The following problems are characteristic of ones you may see on the midterm (though some of them are longer and have more parts than real exam problems).

1. Consider the following program in an imaginary language.

```
PROCEDURE main() IS
      TYPE t = FLOAT;
      VAR x,y,z:INTEGER;
      PROCEDURE sub1() IS
            TYPE t = STRING;
            VAR a,y,z : BOOLEAN;
            PROCEDURE sub2() IS
                  VAR a,b,z: t;
                  BEGIN
                        (* body of sub2 *)
                  END
            BEGIN
                  (* body of sub1 *)
            END
      BEGIN
            (* body of main *)
      END
```

List all the bound identifiers in scope in the body of procedure sub2, and give the kind of entity to which the identifier is bound. If the entity is a variable, also give its type. Make the following assumptions about the language: static scoping is used; redefinitions in an inner scope hide those in an outer scope; all keywords are in upper-case and all identifiers are in lower-case; procedures may be recursive.

2. Consider the following grammar for boolean expressions.

$$
\begin{aligned}
E &\to E \text{ or } E \\
E &\to E \text{ and } E \\
E &\to \text{not } E \\
E &\to (E) \\
E &\to \text{true} \\
E &\to \text{false} \\
E &\to \text{id}
\end{aligned}
$$

(a) Show that this grammar is ambiguous.

(b) Rewrite the grammar to remove the ambiguity and enforce the intended precedence order by introducing new nonterminals. Make sure that your revised grammar accepts the same language as the original.

3. Consider the following context-free grammar. (It corresponds roughly to the syntax of lists in the programming language LISP.) $S$ is the start symbol, and the terminals are a , ( ).

$$
\begin{aligned}
S &\to (\ ) \\
S &\to \text{a} \\
S &\to (\ A\ ) \\
\\
A &\to S \\
A &\to A\ ,\ S
\end{aligned}
$$

(a) Show *precisely* why this grammar is not LL(1). (Hint: This will require computing some, but not all, of the FIRST and FOLLOW sets.)

(b) Rewrite this grammar to make it suitable for recursive descent parsing.

(c) On the basis of your revised grammar from part (b), write a recursive procedure S that parses this grammar by recursive descent. Your procedure may be written in C, C++, Java or pseudo-code. You may

assume the existence of a global variable `token` that holds the next input token, a function `advance()` that reads the next token into `token`, and a function `error` that may be called if the input is not in the language generated by the grammar.

(d) Rewrite your solution to (c) to remove all tail-recursion and inline functions that are used only once. You should end up with a single function `s()`.

4. Consider a Unix-shell-like command language for invoking executable files (by naming them), redirecting standard input or output from or to a data file (using `<` and `>`), and piping the output from one executable to the input of another another (using `|`). Suppose any number of executables can be piped together, but input can be redirected from a file only to the first executable, and output to a file only from the last executable. Legal commands include

```
 simple  > outfile < infile
```

and

```
 lexer < goodfile.pcat | parser | ppast > bug.ast.
```

(a) Write a plain (not extended) BNF grammar for this command language, assuming the terminals are `filename`, '`<`', '`>`', and '`|`'.

(b) Draw parse trees for the two commands given above, according to your grammar.

5. Consider the (very artificial) language, over the alphabet of letters and digits and the dollar sign ($), having the following three kinds of tokens: numbers, consisting of one or more consecutive digits; short identifiers, consisting of a *single* letter; and long identifiers, consisting of one or more letters followed by a single $.

a. Write down a regular expression for each of the three token patterns.

b. Draw a DFA for each expression.

c. Combine the DFA's into a common NFA with a common start state and distinct final states for each token, as described in class.

d. Convert your combined NFA into a DFA, using whatever algorithm you like.

e. Suppose you use this DFA to perform lexical analysis, backtracking to the most recently encountered final state when necessary, as described in class. Give an example of an input that will require backtracking and re-reading exactly 5 characters.