CS235 Languages and Automata **Handout # 7**
Prof. Lyn Turbak **October 16, 2007**
Wellesley College

Problem Set 2 Solutions

Problem 6 from PS1 [45 points]

You were asked to determine the sizes of various sets involving the following sets:

$$Bool = \{T,\ F\} \qquad Sign = \{-,\ 0,\ +\} \qquad Nat = \{0,\ 1,\ 2,\ 3,\ \ldots\}$$

Recall that for finite sets A and B:

$$|A \times B| = |A| \times |B| \qquad |A \to B| = |B|^{|A|} \qquad |\mathcal{P}(A)| = 2^{|A|}$$

a. $|Bool \times Bool| = |Bool| \times |Bool| = 2 \times 2 = 4$. $(Bool \times Bool = \{(T,T),(T,F),(F,T),(F,F)\}.)$

b. $|Bool \times Sign| = |Bool| \times |Sign| = 2 \times 3 = 6$.
$(Bool \times Sign = \{(T,-),(T,0),(T,+),(F,-),(F,0),(F,+)\}.)$

c. $|Sign \times Bool| = |Sign| \times |Bool| = 3 \times 2 = 6$.
$(Bool \times Sign = \{(-,T),(-,F),(0,T),(0,F),(+,T),(+,F)\}.)$

d. $|Sign \times Sign| = |Sign| \times |Bool| = 3 \times 3 = 9$.
$(Bool \times Sign = \{(-,-),(-,0),(-,+),(0,-),(0,0),(0,+),(+,-),(+,0),(+,+)\})$

e. $|Bool \times Nat|$ is countably infinite. To show this, we must describe a bijection $f \in (Nat \to (Bool \times Nat))$. There are many possible bijections; here we describe one.

Informally, f maps the ith even natural (where the first even natural is 0) to $(T, i-1)$ and the ith odd natural (where the first odd natural is 1) to $(F, i-1)$:

0	1	2	3	4	5	6	7	...
↕	↕	↕	↕	↕	↕	↕	↕	
$(T,0)$	$(F,0)$	$(T,1)$	$(F,1)$	$(T,2)$	$(F,2)$	$(T,3)$	$(F,3)$	...

Most students described something like this, but many had trouble formalizing it properly.

To formalize this informal description, we must flesh out four steps:

1. <u>define f:</u> In traditional function notation:

$$f(n) = \begin{cases} (T, n/2) & \text{if } n \in Nat \text{ is even} \\ (F, (n-1)/2) & \text{if } n \in Nat \text{ is odd} \end{cases}$$

   In the functions-as-triples notation we've been using this semester:

$$f = (Nat, Bool \times Nat, \cup_{n \in Nat}\{(2n, (T,n)), (2n+1, (F,n))\})$$

2. <u>define f^{-1}:</u>
   In traditional function notation:

$$f^{-1}(\ (b,n)\ ) = \begin{cases} 2n & \text{if } b \in Bool \text{ is } T \\ 2n+1 & \text{if } b \in Bool \text{ is } F \end{cases}$$

   In the functions-as-triples notation:

$$f^{-1} = (Bool \times Nat, Nat, \cup_{n \in Nat}\{((T,n), 2n), ((F,n), 2n+1)\})$$

3. show that $(f^{-1} \circ f) = id_{Nat}$: Using the traditional function notation, we have two cases:

   $n \in Nat$ is even:

$$\begin{array}{rcll} (f^{-1} \circ f)(n) & = & f^{-1}(f(n)) & \text{by defn. of composition} \\ & = & f^{-1}(\ (T, n/2)\ ) & \text{by defn. of } f \text{ for evens} \\ & = & 2(n/2) & \text{by defn. of } f^{-1} \\ & = & n & \text{by arithmetic for even } n \\ & = & id_{Nat}(n) & \text{by defn. of } id_{Nat} \end{array}$$

   $n \in Nat$ is odd:

$$\begin{array}{rcll} (f^{-1} \circ f)(n) & = & f^{-1}(f(n)) & \text{by defn. of composition} \\ & = & f^{-1}(2n) & \text{by defn. of } f \text{ for odds} \\ & = & 2((n-1)/2) + 1 & \text{by defn. of } f^{-1} \\ & = & n & \text{by arithmetic for odd } n \\ & = & id_{Nat}(n) & \text{by defn. of } id_{Nat} \end{array}$$

