There are six questions. Answer all questions. Be clear, precise, and to the point. Rambling, irrelevant discussions will be penalized.

1. (20 points). We said that recursive descent parsers have problems with grammars that are *left recursive*. What exactly is the problem? Explain briefly.
   How about the rec. descent executor? Does it have problems with such grammars? Explain briefly.

2. (15 points). Write a *pure* BNF grammar for the language of all unsigned decimal integers that contain at least three *consecutive* 1’s. (eg. 10511101 is in the language but not 10151101). Write another (pure) BNF grammar for the language of unsigned decimal integers that contain at least three 1’s (not necessarily consecutive; so both 10511101 and 10151101 would be in this language). You may use the empty string $\varepsilon$ in your productions if you wish.
3. (15 points). Some languages do not have type declarations for variables. Instead, any type of value can be assigned to any variable. But type mismatches are still possible and must be caught; for example, trying to add two variables $xx$ and $yy$ when the current value of $xx$ is boolean, and the value of $yy$ is an integer, is an error. Are these errors, in such languages, context-free, context-sensitive, or run-time? Explain briefly.

4. (25 points). Suppose we want to add a `repeat-until` statement to Core. This should be of the form,

```
repeat SS until cc end;
```

where $SS$ is a `<stmt seq>` and $cc$ is a `<comp>`. This is to be executed by executing $SS$, then evaluating $cc$; if it evaluates to `false`, execute $SS$ again and again evaluate $cc$; keep repeating until after some number of executions of $SS$, $cc$ evaluates to `true`; at that point, the execution of the command terminates. Write down the `Exec-repeat()` procedure; you must use the recursive descent approach; you may use a concrete view of the parse-tree or an abstract view or use the “object-oriented” approach.
5. (15 points). We said that we can’t store the value of an id in the parse-tree (PT) array because the same id might appear in two different places, so we have to store it in the id-table. Why doesn’t this problem appear for, say, expressions? That is, the same expression, say something such as x+y, might appear in two different places; so don’t we need an exp-table in the same way we need an id-table? Explain briefly.

6. (15 points). Some languages (such as C++) include the notion of prototypes using which a programmer can specify that a function that will return a value of a particular type and will expect a set of parameters of specified types will be defined. Other languages (such as Java) do not include such a notion. Why is this? Why do languages such as C++ need this while languages such as Java do not? Does this have anything to do with how the languages are implemented? Explain briefly but clearly.