Please put your answers to each question in the blank spaces provided, either the underlined spaces or the big empty boxes. (If you don’t see the box, turn the page.) Don’t feel that you have to fill the space; the size is the largest space a wordy person with huge handwriting might need. The size of your answer might be much smaller than that. WRITE YOUR NAME at the top of the odd-numbered pages after the word “NAME.” (Putting it only in the supplied spaces helps us to grade papers without knowing who you are; signing every page prevents us from losing your paper, if we’re lucky.) Each of the 6 problems is worth 1 2 3 points, for a total of 10. The marked papers will show each problem graded on a 10-pt scale, but the scores will be normalized so they add up to a total that makes sense on a 10-point scale, which will appear at the top of this page when we’re done grading.

1 The following Prolog program replaces every occurrence of X in a list of atoms with Y:

```prolog
replace_every(_, _, [], []). 
replace_every(X, Y, [X | L], [Y | R]) :- 
    replace_every(X, Y, L, R).
replace_every(X, Y, [Z | L], [Z | R]) :-
    Z \= X,
    replace_every(X, Y, L, R).
```

Examples of replace_every’s behavior:

?- replace_every(a, b, [a,b,c,a,d,c], R).
R = [b, b, c, b, d, c] ;
false.
?- replace_every(a, b, [], R).
R = [] ;
false.

Write a function replace_one(+X, +Y, +L, -R) that replaces exactly one occurrence of X in L with Y, binding R to the result. It fails if L contains no occurrences of X. Examples:

?- replace_one(a, b, [a,b,c,a,d,c], R).
R = [b, b, c, a, d, c] ;
R = [a, b, c, b, d, c] ;
false.
?- replace_one(a, b, [], R).
false.
?- replace_one(a, b, [p, q, r], R).
false.

Note that if there is no occurrence of X in L, replace_one fails. If it succeeds, but backtracking reaches the same goal again, it may succeed again, replacing a different occurrence of X with Y. (It doesn’t matter in what order it generates the substituted versions of L.)

Definition of replace_one:

Hint: The definition may be written using exactly two clauses. Definitions sometimes get simpler when nondeterminism is involved.

2 In this problem we consider a language consisting of applications of functions to arguments, written \(<f(a_1,\ldots,a_n)>\). (The angle brackets are part of the language, not part of the meta-language.) We want to express the meaning of such an expression in a language in which functions are curried, meaning that functions take just one
argument. So, if the meaning of expression $e$ is $e^*$, the meaning of $<f(a_1,\ldots,a_n)>$ should be $\ldots((f^*(a_1^*)(a_2^*)\ldots)(a_n^*)$.