   In the functions-as-triples notation, it is more obvious that $(f^{-1} \circ f) = id_{Nat}$ because for each n in Nat, the pairs $\{(2n, (T, n)), (2n+1, (F, n))\}$ are in the graph of f and the pairs $\{((T, n), 2n), ((F, n), 2n+1)\}$ are in the graph of f^{-1}. So:

   - f maps any natural of the form $2n$ (an even natural) to (T, n) f^{-1} maps (T, n) back to $2n$.
   - f maps any natural of the form $2n+1$ (an odd natural) to (F, n) f^{-1} maps (F, n) back to $2n+1$.

4. show that $(f \circ f^{-1}) = id_{Bool \times Nat}$: Using the traditional function notation, we have two cases:

   The input pair has the form (T, n):

$$\begin{array}{rcll} (f \circ f^{-1})(\ (T, n)\ ) & = & f(f^{-1}(\ (T, n)\ )) & \text{by defn. of composition} \\ & = & f(2n) & \text{by defn. of } f^{-1} \\ & = & (T, (2n)/2) & \text{by defn. of } f \text{ for evens} \\ & = & (T, n) & \text{by arithmetic} \\ & = & id_{Nat \times Bool}(\ (T, n)\ ) & \text{by defn. of } id_{Nat \times Bool} \end{array}$$

   The input pair has the form (F, n):

$$\begin{array}{rcll} (f \circ f^{-1})(\ (F, n)\ ) & = & f(f^{-1}(\ (F, n)\ )) & \text{by defn. of composition} \\ & = & f(2n+1) & \text{by defn. of } f^{-1} \\ & = & (F, ((2n+1)-1)/2) & \text{by defn. of } f \text{ for odds} \\ & = & (F, n) & \text{by arithmetic} \\ & = & id_{Nat \times Bool}(\ (F, n)\ ) & \text{by defn. of } id_{Nat \times Bool} \end{array}$$

   As in the previous round-trip proof, in the functions-as-triples notation, it is more obvious that $(f \circ f^{-1}) = id_{Nat \times Bool}$ because for each n in Nat, the pairs $\{((T, n), 2n), ((F, n), 2n+1)\}$ are in the graph of f^{-1} and the pairs $\{(2n, (T, n)), (2n+1, (F, n))\}$ are in the graph of f. So:

   - f^{-1} maps any pair (T, n) to $2n$ and f maps it back to (T, n).
   - f^{-1} maps any pair (T, n) to $2n+1$ and f maps it back to (F, n).

f. $|Bool \to Bool| = |Bool|^{|Bool|} = 2^2 = 4$.

$$\begin{array}{l} (Bool \to Bool = \{(Bool, Bool, \{(T, T), (F, T)\}), \\ \qquad\qquad (Bool, Bool, \{(T, T), (F, F)\}), \\ \qquad\qquad (Bool, Bool, \{(T, F), (F, T)\}), \\ \qquad\qquad (Bool, Bool, \{(T, F), (F, T)\})\}.) \end{array}$$

g. $|Bool \to Sign| = |Bool|^{|Sign|} = 3^2 = 9$.

$(Bool \to Sign =$
$\{(Bool, Sign, \{(T,-),(F,-)\}), \quad (Bool, Sign, \{(T,-),(F,0)\}), \quad (Bool, Sign, \{(T,-),(F,+)\}),$
$(Bool, Sign, \{(T,0),(F,-)\}), \quad (Bool, Sign, \{(T,0),(F,0)\}), \quad (Bool, Sign, \{(T,0),(F,+)\}),$
$(Bool, Sign, \{(T,+),(F,-)\}), \quad (Bool, Sign, \{(T,+),(F,0)\}), \quad (Bool, Sign, \{(T,+),(F,+)\})\}.)$

h. $|Sign \to Bool| = |Sign|^{|Bool|} = 2^3 = 8$.

