Compilers Comprehensive, November 2006

This is a 60 minute, closed book exam. Please mark your answers in the blue book.

1. (10 points)

   Suppose we wanted to build a lexical analyzer for a simplified version of C, which had literal strings beginning and ending with double-quote characters, `"like this"`, using an automated tool like Lex or Flex.

   (a) We could try using the regular expression `\".*\"` for the pattern describing the string. In Lex and Flex, "dot" represents any character except newline.

   Why won't this work very well?

   (b) On analyzing the difficulty in the previous problem, we could decide to build a superior variant of Flex that returns the *shortest* string matching the pattern.

   i. Without changing the pattern in the previous part, will string literals now be handled properly?

   ii. Would this change cause serious problems? Please explain.

2. (25 points)

   Consider this context-free grammar for "reverse Polish" arithmetic expressions:

$$\begin{array}{lcl} S & \rightarrow & E \\ E & \rightarrow & E\ E\ op \\ E & \rightarrow & id \end{array}$$

   (a) Write out the canonical collections of LR(0) items for this grammar (you do not need to add a special start production – it's already there).

   (b) Give an example to explain why an LR(1) parser would have more states than an LALR(1) parser for this grammar.

   (c) Is this grammar SLR(1)? Explain.

   (d) Explain why the grammar is not LL(1), and then convert it to a grammar for the same language that is suitable for LL(1) parsing.

3. (15 points)

   Local and *global common sub-expression elimination* are optimizations frequently performed in compilers.

   (a) What are they?

   (b) What kind of analysis is required to perform global common subexpression elimination?

   (c) Can common sub-expression elimination ever slow down a program? Explain your answer.

4. (10 points)

   It is easier to write a type-checker for function calls if the definition of every function (which specifies the argument types and return types) appears before the first call to the function (whose type correctness depends on its arguments and return types).

   However, if a program language allows mutually recursive functions (e.g., f calls g and g calls f), this is not possible. If f is defined first, a call to g will appear in the body of f – before g is defined – or *vice versa*.

   Give at least two examples of how programming languages and their compilers deal with this issue.