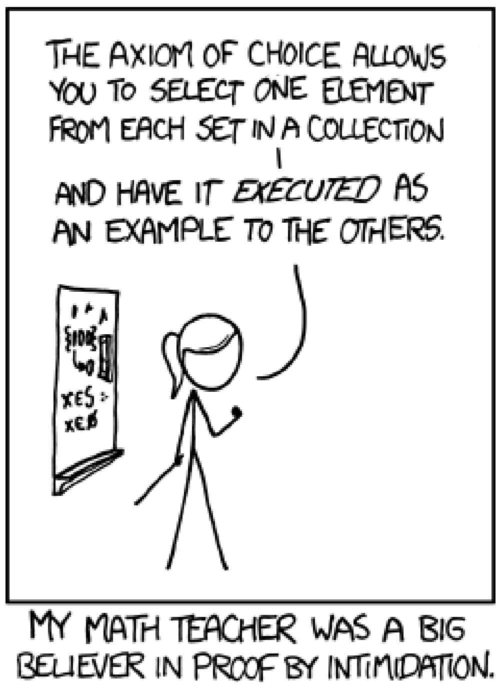
CS52 - Assignment 6

Due Monday 11/2 at 11:59pm



`https://xkcd.com/982/`

Your work for this assignment is to be submitted on paper.[1] *Please write legibly.* I strongly encourage you to figure out your proves on a piece of scratch paper first and then write them a second time, neatly, for your final submission.

To turn it in, you may give it directly Dr. Dave or slide it under his office door.

[1]For those of you who would like to use LaTeX, I have included a source file if you'd like to utilize it, though you do not have to.

Some “Facts”

You may use the following definitions and facts in your proofs below. Please refer to them by number to justify your proofs, e.g. “by Fact 1”.

1. `[]@vl = vl`
2. `ul@[] = ul`
3. `(ul@vl)@wl = ul@(vl@wl)`
4. `[u]@us = u::us`
5. `nrev [] = []`
   `nrev (u::us) = (nrev us) @ [u]`
6. `revApp ul [] = ul`
   `revApp ul (v::vs) = revApp (v::ul) vs`

The Proof is in the Pudding

1. [**2 points**] Multiples of fun

   (a) [**1 point**] Suppose that we evaluate the degree k polynomial $p_0+p_1X+p_2X^2+\ldots+p_kX^k$ using a “brute-force” approach by taking every possible opportunity to multiply. For example, for $k=3$, we would compute

   $$p_0 + p_1 \cdot X + p_2 \cdot X \cdot X + p_3 \cdot X \cdot X \cdot X,$$

   so that there would be a total of 6 multiplications.

   In terms of k, how many multiplications would be required to evaluate a k-degree polynomial using the brute-force method? Give an informal justification; a proof is not necessary.

   (b) [**1 point**] We saw Horner’s method on assignment 2. It is a recursive technique for evaluating polynomials. As above, no multiplications are required for a zero-degree polynomial. For a k-degree polynomial with $k > 0$, we write

   $$p_0 + X \cdot \Big(\underbrace{p_1 + p_2X + \ldots + p_kX^{k-1}}_{(k-1)\text{-degree polynomial}}\Big).$$

   How many multiplications are required when using Horner’s method to evaluate a k-degree polynomial? Again, just give an informal justification.

2. [**4 points**] Prove, by list induction on `vl`, that `revApp ul vl = (nrev vl) @ ul`.

   Notice the special case of this result, that `revApp [] vl = nrev vl`, proves that our two implementations of `rev` really do compute the same results.

3. [**4 points**] Induction into the hall of fun

   (a) [**1.5 points**] We want to count the number of times the operator `::` is used when `nrev` reverses a list of length k.

   Write a recursive relation called `nCons` (like $count_0(k)$ and $count_1(k)$ that we did in class for `uniquify` variants) for this number as a function of k.

   Clarification: You are to count the number of times that `::` is used to construct a list, and *not* the times that the operator appears in the pattern-matching on the left of the equals signs in the definition. You may use without proof the fact that the number of `::` operations used in computing `ul@vl` is exactly the length of `ul`. Do not forget that `[u]` is an abbreviation for `u::[]`.

   (b) [**2.5 points**] Prove that $\texttt{nCons}(k) = (k+1)k/2$.

4. [**4 points**] Induction cookware should not be used for proofs

   (a) [**1.5 points**] Write a recurrence relation called `raCons` for the number of `::` operations used in computing `revApp ul vl` with respect to k, the length of `vl`.

   (b) [**2.5 points**] Prove that $\texttt{raCons}(k) = k$.

5. [**4 points**] Binary induction

   Suppose that b is a positive integer, and e, *when written out in binary*, is a k-bit number.

   (a) [**1.5 points**] What is the maximum number of multiplications required to compute b^e when b^e is computed by the formula below? Express the result in terms of k, *not* e. Give an informal justification; a proof is not necessary.

$$b^e = \begin{cases} 1 & \text{if } e = 0 \text{, and} \\ b \cdot b^{e-1} & \text{otherwise.} \end{cases}$$

   (b) [**1.5 points**] Again in terms of k, what is the maximum number of multiplications required to compute b^e when b^e is computed using the alternative formula below? Give an informal justification; a proof is not necessary. (We will use this formula when we return to `cs52Int` in a future assignment.)

$$b^e = \begin{cases} 1 & \text{if } e = 0, \\ \text{square}(b^{e/2}) & \text{if } 1 < e \text{ and } e \text{ is even, and} \\ b \cdot \text{square}(b^{e/2}) & \text{otherwise.} \end{cases}$$

   Here, $e/2$ is integer division, with truncation. In binary, it amounts to removing the least significant bit, so that $e/2$ is exactly one bit shorter than e. The square function requires one multiplication.

   (c) [**1 point**] Assume that e is a 100-bit number and that a computer can do 10^{11} multiplications a second. (That is fast, but not unreasonable, for the current crop of computers.) For each of the two techniques from part a and part b, estimate how long it would take to compute b^e?

   It will be helpful to use the approximation $2^{10} \approx 10^3$ (remember the party trick?). Also, one year is about $3 \cdot 10^7$ seconds.

6. [**4 points**] Consider the following SML definition of binary trees and an associated "reflection" function.

```
datatype 'a binTree =
     Empty
   | Node of 'a binTree * 'a * 'a binTree;

fun mirror Empty = Empty
  | mirror (Node (lt, g, rt)) =
        Node (mirror rt, g, mirror lt);
```

Prove by induction on the datatype `binTree` that

`mirror (mirror onTheWall) = onTheWall`.

When you're done

- Make sure your name and assignment number are at the top of the paper.
- Make sure that each problem is clearly denoted.
- Make sure that your handwriting, etc. is *very clear*. If a particular problem is very messy please rewrite it on a separate sheet of paper more clearly.
- For each induction proof make sure you have followed *exactly* the induction format discussed in class.
- Make sure that you justify every step in your proofs.

To turn it in, you may give it directly Dr. Dave or slide it under his office door.