1. **Problem 1**

Which compiler phase (e.g. scanner, parser, constrainer, code generator) would be responsible for catching each of the following errors, respectively? Explain why.

1. An illegal character.
2. Premature end-of-line.
3. A undeclared variable.
5. Division by zero.

**Solution (4 points each):**

1. The scanner.
   The scanner is the only phase of compilation that deals with the actual characters of the program. So if an illegal character is found, the scanner will be the first to discover it.

2. The parser.
   When the scanner discover an EOF, it has no way of knowing whether the EOF is premature or not. It would require the further knowledge of the syntax of the program, which is the responsibility of the parser.

3. The constrainer
   An undeclared variable is a semantic error, and should be handled by the constrainer.

4. constrainer / link time
   Depends on the language implementation. If library functions are analyzed at compile time, then the constrainer (through the symbol table) would catch the error. Quite often, however, the errors is caught at link time, when external references (such as references to library functions) are resolved.

5. constrainer / runtime
   Depends on whether the value of the divisor can be calculated at compile time, e.g. the expression n/0. Here, the static semantics analyzer could catch the error. However, if the divisor can't be calculated until run time, then the compiler cannot catch it at all.

2. **Problem 2**

   (Chapter 1, Exercise 1.4)

   1. In your local implementation of C, what is the limit on the size of integers? What happens in the event of arithmetic overflow? What are the implications of size limits on the portability of programs from one machine/compiler to another?

   2. How do the answers to these questions differ for Java? For Ada? For Pascal? (You may need to find a manual.)

**Solution:**

(10 points) 1. The answer here is of course implementation dependent. Using gcc on a Pentium 4 Linux system, an int is 32 bits long. Arithmetic overflow happens silently and numbers wrap around. Size limits significantly impact the
portability of C programs. There are systems in which an int is 16 bits, or 36, or 64. The language definition requires at least 16 bits.

(10 points) 2. Java is uniform across platforms: an int is always 32 bits. Ada allows considerable variation across platforms, but imposes a variety of constraints on range and behavior. It also provides mechanisms that, if used, result in accidental overflow triggering an exception condition. Among other things, the built-in Integer type must span at least \(-2^{15} + 1 \ldots 2^{15} - 1\). Pascal says that integers run from -maxint to maxint, where maxint is an implementation defined whole number. Implementations are free to pick any convenient number.

Scheme guarantees arbitrary arithmetic precision on all platforms. Depending on the size of underlying integers, programs on different platforms may vary in execution speed.

3. Problem 3
Consider the following simplified version of English Grammar:

\[
\begin{align*}
\text{<sentence> ::= <noun phrase> <verb phrase>.} \\
\text{<noun phrase> ::= <determiner> <noun>} \\
\text{| <determiner> <noun> <prepositional phrase>} \\
\text{<verb phrase> ::= <verb>} \\
\text{| <verb> <noun phrase>} \\
\text{| <verb> <noun phrase> <prepositional phrase>} \\
\text{<prepositional phrase> ::= <preposition> <noun phrase>} \\
\text{<noun> ::= boy | girl | dog | ball | photo | feather} \\
\text{<determiner> ::= a | the} \\
\text{<verb> ::= saw | touched | kicked | took} \\
\text{<preposition> ::= by | with}
\end{align*}
\]

a). Draw a Derivation Tree of the following input:
the boy touched the dog with a feather.

b). Is this an ambiguous grammar? Why?

Solution:
(15 points)1. Two derivation trees are possible as shown in Fig. 1 and 2 below.

(5 points)2. The given grammar is ambiguous. A grammar is ambiguous if some phrase in the language generated by the grammar has two distinct derivation trees.

4. Problem 4
Given the following LL(1) grammar:

1. \(P \rightarrow S \$$
2. \(S \rightarrow \{ S \} S\)
3. \(S \rightarrow [ S ] S\)
4. \(S \rightarrow \)

and the following input:
\(( ( ) [ ] ) [ ] $$

1. Draw the Parse Table for the grammar.
2. Draw the LL(1) Parsing Traces of the input sentence above, with the format similar to that in lecture slides (04page#13).
3. Which has higher precedence? "[" or ")"?

**Solution:**
(8 points) 1. Parse Table: refer to Fig. 3.
(8 points) 2. Parsing Traces
Refer to Fig. 4

(4 points) 3. "[" and ")" have the same precedence. (This question asks for the precedence of "[" and ")" in this specific grammar. So any answer that is not related to this grammar will get no credit)

**Problem 5 (20 Points)**

Consider the following grammar:

```
A -> B (',', B)+ => "a" 
  -> B 
B -> B ' &' C => "b" 
  -> C 
C -> D '#' C => "c" 
  -> D 
D -> <identifier> 
```

Write the skeleton of a recursive descent parser for this grammar, including 'BuildTree' statements that will build the AST bottom-up, for the original grammar.
Solution:

Proc A; { A -> B ("," B)+ => "a"
  -> B
}
Var N: Integer;
B:
N :=1;
While Next_Token = T_, do
  Read (T_,);
  B;
  N := N + 1;
Od
Build_Tree ("a", N);
end;

Proc B; { B -> B "&" C => "b"
  -> C
}
C;
while Next_Token = T_& do
  Read (T_&);
  C;
Build_Tree ("b", 2);
Od
end;

Proc C; { C -> D "#" C => "c"
  -> D
}
D;
if Next_Token = T_#
  then Read (T_#);
  C;
Build_Tree ("c", 2);
end
end;

proc D; { D -> <iden< er> }
Read( T_<iden< er> );
end;
Figure 1: derivation tree 1

Figure 2: derivation tree 2

Figure 3: Parse Table
<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Table lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>( () [ ] )</td>
<td>$ $</td>
</tr>
<tr>
<td>S $ $</td>
<td>( () [ ] )</td>
<td>$ $</td>
</tr>
<tr>
<td>( S ) S $ $</td>
<td>( () [ ] )</td>
<td>$ $</td>
</tr>
<tr>
<td>S ) S $ $</td>
<td>( [ ] [ ] )</td>
<td>$</td>
</tr>
<tr>
<td>S ) S $</td>
<td>( [ ] [ ] )</td>
<td>$ $</td>
</tr>
<tr>
<td>S $</td>
<td>( [ ] [ ] )</td>
<td>$ $</td>
</tr>
<tr>
<td>S ] S ) S $ $</td>
<td>( [ ] [ ] )</td>
<td>$ $</td>
</tr>
<tr>
<td>S ] S $ $</td>
<td>( [ ] [ ] )</td>
<td>$ $</td>
</tr>
<tr>
<td>S ] S $</td>
<td>( [ ] [ ] )</td>
<td>$ $</td>
</tr>
<tr>
<td>S ] $</td>
<td>( [ ] [ ] )</td>
<td>$</td>
</tr>
<tr>
<td>S ]</td>
<td>( [ ] [ ] )</td>
<td>$ $</td>
</tr>
<tr>
<td>S ]</td>
<td>( [ ] [ ] )</td>
<td>$</td>
</tr>
<tr>
<td>S ]</td>
<td>( [ ] [ ] )</td>
<td>$ $</td>
</tr>
<tr>
<td>S</td>
<td>( [ ] [ ] )</td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td>( [ ] [ ] )</td>
<td>$</td>
</tr>
</tbody>
</table>