Unfortunately, this term is not legal Prolog, so we will have to express it as

$$\text{fcall}(...\text{fcall}(...\text{fcall}(f^*,a_1^*),a_2^*),\ldots,a_n^*)$$

There are just two syntactic categories, $s$ and $\text{arglist}$. The former is an expression (sentence) of the language, and the latter is a nonempty sequence of comma-separated sentences. The first argument of the predicates $s$ and $\text{arglist}$ is the list of free variables that are expected. The second argument is the semantical value of the expression. The definitions of $s$ and $\text{arglist}$, and the auxiliary function curry, appear below, with three places left blank. Fill them in.

```prolog
s(VL, (a)) --> ['<'], s(VL, MF), ['('], arglist(VL, ML), [')'], [']'], {
(b)
}. 

s(_, add) --> ['+'].
s(_, mult) --> ['*'].
s(_, subtract) --> ['-'].
s(VL, vval(X)) --> [X], {member(X, VL)}. 

arglist(VL, [MA]) --> s(VL, MA). 

arglist(VL, (c)) --> s(VL, MS), [','], arglist(VL, ML). 

curry(F, [MX], fcall(F, MX)).
curry(F, [MA | ML], RA) :-
curry(fcall(F, MA), ML, RA).

The only constants in the language are '+', '*', and '-'. The only other atoms that are allowed (besides punctuation marks) are variables from the list $VL$; their meaning is $vval(varname)$. The auxiliary function curry generates the fcall representation. All you have to do is call it in the right place.

Example:

?- s([x,y,z], M, 
    ['<','*','(',')','<','+','(',')','x','(',')','y','(',')','>','>','>','<', 
    'z','(',')','x','(',')','y','(',')','>','>','>'], []
)
\[ M = \text{fcall}(\text{fcall}(\text{mult}, \text{fcall}(\text{fcall}(\text{add}, \text{vval}(x)), \text{vval}(y))), \text{fcall}(\text{fcall}(\text{vval}(z), \text{vval}(x)), \text{vval}(y))). \]

The input list may be a little hard to read; it's the string \(<*(<+(x,y)>,<z(x,y)>)>\) turned into a list of one-character atoms.

On an obscure Mediterranean island, apprentices in paint stores are given their instructions in a special dialect. Paints are arranged with all paints of one color in a column, all paints of one sheen in a row. If someone calls out “plink,” that means “red.” The apprentice closest to the column with red paints is expected to move to that column. If someone calls out “slob,” that means “semigloss,” and an apprentice moves to the column with semigloss paints. If someone calls out “plank slab,” that means “green gloss,” and the apprentice closest to the intersection of the “green” column and the “gloss” row must move there. (There are other commands for bringing a paint to the front desk etc., but we won’t look at those.) Every utterance in this (fragment of the) language has at most two words, the first referring to the color and second referring to the sheen.

Here is a DCG for this language:

\[
\begin{align*}
S((a)) & \rightarrow C((b)) \\
S((c)) & \rightarrow H((d)) \\
S((e)) & \rightarrow C((f)) H((g)) \\
C(\lambda(X, \text{color}(X, \text{red}))) & \rightarrow \text{plink} \\
C(\lambda(X, \text{color}(X, \text{green}))) & \rightarrow \text{plank} \\
C(\lambda(X, \text{color}(X, \text{blue}))) & \rightarrow \text{plonk} \\
H(\lambda(X, \text{sheen}(X, \text{matte}))) & \rightarrow \text{slib} \\
H(\lambda(X, \text{sheen}(X, \text{semigloss}))) & \rightarrow \text{slob} \\
H(\lambda(X, \text{sheen}(X, \text{gloss}))) & \rightarrow \text{slab} \\
\end{align*}
\]

— using Blackburn and Bos’s notation for \(\lambda\)-expressions.

The omitted arguments in the first four lines are the semantics of the phrases in question. Fill them in. Don’t worry about the stories about telling the apprentices to move; just fill in the semantics relating to the colors and sheens. All the blanks are the same size, but some of the answers are much shorter than others. (Feel free to write “\(\lambda\)am” as “\(\lambda\)”.)
To be clear, the semantics of (e.g.) “plink slab” is \( \text{lam}(X, \text{color}(X, \text{red}) \& \text{sheen}(X, \text{gloss})) \), or something that \( \beta \)-reduces to it. (In case you’ve forgotten, \( \beta \)-reduction means converting \( \lambda(V,A)@T \) to \( A[T/V] \), that is, \( A \) with \( T \) substituted for all free occurrences of the variable \( V \). The “\( @ \)” is the infix operator meaning “apply”: \( f@T \) means “apply \( f \) to argument \( T \).”)

For each of the following pairs, write its most general unifier (MGU); or “Fail — Reason” if they don’t. Reason is either “\( c_1 \neq c_2 \)” if \( c_1 \) and \( c_2 \) are unequal constants or functions that occur in corresponding positions in the two patterns; or “\( v \)” if variable \( v \) would have to be bound to a term containing \( v \) as a proper part (an occur check failure).

1. \( f(X,a) \ ? f(Y,Y) \):

2. \( f(g(a), g(g(a))) \ ? f(Y,Y) \):

3. \( f(X,X) \ ? f(Y,Y) \):

4. \( f(X,X) \ ? f(g(Y), Y) \):

5. \( f(X,X) \ ? f(f(Y,b), f(b,Y)) \):

The expressions have been “standardized apart” so that no variable renaming is necessary.

Express MGUs as lists of expressions of the form \( v = e \), where \( v \) is a variable and \( e \) is an expression. (If \( e \) is itself a variable, you don’t need to write \( e = v \); which variable appears on the left is unimportant.) Vital: \( e \) must not be a variable such that \( e = \ldots \) is an element of the MGU; nor must it contain any such variables. Make sure they are substituted away.

