CSci 401 Introduction to Compilers
Midterm

Spring 1999

Alan Borning

Instructions: closed book, closed notes, 50 minutes, 130 points.

1	/15
2	/25
3	/10
4	/20
5	/40
6	/10
7	/10
Total	/130

1. (15 points) Describe at least two reasonable and feasible choices for writing the lexical analysis and parsing phases of a compiler. Describe the tradeoffs with respect to maintainability, compilation speed, implementation time, need to learn specialized tools, and so forth.

2. (25 points) The lexical-, syntactic-, and semantic analysis phases of compilation each cause parts of the source program to be interpreted in particular ways, and also check certain of the rules of the language being compiled. For each of the following hypothetical rules, say which phase should interpret the source program in accordance with the rule or check the rule. Explain why it is clumsy or impossible to implement in other phases. In case a feature plausibly could be implemented in either of two phases, briefly discuss the tradeoffs between these two alternative implementations. If a feature is implemented in the methods of one of the common data structures shared by the phases, say that as well, and say which phase invokes the methods in question. Assume the three phases are built according to the principles we have discussed.

   (a) The `**` operator is right associative.

   (b) Case is not significant in identifiers.

   (c) Each array subscript has type integer.

   (d) Every `begin` must have a matching `end`.

   (e) In expressions `*` has precedence over `+`.

(f) The type of value returned by a function must match the declared return type.

3. (10 points) Why do most compilers separate lexical from syntactic analysis even though context-free grammars are powerful enough to encompass both tasks? Describe any benefits in terms of maintainability, compilation speed, and implementation time.

4. (20 points) Write a regular expression that define strings consisting of a left square bracket, an even number of `A`'s, and a right square bracket. Thus `[]` and `[AA]` are legal strings but `[A]` is not.

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

5. (40 total points for all parts of this question) Any boolean expression can be put into *conjunctive normal form*, so that the expression consists of a conjunction of disjunctions of literals, where each literal is a variable or the negation of a variable. Less formally, each expression is formed by *and*ing together one or more disjuctions. Each disjunction is formed by *or*ing together one or more literals. A literal is a variable or the result of applying `not` to a variable. We denote *and* by $\wedge$, *or* by $\vee$, and *not* by $\neg$.

   A grammar for expressions in disjunctive normal form is as follows (think C for conjunction, D for disjunction, L for literal, C' for more conjunctions, etc.). C is the start symbol.

$$\begin{array}{lll} C & \rightarrow & D\ C' \\ C' & \rightarrow & \wedge\ D\ C' \ \mid\ \epsilon \\ D & \rightarrow & L\ D' \\ D' & \rightarrow & \vee\ L\ D' \ \mid\ \epsilon \\ L & \rightarrow & \mathbf{id} \ \mid\ \neg\ \mathbf{id} \end{array}$$

   For example, $x \vee \neg y \wedge z$ is an expression in conjunctive normal form, and is parsed so that $\wedge$ has the lowest precedence, followed by $\vee$, and with $\neg$ at the highest precedence. If x, y, and z are all false, then $\neg y$ is true, so $x \vee \neg y$ is true, and finally the whole expression $x \vee \neg y \wedge z$ is false.

   (a) (5 points) Using the grammar give a leftmost derivation for $x \wedge \neg y$.

   (b) (5 points) Using the grammar give a rightmost derivation for $x \wedge \neg y$.

(c) (10 points) Compute FIRST for each right hand side and FOLLOW for each nonterminal (in the table below).

Rule				FIRST	FOLLOW
(1)	C	$\rightarrow$	$D\ C'$		
(2)	C'	$\rightarrow$	$\wedge\ D\ C'$		
(3)	C'	$\rightarrow$	ϵ		
(4)	D	$\rightarrow$	$L\ D'$		
(5)	D'	$\rightarrow$	$\vee\ L\ D'$		
(6)	D'	$\rightarrow$	ϵ		
(7)	L	$\rightarrow$	**id**		
(8)	L	$\rightarrow$	$\neg$ **id**		

(d) (10 points) 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.

	Lookahead Symbol				
Nonterminal	**id**	$\vee$	$\wedge$	$\neg$	**$**
C					
C'					
D					
D'					
L					

(e) (5 points) Is the grammar LL(1)? Why or why not? How is this reflected in the parsing table above?

(f) (5 points) Rewrite the grammar using EBNF so that the only nonterminals are C, D, and L (get rid of C' and D').

6. (10 points) Give an example of a grammar that is LL(3) but not LL(2).

Give an example of a grammar that is not LL(k) for any value of k.

7. (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: a1;
var x2: a1;
var x3: a2;
var x4: a3
```

which have *name-equivalent* types?

Which have *structurally-equivalent* types?