$(Sign \to Bool =$
$\{(Sign, Bool, \{(-,T),(0,T),(+,T)\}), \quad (Sign, Bool, \{(-,T),(0,T),(+,F)\}),$
$(Sign, Bool, \{(-,T),(0,F),(+,T)\}), \quad (Sign, Bool, \{(-,T),(0,F),(+,F)\}),$
$(Sign, Bool, \{(-,F),(0,T),(+,T)\}), \quad (Sign, Bool, \{(-,F),(0,T),(+,F)\}),$
$(Sign, Bool, \{(-,F),(0,F),(+,T)\}), \quad (Sign, Bool, \{(-,F),(0,F),(+,F)\})\}.)$

i. $|Sign \to Sign| = |Sign|^{|Sign|} = 3^3 = 27$.

$(Sign \to Sign =$
$\{(Sign, Bool, \{(-,-),(0,-),(+,-)\}), \quad (Sign, Bool, \{(-,-),(0,-),(+,0)\}), \quad (Sign, Bool, \{(-,-),(0,-),(+,+)\}),$
$(Sign, Bool, \{(-,-),(0,0),(+,-)\}), \quad (Sign, Bool, \{(-,-),(0,0),(+,0)\}), \quad (Sign, Bool, \{(-,-),(0,0),(+,+)\}),$
$(Sign, Bool, \{(-,-),(0,+),(+,-)\}), \quad (Sign, Bool, \{(-,-),(0,+),(+,0)\}), \quad (Sign, Bool, \{(-,-),(0,+),(+,+)\}),$
$(Sign, Bool, \{(-,0),(0,-),(+,-)\}), \quad (Sign, Bool, \{(-,0),(0,-),(+,0)\}), \quad (Sign, Bool, \{(-,0),(0,-),(+,+)\}),$
$(Sign, Bool, \{(-,0),(0,0),(+,-)\}), \quad (Sign, Bool, \{(-,0),(0,0),(+,0)\}), \quad (Sign, Bool, \{(-,0),(0,0),(+,+)\}),$
$(Sign, Bool, \{(-,0),(0,+),(+,-)\}), \quad (Sign, Bool, \{(-,0),(0,+),(+,0)\}), \quad (Sign, Bool, \{(-,0),(0,+),(+,+)\}),$
$(Sign, Bool, \{(-,+),(0,-),(+,-)\}), \quad (Sign, Bool, \{(-,+),(0,-),(+,0)\}), \quad (Sign, Bool, \{(-,+),(0,-),(+,+)\}),$
$(Sign, Bool, \{(-,+),(0,0),(+,-)\}), \quad (Sign, Bool, \{(-,+),(0,0),(+,0)\}), \quad (Sign, Bool, \{(-,+),(0,0),(+,+)\}),$
$(Sign, Bool, \{(-,+),(0,+),(+,-)\}), \quad (Sign, Bool, \{(-,+),(0,+),(+,0)\}), \quad (Sign, Bool, \{(-,+),(0,+),(+,+)\})\}.)$

j. $Bool \to Nat$ is countably infinite. Intuitively, this is true because every element of $Bool \to Nat$ has the form

$$(Bool, Nat, \{(T, n_1), (F, n_2)\})$$

where n_1 and n_2 are natural numbers. So each function in $Bool \to Nat$ is effectively a pair of naturals, and $Bool \to Nat$ is effectively the same set as $Nat \times Nat$, which we know from lecture is countable.

To show this formally, we need to establish a bijection g between $Nat \times Nat$ and $Bool \to Nat$. Since we already know there is a bijection f between Nat and $Nat \times Nat$, and the composition of bijections g and f is a bijection, this would establish a bijection from Nat to $Bool \to Nat$. From this, the countability of $Bool \to Nat$ follows.

