1. Why were interpretive systems tolerated in the late 1940’s to mid 1950’s, given that an interpretive system adds so much overhead?

   Floating-point emulation in software masked the performance loss due to interpretation of a language.

2. Imagine a programming language that did not have function calls or compound statements. Briefly state the implication of this. (Suggestion: consider how you might implement an if-then statement.)

   What would otherwise be a block statement

   \[
   \text{Eg. if (cond1) } \begin{cases} \text{ s1; } \\ \text{ s2; } \\ \text{ else } \begin{cases} \text{ s3; } \\ \text{ s4; } \\ \end{cases} \end{cases}
   \] 

   becomes

   \[
   \text{BNE cond1 target1 } \\
   \text{ s1 } \\
   \text{ s2 } \\
   \text{ BR end target1: } \\
   \text{ s3 } \\
   \text{ s4 } \\
   \text{ end: }
   \]

3. The Spanish word \textit{manana} literally is translated as \textit{tomorrow}, however it also carries with it an implication of "some day" or "later on," not necessarily exactly within the 24 hour period from now. Someone who speaks Spanish might understand this, while a person who speaks English only may not have ever thought of this particular concept. This is an example of what hypothesis? Explain the hypothesis.
4. Write a grammar to accept the language: \{a^n b^n \mid n \geq 0\}

\[ S \rightarrow aSb \mid \epsilon \]

5. Draw the LISP linked-list interpretation of:

\[(A \ (B \ C \ (D \ E) \ (F \ (G \ H))) \ K )\]

![LISP linked-list interpretation diagram]

6. Consider the attribute grammars such as the one discussed in class. Give an example of:

a) A \textit{synthesized} attribute

actual type of the left-hand-side of an assignment statement.

b)An \textit{inherited} attribute

expected type of the right-hand-side of an assignment statement.

c)An \textit{intrinsic} attribute

type of a variable
7. Suppose you were writing code to find flight information from Chicago to Nome, Alaska, where Chicago has a large busy airport and Nome has very few flights. Which would be better: forward chaining or backwards chaining? Explain your answer.

Backwards-chaining from Nome would be better because it prunes the size of the search space more quickly.

8. Does the following grammar maintain correct associativity for addition and multiplication? Explain why or why not.

```
<assign> --> <id> := <expr>
{id} --> A | B | C
<expr> --> <expr> + <term> | <term>
<term> --> <factor> * <term> | <factor>
<factor> --> ( <expr> ) | <id>
```

No it does not, since the production rule for multiplication is right-recursive. This means that the right-most part of a string of multiplications will be done before that on the left, which is the opposite of what we would expect mathematically.

9. Rewrite the following EBNF grammar as a syntax diagram

```
A -> ( 'a', { 'b' <A> }, { <A> [ <B> ] <A> } )
B -> ( [ 'b' ], { 'c' [ <B> ] 'b' } )
```
10. Write the PROLOG propositions to delete all instances of an element in a list. Call it \textit{deleteAll}. For example, calling:
\begin{verbatim}
   deleteAll( a, [a,b,c,a,d], Answer).
\end{verbatim}
would give:
\begin{verbatim}
   Answer = [b,c,d].
\end{verbatim}

% delete all occurrences of an element in a list
% e.g.
%deleteAll(a,[a,b,a,c], Answer)
% should give:
% Answer = [b,c]
%

% base condition. Deleting element from empty list gives empty list
deleteAll( Element, [], []).

% recursive condition. When element is at the front of the list,
% delete that element and recursive try again on the remainder of the list.
deleteAll( H, [H|Tail], Result):-
   deleteAll(H, Tail, Result).

% the "average case, where we pull off the first element, storing
% it to add back on the way out of the recursion
deleteAll( E, [H|Tail], [H|NewList]):-
   E \rightleftarrows H,
   deleteAll( E, Tail, NewList).