W.L. Ruzzo **CSci 401** **Spring 1998**

Instructions: closed book, closed notes, 50 minutes, 100 points. *It may be too long; don't panic.*

1. (10 points) Why are most compilers broken up into a bunch of little "phases" (lexer, parser, etc.), each doing something relatively simple?

2. (10 points) Why do most compilers separate lexical from syntactic analysis even though context-free grammars are powerful enough to encompass both tasks?

3. (10 points) Give one example (there are many) of a common programming language feature that is easily checked by a context-free parser but would be difficult or impossible to check by a lexical analyzer built acording to the principles we have discussed. Also, give an example of a feature that is difficult or impossible to check in the parser, but is compile-time checkable. In both cases, briefly justify.

4. (10 points) Suppose we wanted to add *character string literals* to the proposed language PL/0.1. A string literal begins and ends with the double quote character (") and may not contain unescaped double quote characters. Any character c may be included by "escaping it" by preceeding it with a backslash (\). I.e., if $\backslash c$ appears in the string literal, then c is included in the string denoted by the string literal. The initial and final double quotes are, of course, excluded from the string denoted by the literal. (In particular, note that by these rules, the only way to include a double quote in a string is to escape it. There is one other character that must always be escaped; what is it?)

   For example, **"abc"**, **"a\bc"**, and **"a\"bc"** are 3 legal string literals denoting the strings **abc**, **abc**, and **a"bc** respectively.

   Give a regular expression defining legal string literals. If you use any meta-notation other than the basic union, concatenation and Kleene star operations, explain it.

   Give a deterministic finite automaton accepting legal string literals. Don't forget to indicate the initial and final states.

5. (30 points) For the grammar below with start symbol E:

   (a) Show the parse tree for the string "**((a)** $\uparrow$ **a)**".

   (b) Compute FIRST for each right hand side and FOLLOW for each nonterminal (in the table below).

Rule				FIRST	FOLLOW
(1)	E	$\rightarrow$	$(\ T\ )$		
(2)	E	$\rightarrow$	**a**		
(3)	T	$\rightarrow$	$E\,T'$		
(4)	T'	$\rightarrow$	$\uparrow T$		
(5)	T'	$\rightarrow$	ϵ		

   (c) Fill in the "parsing table" below, showing for each nonterminal and each lookahead symbol which productions (if any) could be used to expand that nonterminal when parsing a string with that lookahead. Cells you leave blank are assumed to be cases where the parser should signal an error.

Nonterminal	Lookahead Symbol				
	a	(	)	$\uparrow$	**$**
E					
T					
T'					

(d) Is the grammar LL(1)? Why or why not?

(e) Sketch the procedure corresponding to T' in a recursive descent parser for this language. Include a sketch of how the abstract syntax tree is constructed, assuming the $\uparrow$ operator is right-associative, i.e., that $\mathbf{(a{\uparrow}a{\uparrow}a) = (a{\uparrow}(a{\uparrow}a))}$.

6. (10 points) Of the variables `x1, ..., x6` declared below

```
type a1 = array[10] of int;
type a2 = array[10] of int;
type a3 = array[10] of bool;
var x1, x2: array[10] of int;
var x3: a1;
var x4: a1;
var x5: a2;
var x6: a3
```

which have *name-equivalent* types? Which have *structurally-equivalent* types?
(put a "Y" in i,j below if xi is equivalent to xj.)

7. (20 points) Suppose you were writing the type-checking phase of a C compiler, and suppose the following was a portion of a program being compiled.

```
typedef struct {
    int x;
} foo_t;
typedef struct {
    foo_t y;
} bar_t;

int i;
bar_t *barp;
...
i = barp->y.x; /* THIS */
...
```

Recall that in C the indirect selection ("`->`") and direct selection ("`.`") operators have equal precedence, and both are left associative. Sketch the abstract syntax that the parser should produce from the statement marked THIS, indicate the type corresponding to each AST node, and briefly describe the processing your type checker would do in order to determine those types, and indicate the main error conditions it would be looking for in the process.