If an occur-check failure blocks unification, there may be more than one candidate for the variable to blame, depending on how you manage the bindings; just specify one variable that’s involved in the failure.

This question compares the CYK algorithm for parsing with context-free grammars and Eisner’s algorithm for projective dependency parsing.
for (length <- 2 to n;
    s <- 0 to (n - length))
{
    val t = s + length - 1;
    ...

CYK:
    C((s,t)) = empty set;
    for (q <- s to (t - 1);
        n1 <- C((s,q));
        n2 <- C((q+1, t));
        r <- rules in the grammar
            of the form
            n --> n1 n2)
    C((s,t)) = C((s,t)) + r.lhs
}

Eisner:
    E((s,t)).RL
    = max_{s \leq q < t} \left( E((s,q)).LL + E((q+1,t)).RR + \lambda_{w_{t},w_{s}} \right)
    E((s,t)).LR
    = max_{s \leq q < t} \left( E((s,q)).LL + E((q+1,t)).RR + \lambda_{w_{s},w_{t}} \right)
    E((s,t)).RR
    = max_{s \leq q < t} \left( E((s,q)).RR + E((q,t)).RL \right)
    E((s,t)).LL
    = max_{s < q \leq t} \left( E((s,q)).LR + E((q,t)).LL \right)

Figure 1: Side-by-side views of CYK and Eisner algorithms. The dynamic-
programming table is called C in the former and E in the latter. Each entry C((s,t))
is a list of all the nonterminals that the phrase from s to t could be parsed as. And entry E((s,t)) is a record containing 4 numbers describing, LL, LR (the LS), RR, and RL (the Rs). The L scores are the best score obtainable for an interval assuming its head is at s; the Rs the best scores assuming its head at t. The distinction between the two kinds of LS and the two kinds of Rs comes down to the structure of the prime child, the rightmost child of an L and the leftmost of an R. So the four fields of E((s,t)) have the following meaning:

1. LL: Maximum score of L whose prime child is an L in some interval
   \([q,t], s < q \leq t.\)
2. LR: Maximum score of L whose prime child is an R in some interval
   \([q,t], s < q \leq t.\)
3. RR: Maximum score of R whose prime child is an R in some interval
   \([s,q], s \leq q < t.\)
4. RL: Maximum score of R whose prime child is an L in some interval
   \([s,q], s \leq q < t.\)
Eisner max computation
Given interval $l,m$ and function $f$,
with $l \leq m$
best = $f(l)$;
for ($q \leftarrow (l+1)$ to $m$)
    let $fq = f(q)$ in
    if $fq >$ best then best = $fq$
return best

Called 4 times, with $[1,m] = [s, t-1]$ except for the last time, when $[1,m] = [s+1, t]$. $f(q)$ = expression to be maximized.

Figure 2: Schematic version of max computations in figure 1

To avoid straining your memory, we provide Figures 1 and 2, which show the central loops of the two algorithms, in pseudo-code. The first three lines of figure 1 are the same. The CYK code that follows is on the left, the corresponding Eisner code on the right.

(The notation
\[
\text{for } (v_1 \leftarrow g_1; \\
v_2 \leftarrow g_2; \\
\ldots \\
v_k \leftarrow g_k) \quad E
\]
means: For every combination of values generated by all the $g_i$, and successively bound to the variables $v_i$, execute $E$. A “generator” might be a range, such as $(l+1)$ to $m$ or a set, such as $\mathcal{C}((s,t))$, the set of nonterminals proven to span the range $[s,t]$.)

Figure 2 shows the loop structure of Eisner’s algorithm hidden in the max expressions, implemented as loops not unlike the inner loop of the CYK code.

5a Eisner’s algorithm finds only the best projective dependency graph. What does that mean?
5b The CYK algorithm requires the grammar it uses to be in Chomsky Normal Form, which means that all rules are of one of two types: $N \rightarrow N_1 N_2$, or $N \rightarrow T$, where the $N$s are nonterminal symbols and $T$ is a terminal symbol. The Eisner algorithm, which is similar in some ways, does not require that the system of dependencies (captured in the $\delta_{w,w'}$ values) obey any similar constraint. Why not?

Hint: Think about what the two algorithms have in common, as dynamic-programming algorithms; what allows them to get away with only $O(n^3)$ operations?

6 This is a question about a one-layer neural net with three inputs and one output. With the dummy input added, there are four inputs, so we need 4 weights, and they are $w[0] = [-15, 10, 10, 10]$. The three real inputs are $x[0][1]$, $x[0][2]$, and $x[0][3]$; the dummy is $x[0][0] \equiv 1$. The input can be considered a column vector of Boolean
values. The output $x[1]$ is size-2 column vector, but only the second element is meaningful:

$$x_{out} = x[1][1] = g(w[0] \cdot x[0]) = g(w[0]^T \cdot x[0])$$

where $g(s)$ is a step function $= 0$ for $s < 0$ and $= 1$ for $s \geq 0$.

Express what this neural network computes as a Boolean function of the three non-dummy $x[0]$ values (i.e., using $\wedge$, $\vee$, and $\neg$):

You can abbreviate $x[0][i]$ ($i \in \{1, 2, 3\}$) as $x_i$. If you can’t think of a way to express the answer using Boolean combinations, express it some other way.

---

1It could also be a sigmoid, but that’s not necessary for this simple example.