CSC/MAT-220: Lab 4

Due: 10/12/2018

In ML, the primary means of working out an iterative task is through the recursive function. A *recursive function* computes the result of a call by reducing the problem to smaller subproblems until the smaller problem can be solved trivially. The *recursive step* is the simplifying step used to make the problem smaller, and results in the function calling itself until the *base case* (smallest subproblem) has been reached.

In order for a function to call itself, it must have a name by which it can refer to itself. In ML, this is achieved by using a *recursive value binding*. Consider the example below.

```
(* SML Example 1 *)
val rec factorial : int->int =
fn 0 => 1 | n:int => n*factorial(n-1);
```

Note that the above function is evaluating an expression on a case by case bases. This is a powerful feature of ML known as *case analysis*. Each case is separated by a vertical bar, i.e., |. Let us consider the application of case analysis in the evaluation of `factorial(3)`. The first subproblem is evaluating the expression

```
(fn 0=>1 | n:int => n*factorial(n-1))(3).
```

Since the first case is not satisfied and the second case is, we are left with the subproblem

```
3*(fn 0=>1 | n:int => n*factorial(n-1))(2).
```

This pattern continues until the subproblem is trivial

```
3*2*1*(fn 0=>1 | n:int => n*factorial(n-1))(0).
```

Now, the first case is satisfied so the expressions evaluates to 6.

Tail Recursion

Note that the size of our subproblem grows until we reach the base case, at which point there are no more recursive calls and the computation can complete. The growth in the expression's size corresponds directly to a growth in the runtime storage required to record the state of the pending computation. This can be a huge problem with recursive functions, fortunately this can often be resolved using *tail recursion*, where we evaluate additional computations before making the recursive call. Consider the example below.

```
(* SML Example 2 *)
local
        val rec helper : int*int->int =
        fn (0,r:int) => r | (n:int,r:int) => helper(n-1,n*r);
in
        val factorial : int->int =
        fn n:int => helper(n,1);
end
```

Now, the evaluation of `factorial(3)` results in a call to the helper function, with the following evaluation trace:

i. `helper(3,1)`

ii. `helper(2,3)`

iii. `helper(1,6)`

iv. `helper(0,6)`

v. `6`

Note that there is no growth in the size of the expression because there are no pending computations to be resumed upon completion of the recursive call.

Here is another example of a function that would benefit from tail recursion.

```
(* SML Example 3 *)
val rec fib : int->int =
fn 0 => 1 | 1 => 1 | n:int => fib(n-1)+fib(n-2)
```

Let T_n denote the computation time and M_n denote the memory used in computing the nth Fibonacci number via the function from Example 3. We saw in class that both T_n and M_n have solutions of the form

$$c_1\left(\frac{1+\sqrt{5}}{2}\right)^n + c_2\left(\frac{1-\sqrt{5}}{2}\right)^n$$

and therefore grows exponentially with respect to n. We can do much better using tail recursion as shown in the example below.

```
(* SML Example 4*)
local
    val rec helper : int->int*int =
    fn 0 => (1,0) | 1 => (1,1) | n:int =>
    let
        val (a:int,b:int) = helper(n-1)
    in
        (a+b,a)
    end
in
    val fib : int->int =
    fn n:int =>
    let
        val (a:int,_) = helper(n)
    in
        a
    end
end
```

Mutual Recursion

It is often useful to define two functions simultaneously, each of which depends on the other. As a simple example, consider the following functions *even* and *odd*.

```
(* SML Example 5 *)
val rec even : int->bool =
fn 0 => true | n:int => odd(n-1)
and odd : int->bool =
fn 0 => false | n:int => even(n-1)
```

Below is a more complicated example.

```
(* SML Example 6 *)
val match : int list -> bool =
fn x:int list =>
let
    val rec need_one : int list->bool =
    fn  [] => true
    |   1::x' => need_two(x')
    |   _ => false
    and need_two : int list->bool =
    fn  [] => false
    |   2::x' => need_one(x')
    |   _ => false
in
    need_one(x)
end
```

This is a little "state machine" for deciding if a list of ints alternates between 1 and 2, not ending with a 1.

Assignment

At the top of your file include a commented line with your name, lab number, and date. In addition, be sure to comment generously throughout so I can follow your steps. Finally, when you are done upload your source code to Dropbox. The file you you turn in should be labeled as follows: *yourname_labnumber.sml*.

I. Analyze the function *fib* declared in Example 4; determine a solution to T_n and M_n.

II. Perform an evaluation trace for the function *match* in Example 6 on the list $[1, 2, 1]$ and $[1, 2, 1, 2]$.

III. Below is pseudo code for the Euclidean algorithm which is an efficient way to compute the greatest common divisor of two integers.

```
function euclid(a,b)
        while b != 0
                t = b;
                b = a mod b;
                a = t;
        return a;
```

Write a recursive function that performs the Euclidean algorithm in SML. Use your function to compute the greatest common divisor of the integers 462 and 1071.

IV. Finish the following code for Pascal's triangle

```
local
    val rec helper : (int list)*(int list)->int list =
in
    val rec pascal : int->int list =
    fn  0 => [1]
    |   1 => [1,1]
    |   n:int =>
end
```

The *helper* function is intended to perform element-wise addition on two int lists of the same length, and the *pascal* function is intended to compute the nth row of Pascal's triangle. Use the pascal function to evaluate $\binom{30}{10}$. Note that you can use the List structure to find a function for evaluating a specific element in the list.