CS109A Assignment #3
Due 3:15 PM, 2/7/96

Note: Please remember to separate the written homework from the programming problems and hand in each separately.

Written Homework

H1. (10 points) What is wrong with the following inductive "proof"?

$S(n)$: in a field with n horses, all have the same color

Basis: $n = 1$. If there is one horse, then surely all horses have the same color.

Induction: Suppose $S(n)$ is true for some $n \geq 1$ and consider a field with $n + 1$ horses. Remove one horse, say H_1, from the field. Now there are n horses, and by the inductive hypothesis, all have the same color. Next, bring back H_1 and remove another horse, say H_2. Again there are n horses, and all have the same color by the inductive hypothesis. Let H be some horse that has remained in the field during the time that H_1 and then H_2 were removed. All the horses except perhaps H_1 are known to have the same color. Because H and H_1 were in the field when H_2 was removed, we know that H and H_1 also have the same color. Thus, H_1's color is the same as that of all the other horses, and all $n + 1$ horses have the same color.

H2. (10 points) Define recursively the set of even-parity strings, by induction on the length of the string. *Hint*: It helps to define two concepts simultaneously, both the even-parity strings and the odd-parity strings. (Exercise 2.6.7, p. 69, FCS)

H3. (15 points) For each of the following, determine whether $f(n)$ is $O(g(n))$. Prove your answer; either give witnesses or argue that no witnesses exist.

a) $f(n) = n^2$; $g(n) = 1000n$.

b) $f(n) = 6^n$; $g(n) = 2^n$.

c) $f(n) = \log_2 n$; $g(n) = \sqrt{n}$.

H4. (15 points) The C function on the next page computes and prints all prime numbers less than or equal to n. Draw the structure tree for the function, and give a big-oh upper bound on running time for each node of the structure tree.

```
void print_primes(int n)
    {
        int i, candidate, is_prime;
        for (candidate=2; candidate<n; candidate++) {
            is_prime=TRUE;
            for (i=2; i*i<=candidate; i++)
                if (candidate%i==0)
                    is_prime=FALSE;
            if (is_prime)
                printf("%d\n",candidate);
        }
    }
```

Programming Problems

These problems are sufficiently complex that we expect appropriate comments to be added to your code. While we won't go crazy if you fail to comment a 3-line function, especially if the function name is a clue to what it does, we do expect that large functions will have comments explaining your logic, both for this assignment and in the future.

P1. (10 points) On p. 39 of EMLP is a function `reverse` that reverses a list. However, it is a bad design; it takes time proportional to n^2 (or $O(n^2)$, as we shall soon learn to say) to reverse a list of length n. There is a better way to reverse lists, that takes time proportional to the length of the list and doesn't use the `@` operator, just cons (`::`). Write this function in ML. *Hint*: look at the class problem of converting integers to binary in the proper order to get an idea of the proper order of the recursive call and the "extra stuff." Show the result of your function on `nil` and `["a", "b", "c"]`.

P2. (10 points) Write a function `printBinInt(x)` that nicely prints a binary integer x. For example, on binary integer `(4,[1,0,1,1])` you should print something like

```
Binary Integer: length = 4; bits = 1101
```

and then skip to the next line. Remember, the list needs to be printed in reverse order, for which you can use your function from P1. Show your output on binary integers `(0,nil)` and `(4,[1,0,1,1])`.

P3. (30 points) Write functions `add(x,y)` and `sub(x,y)` that perform, respectively, addition and subtraction of binary integers. However, since we have no representation for negative integers, you may assume that the first argument of `sub` is at least as great as the second argument. You should declare and raise suitable exceptions if the data is incorrect, e.g. the first component of a binary integer doesn't equal the length of its second component, the second component has an integer other than 0 or 1, or the second argument of `sub` exceeds the first. Show the result of applying `add` and `sub` to the following pairs of arguments:

i) `genBinInt(6)` and `(5,[1,1,0,1,1])`, where `genBinInt` is your function from P2 of Assignment 2.

ii) `(4,[0,1,0,1])` and `(3,[1,1,1])`.

iii) `genBinInt(6)` and `(1,[0])`.

This problem is a challenge, regardless of the language you write it in. Here are three hints that may help you.

a) The heart of the algorithms is to handle a carry or a borrow. That is, for addition, we need to solve the more general problem of adding two binary integers with a carry c into the low-order place. We can thus write a recursive function `addc(x,y,c)` that computes the sum of binary integers x and y, plus the carry c, which is either 0 or 1. It is then a simple matter to have `add(x,y)` call `addc(x,y,0)`. Similarly, solve the subtraction problem by writing a function `subb(x,y,b)` that computes the difference of x and y, minus a "borrow" b, which is either 0 or 1.

b) The inductive step of the recursion for `addc` or `subb` involves 8 cases in principle: these correspond to the 8 choices for the carry (or borrow) and the low-order bits of the two binary integers. The latter are conveniently the heads of the lists in the second components, by the way. However, if you analyze the situation, you can simplify the 8 cases considerably.

c) These functions are complex enough that you might not get them right, even though SML accepts what you have written. To help with debugging, here is a general strategy for tracing the action of a recursive ML function. We take the function we want, say `addc`, and define it mutually recursively with a "shadow function," say `addc1`. (See pp. 44–45 of EMLP for how mutually recursive functions are declared.) Whenever `addc` would call itself recursively, say `addc(x,y,c)`, it instead calls `addc1(x,y,c)`. Then `addc1` prints out its arguments and calls `addc` with the same arguments. As a result, there is no change to what `addc` does, except that at every recursive call, we get to see what the arguments of that call were. When you are satisfied that `addc` is correct, "comment out" all of `addc1` except for the call to `addc`.