We will now flesh out the four steps of bijectionhood:

1. define g: In the functions-as-triples notation:

$$g = (Nat \times Nat, Bool \to Nat, \{((n_1, n_2), (Bool, Nat, \{(T, n_1), (F, n_2)\})) \mid n_1, n_2 \in Nat\})$$

2. define g^{-1}: In the functions-as-triples notation:

$$g^{-1} = (Bool \to Nat, Nat \times Nat, \{(h, (h(T), h(F))) \mid h \in Bool \to Nat\})$$

3. show that $(g^{-1} \circ g) = id_{Nat \times Nat}$:

$$\begin{array}{lcll} (g^{-1} \circ g)((n_1, n_2)) & = & g^{-1}(g((n_1, n_2))) & \text{by defn. of composition} \\ & = & f^{-1}((Bool, Nat, \{(T, n_1), (F, n_2)\})) & \text{by defn. of } g \\ & = & (n_1, n_2) & \text{by defn. of } g^{-1} \\ & = & id_{Nat \times Nat}((n_1, n_2)) & \text{by defn. of } id_{Nat \times Nat} \end{array}$$

4. show that $(g \circ g^{-1}) = id_{Bool \to Nat}$: For any $x, y \in Nat$, let $h_{x,y}$ be shorthand for the function $(Bool, Nat, \{(T, x), (F, y)\})$.

$$\begin{array}{rcll} (g \circ g^{-1})(h_{x,y}) & = & g(g^{-1}(h_{x,y})) & \text{by defn. of composition} \\ & = & g((x,y)) & \text{by defn. of } g^{-1} \\ & = & (Bool, Nat, \{(T,x),(F,y)\}) & \text{by defn. of } g \\ & = & h_{x,y} & \text{by defn. of } h_{x,y} \\ & = & id_{Nat \to Bool}(h_{x,y}) & \text{by defn. of } id_{Nat \to Nat} \end{array}$$

k. $|Nat \to Bool|$ is uncountably infinite. One way to see this is that PS1 Problem 8 proves that there is a bijection between $Nat \to Bool$ and $\mathcal{P}(Nat)$. Since $\mathcal{P}(Nat)$ is uncountably infinite (see part **n**), the set $Nat \to Bool$ must be uncountably infinite as well.

But you were asked to give a diagonalization proof for uncountably infinite sets. Here is such a proof for $Nat \to Bool$:

> *Proof by Diagonalization (a form of Proof by Contradiction)* Towards a contradiction, assume that $Nat \to Bool$ is countably infinite. Then there is a bijection $f \in (Nat \to (Nat \to Bool))$. I.e., f is a function that maps natural numbers to predicates over the natural numbers. For example, $f(0)$ might be the "is even" predicate, $f(1)$ might be the "is between 2 and 4" predicate, $f(2)$ might be the "is everywhere false" predicate, $f(3)$ might be the "is prime" predicate, and so on.
>
> We can depict the bijection f as a two-dimensional table in which the rth row (starting at $r = 0$) denotes the predicate returned by $f(r)$ and the cth column (starting at $c = 0$) of the rth row denotes the truth value $(f(r))(c)$. I.e., if p is the predicate denoted by $f(r)$, then $(f(r))(c) = p(c) =$ T or F, as determined by p. Here is what a sample table might look like:
>
> | | 0 | 1 | 2 | 3 | 4 | 5 | $\cdots$ |
> |---|---|---|---|---|---|---|---|
> | $f(0)$ | **T** | F | T | F | T | F | $\cdots$ |
> | $f(1)$ | F | **F** | T | T | T | F | $\cdots$ |
> | $f(2)$ | F | F | **F** | F | F | F | $\cdots$ |
> | $f(3)$ | F | F | T | **T** | F | T | $\cdots$ |
> | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ |
>
> Now consider the predicate p_{diag} which is defined by negating the booleans in the diagonal of the table:
>
> $$p_{diag} = (Nat, Bool, \{(n, \neg(f(n))(n)) \mid n \in Nat\})$$
>
> By construction, p_{diag} cannot be equal to $f(n)$ for any $n \in Nat$, because $p_{diag}(n) \neq (f(n))(n)$. So p_{diag} is not in the range of f, and f cannot be a bijection as assumed.
>
> We conclude that there is no bijection from Nat to $Nat \to Bool$, meaning that $Nat \to Bool$ is uncountably infinite.

l. $|\mathcal{P}(Bool)| = 2^{|Bool|} = 2^2 = 4$. $(\mathcal{P}(Bool) = \{\{\}, \{T\}, \{F\}, \{T, F\}\}.)$

m. $|\mathcal{P}(Sign)| = 2^{|Sign|} = 2{\times}3 = 8$. $(\mathcal{P}(Sign) = \{\{\}, \{-\}, \{0\}, \{+\}, \{-, 0\}, \{-, +\}, \{0, +\}, \{-, 0, +\}\}.)$

n. $|\mathcal{P}(Nat)|$ = uncountably infinite, by the diagonalization proof given in Stoughton.

