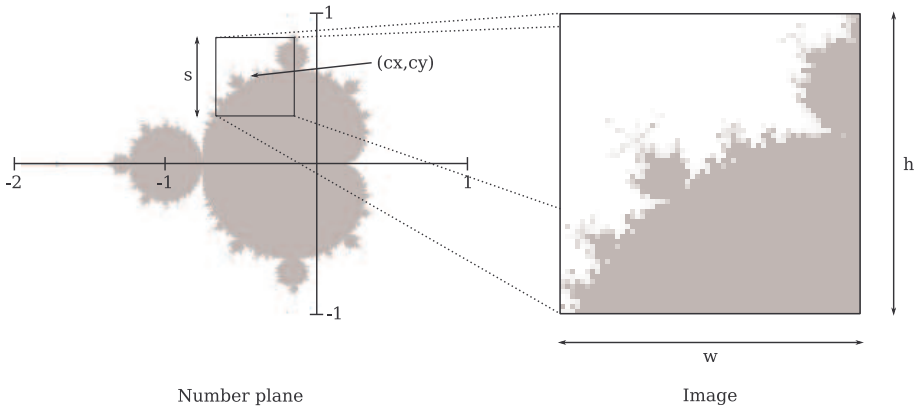

Figure 1: The function rescale converts a pixel value to a point on the number plane.

Here `compute (cx,cy,s)` will write an image to disk (`mandelbrot.ppm`) containing the Mandelbrot set for the chosen region (experiment to find a sensible value for `maxIter`).

7. Draw the Mandelbrot set for $(cx, cy, s) = (-0.74364990, 0.13188204, 0.00073801)$.

Your submission should contain

- the printed source code as outlined above, including a comment with your name and college in the same manner as the previous exercises, and comments answering the questions as requested in the outline above;
- printouts of the gradient fill test and the selected portion of the Mandelbrot set;
- the amount of time you spent on the exercise.