Problems: Partial Evaluation and Symbolic Composition

1. (a) Describe three ways in which partial evaluation can speed up the execution of a program. That is, what are three optimizations that a partial evaluator may apply so that the execution of p_s on d is faster than the execution of p on $[s, d]$.
   (b) Explain why partial evaluation might slow a program down.
2. (a) One way of speeding up the creation of specialized programs is via the notion of a *generating extension*. A program $\mathrm{p}gen$ is a generating extension for p if

   $$[\![\mathrm{p}gen]\!]s = p'_s, \text{ such that for all } d, [\![p'_s]\!]d = [\![p]\!][s, d].$$

   That is, unlike normal partial evaluation of p, $\mathrm{p}gen$ already has p "built into it" so that, when supplied with an argument s, it creates a program p'_s, where p'_s operates just like the program p_s produced via partial evaluation. (Note that p'_s and p_s are not necessarily *identical* programs—just ones with *identical behaviors*.)

   Suppose that `DotProduct` computes the dot product of two vectors of length N:

```
const int N = <some constant>;

int DotProduct(int x[], int y[]) // x and y assumed to be of length N
{
  int answer = 0;

  for (int i = 0; i < N; i++) {
    answer = answer + x[i] * y[i];
  }
  return answer;
}
```

   Write a procedure `DotProduct-gen` that writes a version of `DotProduct`, specialized to the value of `x[]`, to the standard output:

```
void DotProduct-gen(int x[])
{
  // MISSING -- body of DotProduct-gen
}
```

   (b) Compared with applying a partial evaluator to p and s, why is applying $\mathrm{p}gen$ to s likely to be faster?
   (c) Suppose that $\mathrm{p}e$ is a self-applicable partial evaluator. Let $\mathrm{c}ogen \stackrel{\text{def}}{=} [\![\mathrm{p}e]\!][\mathrm{p}e, \mathrm{p}e] = \mathrm{p}e_{\mathrm{p}e}$. Show that $[\![\mathrm{c}ogen]\!]p$ yields a program that is a generating extension for p.

Questions 3 and 4 explore certain aspects of *symbolic composition*, which is a program transformation that bears some relationship to partial evaluation.

3. An $m \times n$ matrix M over the real numbers $\mathcal{R}$ determines a linear transformation $[\![M]\!]\colon \mathcal{R}^n \to \mathcal{R}^m$. That is, if $v \in \mathcal{R}^n$, then $[\![M]\!](v)$ is a vector $u \in \mathcal{R}^m$. (We can compute u by doing a matrix-vector multiplication: $u \;=\; M \times v$.)

   If M, N are matrices of dimensions $m \times n$ and $n \times p$, respectively, and $M \times N$ is their matrix product, then $[\![M \times N]\!]\colon \mathcal{R}^p \to \mathcal{R}^m$. We have

   $$([\![M]\!] \circ [\![N]\!])(w)) = [\![M]\!]([\![N]\!](w)) = [\![M \times N]\!](w),$$

   which means that $M \times N$ represents the *symbolic composition* of M and N.

   Suppose that we have a collection of vectors $\{v_i\}$ that we wish to transform by $[\![M]\!] \circ [\![N]\!]$. We can do the computation either as $\{M(N(v_i))\}$ ("sequential application") or as $\{(M \times N)(v_i)\}$ ("symbolic composition"). What is the break-even point for symbolic composition? That is, how many vectors do we have to have for it to be better to use the symbolic-composition method rather than the sequential-application method? (Your answer should focus on the number of scalar-multiplication operations performed; you do not have to count additions exactly.)

4. A *(nondeterministic) finite-state transducer* is a finite-state machine that transforms input strings from Σ^* into output strings from Δ^* (where, in general, Σ and Δ are two different alphabets). A finite-state transducer is similar to a standard finite-state automaton except that it also has an output alphabet Δ, and the transiton relation, λ, associates each transition with an output symbol in $\Delta \cup \{\epsilon\}$. Formally, a finite-state transducer has five components:

   Q, a set of states
   Σ, the input alphabet
   Δ, the output alphabet
   $\lambda \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Delta \cup \{\epsilon\}) \times Q$, the transition relation
   q_0, the initial state

   (Note that there is no set of final states.)

   At run-time, whenever the machine is in state q and the current input symbol is a, the permissible transitions—with output b—are to the states r such that $\langle q, a, b, r\rangle \in \lambda$. For an input string x, the machine's output string can be any of the strings of output symbols generated in this nondeterministic fashion.

   It is convenient to think of a finite-state transducer as a directed multi-graph whose nodes are the states, and where each tuple $\langle q, a, b, r\rangle \in \lambda$ corresponds to an edge from q to r, labeled with the pair "(a, b)" (meaning that on a transition from q to r on which a is "consumed" from the input string, b is generated in the output string, where a and b are possibly ϵ).

   (a) Give the formal definition of a nondeterministic finite-state transducer M that is the "single-error introducer" from $\{0, 1\}^*$ to $\{0, 1\}^*$. That is, M should be a transducer that "corrupts" *up to one bit* of the input string. For example, if the input string is 101, M can produce any of the following strings: 101, 100, 111, 001, 01, 11, 10, 0101, 1101, 1001, 1011, and 1010 (but not, for instance, 011, 000, 00, or 0000).

   (b) Suppose that you are given two finite-state transducers: M, which transforms strings from Σ^* to strings from Δ^*, and N, which transforms strings from Δ^* to strings from Γ^*.

   Give an algorithm for the composition of N and M; that is, the output of the algorithm is to be a single finite-state transducer $P = N \circ M$ that transforms strings from Σ^*

directly to Γ^*, such that P gives the same transduction that we would have if M were to be applied first and then N applied to M's output. (Of course, since P is a *single* finite-state transducer, there is no opportunity for it to produce any kind of "intermediate string.")

Hint/warning: make sure that you specify how the algorithm handles transitions that involve ϵ—i.e., transitions of the form $\langle q, \epsilon, b, r\rangle$, $\langle q, a, \epsilon, r\rangle$, and $\langle q, \epsilon, \epsilon, r\rangle$.

(c) Give the composed transducer that your construction from Part (b) creates when the machine from Part (a) is composed with itself.