Problem 9 from PS1 [15 points]

Use induction to prove the following fact: the sum of the first k odd natural numbers is k^2.

Proof: The ith odd natural number (starting at index 1) is $2i-1$. Let $S(k)$ stand for the sum of the first k odd natural numbers. I.e.:

$$S(k) = \sum_{i=1}^{k}(2i-1)$$

We wish to prove following proposition: for all $k \in Nat, S(k) = k^2$.

(Basis Step) For $k = 0$, $S(0) = \sum_{i=1}^{0}(2i-1) = 0$ (because an empty sum is 0) $= 0^2$. Note that it is insufficient to use $k = 1$ as the basis step, since this will not cover the case where $k = 0$.

(Inductive Step) Suppose $k \geq 1$. We can proceed using one of the following two approaches:

1. Approach 1: Assume the induction hypothesis (IH) $S(k) = k^2$, and prove that $S(k+1) = (k+1)^2$:

$S(k+1)$	$=$	$\sum_{i=1}^{k+1}(2i-1)$	by definition of S
	$=$	$\sum_{i=1}^{k}(2i-1) + (2(k+1)-1)$	by definition of $\sum$
	$=$	$S(k) + (2k+1$	by definition of S and arithmetic
	$=$	$k^2 + 2k + 1$	by IH
	$=$	$(k+1)^2$	by arithmetic

2. Approach 2: Assume the induction hypothesis (IH) $S(k-1) = (k-1)^2$, and prove that $S(k) = k^2$:

$S(k)$	$=$	$\sum_{i=1}^{k}(2i-1)$	by definition of S
	$=$	$\sum_{i=1}^{k-1}(2i-1) + (2k-1)$	by definition of $\sum$
	$=$	$S(k-1) + (2k-1)$	by definition of S
	$=$	$(k-1)^2 + (2k-1)$	by IH
	$=$	$(k^2 - 2k + 1) + (2k-1)$	by arithmetic
	$=$	k^2	by arithmetic

Important Note: As shown above, in a proof by induction, the induction hypothesis should always be stated explicitly in the inductive step.

Problem 10 from PS1 [15 points]

Use strong induction to prove the following fact: for all $n \in Nat$, if $n \geq 8$, then there are $j, k \in Nat$ such that $n = 3j + 5k$.

Proof:

(Basis Step) For this problem, there are *three* base cases:

1. $n = 8$: $8 = 3 \cdot 1 + 5 \cdot 1$, so $j = 1$ and $k = 1$.
2. $n = 9$: $9 = 3 \cdot 3 + 5 \cdot 0$, so $j = 3$ and $k = 0$.
3. $n = 10$: $10 = 3 \cdot 0 + 5 \cdot 2$, so $j = 0$ and $k = 2$.

(Inductive Step) Suppose $n \geq 11$. Assume the following induction hypothesis (IH):

for all $i \in Nat$ such that $8 \leq i < n$, there are $j', k' \in Nat$ such that $i = 3j' + 5k'$.

Because it assumes that the desired property holds for *all* integers between 8 and $n-1$, this is an induction hypothesis for strong induction. (Regular induction would assume the property holds only for $n-1$.)

We wish to show that there are $j, k \in Nat$ such that $n = 3j + 5k$:

$$\begin{array}{rcll} n & = & 3 + (n-3) & \text{by arithmetic} \\ & = & 3 + (3j' + 5k') & \text{by IH (since for } n \geq 11,\ 8 \leq (n-3) < n) \\ & = & 3(j'+1) + 5k' & \text{by arithmetic} \end{array}$$

So the j and k we seek are $j = j' + 1$ and $k = k'$.

Problem 11 from PS1 [10 points]

Find the error in the following proof that all horses are the same color.

