1. (10 pts)
Consider the following grammar for Pascal variable declarations:

```
var-decl  →  var decl-list
decl-list  →  decl ; decl-list
decl-list  →  decl
decl  →  id-list : id
id-list  →  id-list , id
id-list  →  id
```

Indicate any problems in this grammar that prevent it from being parsed by a recursive-descent parser with one token lookahead.

Transform this grammar into a form that can be parsed with a recursive-descent parser with one token lookahead. Make as few changes to the grammar as possible. Do not use Extended BNF.

2. (10 pts)
Consider the following grammar:

```
S  →  S  S  +
   |  S  S  *
   |  a
```

Show both a leftmost derivation and a rightmost derivation of the string ‘a a + a *’.

Present the derivations in the style

```
S  =>  S  S  +
   =>  S  S  +
   =>  ... 
```

where in each step you replace one nonterminal by the right-hand side of a grammar rule until you end up with the symbols ‘a a + a *’ on a line by itself. Underline the nonterminal you are expanding and write the expanded line underneath.
3. (10 pts)
Part of the concrete syntax given in the 1960 Report on ALGOL 60 is as follows:

\[
\begin{align*}
<\text{statement list}> &\rightarrow <\text{statement}> \\
&\quad | <\text{statement list}> ; <\text{statement}>
\end{align*}
\]

\[
\begin{align*}
<\text{statement}> &\rightarrow <\text{assignment}> \\
&\quad | \text{if } <\text{expression}> \text{ then } <\text{statement}> \\
&\quad \quad | \text{if } <\text{expression}> \text{ then } <\text{statement}> \text{ else } <\text{statement}> \\
&\quad \quad | \text{begin } <\text{statement list}> \text{ end}
\end{align*}
\]

In the 1963 report on Algol, this was changed to:

\[
\begin{align*}
<\text{statement list}> &\rightarrow <\text{statement}> \\
&\quad | <\text{statement list}> ; <\text{statement}>
\end{align*}
\]

\[
\begin{align*}
<\text{statement}> &\rightarrow <\text{unconditional statement}> \\
&\quad | \text{if } <\text{expression}> \text{ then } <\text{unconditional statement}> \\
&\quad \quad | \text{if } <\text{expression}> \text{ then } <\text{unconditional statement}> \text{ else } <\text{statement}>
\end{align*}
\]

\[
\begin{align*}
<\text{unconditional statement}> &\rightarrow <\text{assignment}> \\
&\quad | \text{begin } <\text{statement list}> \text{ end}
\end{align*}
\]

Show using examples what if-statements in Algol looked like (with one or more statements in the then/else-parts). Demonstrate with an example why the original syntax was unsuitable and why the revised syntax is an improvement.

4. (10 pts)
Design a grammar for Modula-2 if-statements. In Modula-2, all keywords are written in uppercase characters. If-statements use the keyword \texttt{THEN} instead of parentheses (as in C) around the boolean expression, allow zero or more \texttt{ELSIF}-THEN parts, have an optional \texttt{ELSE} part at the end, and are terminated with the two tokens \texttt{END IF}. The body of a then-part or the else-part is a statement list instead of a single statement as in C. For example, the following two statements are legal Modula-2 if-statements:

\[
\begin{align*}
\text{IF } x < y \\
\text{THEN } x := 3; y := 4 \\
\text{END IF};
\end{align*}
\]

\[
\begin{align*}
\text{IF } x < y \\
\text{THEN } x := 3 \\
\text{ELSIF } x = y \text{ THEN } x := 4 \\
\text{ELSIF } x < z \text{ THEN } x := 5 \\
\text{ELSE } y := 6 \\
\text{END IF}
\end{align*}
\]
The semicolon is not part of the If-statement but is only used to separate statements.

For reference, here is the production for if-statements in languages descending from C:

\[
<\text{statement}> \rightarrow \text{if } ( \ <\text{expression}> \ ) <\text{statement}> \\
[ \ \text{else } <\text{statement}> \ ]
\]

Write the corresponding grammar rule for Modula-2 if-statements.

Don’t just extend the C grammar, write the grammar for Modula-2 if-statements from scratch. You don’t need to write the grammar for assignments (or other statements) and boolean expressions; you can assume they are given. You can use brackets for optional parts in the grammar and braces for parts that can be repeated. Also show the grammar for then- and else-parts (sequence of statements separated by semicolons).

5. (10 pts)

Given the following C++ class declaration:

```cpp
class C {
private:
    int x;
    int y;
public:
    C(int i, int j) { x = i; y = j; }
    virtual C * foo(int i) { x = i; return this; }
    virtual int bar() { return x + y; }
    int blah() { return y; }
};
```

Show the layout of objects of class C and the contents of class C’s virtual function table. Explain in detail what happens at run time when the code

```cpp
C * p = new C(3, 5);
int i = p->foo(37)->bar();
```

is executed. Which value will be assigned to i?