1

\[
\begin{align*}
\text{replace_one}(X, Y, [X \mid L], [Y \mid L]). \\
\text{replace_one}(X, Y, [Z \mid L], [Z \mid R]) :&- \\
& \quad \text{replace_one}(X, Y, L, R).
\end{align*}
\]

One or two points were taken off for programs that seemed “debuggable.” If they were completely on the wrong track, the scores went below 8.

Most common bug: Adding to the second clause the conjunct \(Z \neq X\). This is wrong. Even if \(Z = X\), if Prolog backtracks to this goal, it should try clause 2. One point off for this bug. (I made this mistake in my first version, too.)

2

(a) Expression

(b) \text{curry}(MF, ML, Expression) \quad \text{[The variable name Expression is not important, so long as whatever variable name is used occurs in both (a) and (b).]}

(c) \([MS \mid ML]\)

It was disheartening to see how many people tried to call \text{curry} as a function. There was only one place \text{curry} could occur, and that was inside the braces.
As usual, the variable names don’t matter, so long as the connections are preserved.

A simpler answer to (d) would be \( \text{lam}(X, \text{M}_1@X & \text{M}_2@X) \). In the genuine \( \lambda \)-calculus, this expression would be disallowed, because you can’t substitute an expression into a \( \lambda \)-expression without worrying about \( \alpha \)-conversion.\(^1\) However, in Blackburn and Bos’s Prolog encoding, anything goes, and in the present application no problems can actually ensue. The technically correct solution is the answer shown,\(^2\) but full credit was given for the simpler answer.

This problem was solved by almost no one. A few people came up with the following system of answers: (a) \( \text{lam}(X, \text{color}(X, C)) \); (b) \( \text{lam}(X, \text{color}(X, C)) \); (c) \( \text{lam}(X, \text{sheen}(X, H)) \); (d) \( \text{lam}(X, \text{sheen}(X, H)) \); (d) \( \text{lam}(X, \text{color}(X, C) & \text{sheen}(X, H)) \); (d) \( \text{lam}(X, \text{color}(X, C)) \); (e) \( \text{lam}(X, \text{color}(X, C)) \). Annoyingly, this will actually work for this crude system, so it got full credit as well, even though it is grossly inelegant.

The reason the problem was so hard to solve is that two ideas just didn’t get across: (1) that DCG rules are backward-chaining rules; and (2) that unification allows information to flow in either direction across rules; in this case it flows right-to-left.

\(^4\) I thought the instructions were clear, but some students thought I meant they should literally write “\( c_1 \neq c_2 \)” when two constants failed to match. People, italicized letters in mathematical formulas are variables.

\(^1\) Briefly, suppose \( M_1 \) or \( M_2 \) had free occurrences of \( X \). Then substituting them into \( \text{lam}(X, \_@X & \_@X) \) would mean something other than what was intended. The solution is to rename variable \( X \) to avoid this kind of “variable capture.”

\(^2\) This version avoids the problem because \( \beta \)-conversion is defined to include whatever \( \alpha \)-conversions are necessary.
Each subanswer was worth 2 sub-points. In 4-2, the answer is not $g(a) \neq g(g(a))$. The unification algorithm first checks that $g = g$, then tries to unify the arguments of $g$ in the two terms. One point off for getting this wrong. For more subtle reasons, the answer is not $a \neq g(a)$ either, although only a half point was lost for this one. The reason is that $a$ wouldn’t match $g(Z)$ either, but $g(Z)$ is not a constant. It’s the fact that $a$ has the wrong functor (and the wrong number of arguments — 0) that causes unification to fail.

1. $X = a, Y = a$
2. Fail — $a \neq g$
3. $X = Y$ or $Y = X$
4. Fail — $X$ (or $Y$)
5. $X = f(b, b), Y = b$

Half a point was taken off for the first breach of the requirement that, in $V = e$, $e$ did not contain any variable $V'$ that appeared in another equation $V' = e'$. A full point was taken off for the second breach.

5a A projective dependency graph is one with the property that (pick any of the following)

- i . . . If $w_j$ is a dependent of $w_i$, then all the words between $i$ and $j$ have $w_i$ as an ancestor in the dependency graph.
- ii . . . The graph of the dependency relationship has no arcs that cross other arcs.
- iii . . . The graph of the dependency relationship is nested.

Unfortunately, most students assumed I wanted them to define “best,” which seems bizarre, given that the question went into exhaustive detail about the max computation. But if someone really thought the question was asking about the $\delta_{w,w'}$ I gave them a few points.
It’s simply not true that dependency parsers are looking for the most likely parse, unless $\delta_{w,w'}$ is interpreted as the logarithm of a probability. If you claim that that’s the goal, you have to mention the bit about the logarithms.

5b Eisner’s algorithm depends on the fact that a projective graph covering all the words in interval $[s,t]$ and headed at $s$ or $t$ can be assembled from exactly two subgraphs of the same kind. (The subgraphs come in four varieties.) The CYK algorithm also depends on combining exactly two trees to make a new tree, which depends on the grammar being in CNF. In other words, projectivity is enough, although perhaps it’s a bigger constraint than CNF in the end. Part (a) was intended to nudge people toward the right idea, but since many people missed the intent of (a), the nudge didn’t help.

There were all kinds of answers to question 5. Apparently dynamic programming is not that well understood among CS students. Many of the answers amounted to core dumps, which were pointless given that the question spelled out most of what was regurgitated in these answers.

$6 \ (x_1 \land x_2) \lor (x_1 \land x_3) \lor (x_2 \land x_3)$

Output is 1 if at least 2 inputs are 1.