Claim: *In any set of n horses, all horses are the same color.*

Proof: *By induction on n*

Basis Step ($n = 1$): In any set containing just one horse, all horses clearly are the same color.

Induction step ($n > 1$): Assume that the claim is true for $n-1$ (this is the induction hypothesis) and prove that it is true for n. Take any set H of n horses. We show that all the horses in this set are the same color. Remove one horse a from H to obtain the set $(H - \{a\})$ with just $n-1$ horses. By the induction hypothesis, all the horses in $(H - \{a\})$ are the same color. Now replace the removed horse a and remove a different one b to obtain the set $(H - \{b\})$. By the same argument, all the horses in $(H - \{b\})$ are the same color. Since both a and b have the same color as all the other horses, a and b must have the same color. So all horses in H have the same color.

Answer: The alleged "proof" fails in the case where $n = 2$. In this case, we have $H = \{a, b\}$ for two distinct horses a and b.

- Removing one horse a from H yields the singleton set $\{b\}$, which does in fact have all horses of the same color. So this reasoning is still correct.
- Removing one horse b from H yields the singleton set $\{a\}$, which does in fact have all horses of the same color. So this reasoning is still correct as well.
- "Since both a and b have the same color as all the other horses, a and b must have the same color." In this case, "all the other horses" is $(H - \{a, b\}) = \{\ \}$, which is the empty set. It is meaningless to say that a and b have the same color as all the members of the empty set, so this does not allow us to conclude that a and b have the same color in this case. **This is the place where the proof breaks down.**

Problem 1 from PS2 [15 points]
a.

```
- extend ("com", ["compare","contrast","commune","compute","think"])
val it = ["compares","communes","computes"] : string list

- caesar (1, "HAL")
val it = "IBM" : string

- twist (#"n", "computation")
val it = "cmainoputto" : string

- transform "Isn't your CS235 class at 9:50 in SCI 104?"
val it = "isntyourcsclassatinsci" : string
```

b. You were asked to give English descriptions for the four functions above. Ideally, such descriptions should be the sort of thing you expect to read in an API — they should describe *what* the function does (its input/output behavior), not *how* it works. If you find yourself explaining the algorithm of the function in English, you're doing the wrong thing.

`extend` takes a prefix string *pre* and a list of strings *strs* and returns the list of strings that result from adding the character `s` to each of the strings in *strs* that begin with *pre*.

`caesar` takes an integer n and a string *str* and returns the string that results from shifting each character's ASCII value up by n (wrapping at ASCII value 256). This is called a "Caesar cipher" after Julius Caes*ar, who used a simple shift by 3.

`twist` takes a character c and a string *str* and returns a string that is the result of concatenating all the characters in *str* lexically $\leq c$ (maintaining their relative position in *str*) with all of the characters in *str* lexically $> c$ (again, maintaining their relative position in *str*).

`twist` takes a string *str* and returns the a string that contains lowercase versions of all the alphabetic characters in *str*, maintaining their relative position in *str*.

c. Below are two nonrecursive definitions of `transform2`. The second is defined purely via function composition, and so doesn't even need a parameter!

```
fun transform2 s =
  String.implode
    (List.map  Char.toLower
      (List.filter Char.isAlpha (String.explode s)))

val transform2 =
  String.implode
  o (List.map  Char.toLower)
  o (List.filter Char.isAlpha)
  o String.explode
```

Problem 2 from PS2 [35 points]
a. Here's one version of the non-tail-recursive `countabRec`, which uses the pattern-matching capability of `let` to decompose the components of the pair returned by the recursive call:

```
fun countabRec s =
  if s = "" then
    (0,0)
  else let val (a,b) = countabRec (butFirst s)
       in if (first s) = "a" then (a+1,b)
          else if (first s) = "b" then (a,b+1)
          else (a,b)
       end
```

