This exercise continues our exploration of *lex*, *yacc*, *C*, and *make*, and their use in implementing simple languages, such as the graph calculator of Homework 2.

1. List the members of your group below:

2. Describe some strange symptoms that may arise from the casual use of `yytext` as a string in the C code fragments of a yacc grammar that uses lex for tokenization. Provide a simple solution.

3. Provide a C code fragment illustrating a good use of assertions.

4. What does the following lex directive achieve? What is likely to occur if it is removed from the lex file in which it is used?

```
%option noyywrap nounput noinput
```

5. What is the default C type for terminals and nonterminals of a yacc grammar? How may it be changed? How may we use different types for different nonterminals in yacc?

6. Provide a yacc code fragment that implements a string concatenation operator '+' (e.g., `alpha + beta` should yield `alphabeta`).

7. What is a good way to allow lex and yacc to share the the integers used to identify tokens without repetition in source files (which invites bugs due to inconsistent definitions)? How may we let yacc choose default values for some tokens while providing specific values for others?

8. What is a shift-reduce conflict? Provide a simple example, with explanation, and describe how to fix it.

9. What is an easy way to tell yacc that the * operator binds more tightly than the + operator?

10. What is the default action associated with a yacc rule?

11. Provide a *make* rule that generates a PDF `foo.pdf` file from a PostScript file `foo.ps` (for all values of *foo*).

12. Provide **good** test input for the graph calculator, and the expected output. Your input should be designed to make it unlikely that an incorrect implementation produces the correct output.