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 $p_{gen}$ is a generating extension for $p$ if $[p_{gen}]s = p'_s$, such that for all $d$, $[p'_s][d] = [p][s,d]$.

That is, unlike normal partial evaluation of $p$, $p_{gen}$ 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 $N$:

```c
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:

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

(b) Compared with applying a partial evaluator to $p$ and $s$, why is applying $p_{gen}$ to $s$ likely to be faster?

(c) Suppose that $pe$ is a self-applicable partial evaluator. Let $cogen \overset{def}{=} [pe][pe,pe] = pe_{pe}$. Show that $[cogen][p]$ yields a program that is a generating extension for $p$.

3. This question and the next one explore certain aspects of symbolic composition, which is a program transformation that bears some relationship to partial evaluation.
An \( m \times n \) matrix \( M \) over the real numbers \( \mathcal{R} \) determines a linear transformation \([M] : \mathcal{R}^n \to \mathcal{R}^m\). 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 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 \([M \times N] : \mathcal{R}^p \to \mathcal{R}^m\). We have

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

which means that \( M \times N \) represents the \textit{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?

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

\begin{itemize}
  \item \( Q \), a set of states
  \item \( \Sigma \), the input alphabet
  \item \( \Delta \), the output alphabet
  \item \( \lambda \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Delta \cup \{\epsilon\}) \times Q \), the transition relation
  \item \( q_0 \), the initial state
\end{itemize}

(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 \( (q, a, b, r) \in \lambda \). 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 \( (q, a, b, r) \in \lambda \) 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 \( \epsilon \)).

(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” \textit{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 (but not, for instance, 011 or 000).

(b) Suppose that you are given two finite-state transducers: \( M \), which transforms strings from \( \Sigma^* \) to strings from \( \Delta^* \), and \( N \), which transforms strings from \( \Delta^* \) to strings from \( \Gamma^* \). Describe how to create a single finite-state transducer \( P = N \circ M \) that transforms strings from \( \Sigma^* \) directly to \( \Gamma^* \), 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 \textit{single} finite-state transducer, there is no opportunity for it to produce any kind of “intermediate string.”)

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