An alternative solution is to define a helper function `addPair` that performs elementwise addition on pairs and use this to (possibly) increment the components of the pair returned by the recursive call:

```
fun countabRec2 s =
  if s = "" then
    (0,0)
  else
    addPair (if (first s) = "a" then 1 else 0,
             if (first s) = "b" then 1 else 0)
            (countabRec2 (butFirst s))
```

b. In SML, iteration (looping) is expressed via tail recursion — i.e., a recursive function that has no pending operations to perform after the recursive call. Here is one version of `countabIter` that uses a local tail-recursive `loop` function to process the string:

```
fun countabIter s =
  let val len = String.size s
      fun loop (i,a,b) = (* i = string index, a = count of as, b = count of bs *)
        if i = len then
          (a,b)
        else let val c = String.sub (s, i)
             in if c = #"a" then loop (i+1, a+1, b)
                else if c = #"b" then loop (i+1, a, b+1)
                else loop (i+1, a, b)
             end
  in loop (0,0,0)
```

Here's another version that instead uses the `addPair` function from part **a** to perform the addition:

```
fun countabIter2 s =
  let val len = String.size s
      fun loop (i,pair) = (* i = string index, p = pair of count of as and count of bs *)
        if i = len then
          pair
        else let val c = String.sub (s, i)
             in loop (i+1, addPair pair (if c = #"a" then 1 else 0,
                                         if c = #"b" then 1 else 0))
             end
  in loop (0,(0,0))
  end
```

c. The `isInL1` function is straightforward:

```
fun isInL1 s =
  let val (a,b) = countabRec s (* could use countabIter here instead *)
  in (a mod 2) = 0 orelse (b mod 2) = 1
  end
```

Many students wrote something more complicated for the body of the `let`, such as

```
if (a mod 2) = 0 then
  if (b mod 2) = 0 then false else true
else false
```

When combining boolean expressions, strive to use `not`, `andalso`, and `orelse` rather than `if` expressions involving `true` or `false`.

Problem 3 from PS2 [25 points] Problems like `genStrings` are excellent examples of the divide/conquer/glue problem-solving techniques from CS111 and CS230.

To solve such problems, it helps to consider a concrete example that's not too small (because then it's hard to see patterns) but isn't too large (because then it's too complex). In this case, consider the "problem" of computing

```
genString(3, [#"a", #"b"])
```

whose solution we expect to be

```
["","a","b","aa","ab","ba","bb","aaa","aab","aba","abb","baa","bab","bba", "bbb"]
```

In divide/conquer/glue, we assume that our function simply works on smaller subproblem(s). In this case, the subproblem is

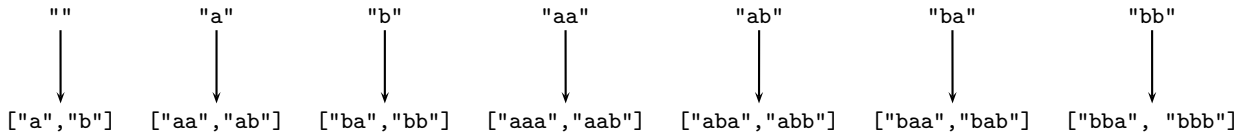
```
genString(2, [#"a", #"b"])
```

whose subsolution is

```
["","a","b","aa","ab","ba","bb"]
```

Now the challenge is to determine a glue step that will transform the subsolution (the solution to the subproblem) to the solution for the whole problem. In this case there are two natural glue steps, each of which leads to a different definition for `genString`:

1. Glue #1: If we postpend each character in the given character list to each element of the subsolution, then we get a list of list of strings containing all of the strings of our solution in the correct order *except* for the empty string at the front:



   This gives rise to the following definition for `genStrings`:

```
fun genStrings1 (n, cs) =
  if n <= 0 then
    [""]
  else
    let fun postpendChar s c = s ^ (String.str c)
        fun postpendAllChars str = map (postpendChar str) cs
    in "" :: (List.concat (List.map postpendAllChars (genStrings1 (n-1, cs))))
    end
```

   `List.concat` is necessary to collect append all the lists in the list of list of strings into a single list of strings. The empty string must be consed onto the front of the result list to complete the solution.

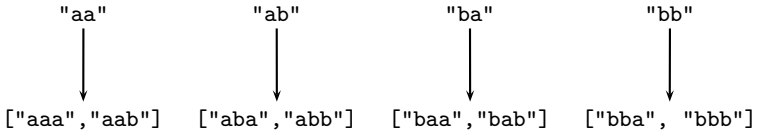
   The local functions `postpendChar` and `postpendAllChars` enhance the readability of the definition, but they are not necessary. Here is the result of "inlining" these local functions:

```
fun genStrings (n, cs) =
  if n <= 0 then
    [""]
  else
    "" :: (List.concat (List.map (fn s => (List.map (fn c => s ^ (String.str c))
                                                    cs))
                                 (genStrings (n-1, cs))))
```

2. Glue #2: Observe that the subsolution `["","a","b","aa","ab","ba","bb"]` is a prefix of the solution, accounting for all but the strings that have length 3:

```
["aaa","aab","aba","abb","baa","bab","bba", "bbb"]
```

Furthemore, observe that these strings of length 3 can be obtained from the strings of length 2 in the subsolution by the same postpending process described for Glue #1:



This reasoning leads to the following alternative solution for `genStrings`:

```
fun genStrings (n, cs) =
  if n <= 0 then
    [""]
  else
    let val subsoln = genStrings (n-1, cs)
        fun postpendChar s c = s ^ (String.str c)
        fun postpendAllChars str = map (postpendChar str) cs
        fun has_size_n_minus_1 str = (String.size str) = (n-1)
     in subsoln @ (List.concat
                   (List.map postpendAllChars
                             (List.filter has_size_n_minus_1 subsoln)))
    end
```

In this approach, the solution is constructed by appending the subsolution to the the list of all the strings that result from postpending all the characters in `cs` onto each of strings in the subsolution that has size `n`-1. It is important to give the name `subsoln` to the result of the expression `genStrings (n-1, cs)` and use this name twice rather than evaluating the expression twice (which would dramatically increase the asymptotic complexity of the running time for the solution).

As above, the local functions enhance readability, but aren't necessary. Here is the same solution without them:

```
fun genStrings (n, cs) =
  if n <= 0 then
    [""]
  else
    let val subsoln = genStrings (n-1, cs)
     in subsoln @ (List.concat
                    (List.map (fn s => (List.map (fn c => s ^ (String.str c))
                                                 cs))
                              (List.filter (fn s => (String.size s) = (n-1)) subsoln)))
    end
```

Problem 4 from PS2 [15 points] In this problem, you were asked to construct a regular expression for the language L_1 consisting of all strings over the alphabet $\{\texttt{a}, \texttt{b}\}$ with an even number of as or an odd number of bs. Note that 0 is an even number, so any string consisting solely of bs is in L_1

There are an infinite number of correct solutions. Good solutions are relatively concise expressions whose correctness is apparent. Here are two such solutions:

1. `b*(ab*ab*)* + a*ba*(ba*ba*)*`

2. `((a+b)(a+b))* + b*(ab*ab*)*`

In both of these solutions, `b*(ab*ab*)*` denotes the language of all strings with an even number of as. Putting the `b*` *outside* `(ab*ab*)*` handles the case of zero as — i.e., it includes all strings consisting solely of bs. Many students included the regular expression `(b*ab*ab*)*`, which is similar, but does not include strings with zero as.

By symmetry, `a*(ba*ba*)*` denotes the language of all strings with an even number of bs. To handle an *odd* number of bs, we must have at least one. This effect is obtained by prepending `a*b` to `a*(ba*ba*)*` to yield `a*ba*(ba*ba*)*`. Solution 1 is simply the sum of the regular expressions for even a's and odd b's.

Solution 2, which is adapted from Kristen Taylor's solution, is intriguing. Why does it work? The regular expression `((a+b)(a+b))*` denotes all even length strings of as and bs. All of these are in L_1 because they either contain an even number of as or an odd number of as, and if they contain an odd number of as, they necessarily contain an odd number of bs!

What strings in L_1 are not described by `((a+b)(a+b))*`? They must be odd-length strings that contain an even number of as or an odd number of bs. But for odd-length strings, one of these necessarily implies the other! So if we include the regular expression for *all* strings with an even number of as, we will include all the strings missing in `((a+b)(a+b))*` without adding any strings that don't belong in L_1. Symmetrically, the following is also a solution:

```
((a+b)(a+b))* + a*ba*(ba*ba*)*
```