Final Exam

CSC 173

19 December 2000

Directions; PLEASE READ

This exam has 15 questions, many of which have subparts. Each question indicates its point value. The total is 100 points. **Questions 3(b) and 11(c) are for extra credit only**; they are not part of the 100 points.

Please show your work here on the exam, in the space given. Do not write on the backs or in the margins. Put your name on every page. If your answer won't fit in the given space then you're trying to write too much. Scratch paper is available if you need it, but I will collect **only the exams**.

I have tried to make the questions as clear and self-explanatory as possible. If you do not understand what a question is asking, make some reasonable assumption and *write that assumption down* next to your answer. I will decline to answer questions during the exam.

You will have a maximum of three hours to work. Good luck!

1. (a) (4 points) Do we (1) build data structures out of abstract data types, or (2) implement abstract data types using data structures?

   We implement abstract data types using data structures.

   (b) (4 points) Is a relation a data structure or an abstract data type? How about a hash table?

   A relation is an abstract data type; a hash table is a data structure.

2. A context-free grammar is said to be *ambiguous* if and only if some string in the language has more than one parse tree.

   (a) (5 points) Express this statement in predicate logic, using the following predicates.

ambiguous (G)	grammar G is ambiguous
parse_tree_for (T, S, G)	T is a parse tree for string S under grammar G
not_equal (T1, T2)	trees T1 and T2 differ

   $(\forall G)[ambiguous(G) \Leftrightarrow (\exists S)(\exists T_1)(\exists T_2)[parse\_tree\_for(T1, S, G) \wedge parse\_tree\_for(T2, S, G) \wedge not\_equal(T1, T2)]]$.

   (b) (5 points) Why is ambiguity undesirable in grammars used by computers?

   Because an ambiguous grammar is hard to parse. It certainly can't be parsed deterministically in linear time. (Of course some unambiguous grammars can't be

parsed deterministically in linear time either.) Also, since we commonly associate "meaning" with parse trees, an ambiguous grammar suggests the existence of a string with more than one meaning, which is certainly undesirable if the string is a computer program.

3. (a) (5 points) Convert the following propositional logic statement into conjunctive normal form (ANDs on the outside, ORs on the inside, NOTs applied only to individual propositions): $(\mathrm{A} \rightarrow \mathrm{B}) \vee \neg(\mathrm{C} \vee \mathrm{D})$.

   $(B \vee \neg A \vee \neg C) \wedge (B \vee \neg A \vee \neg D)$.

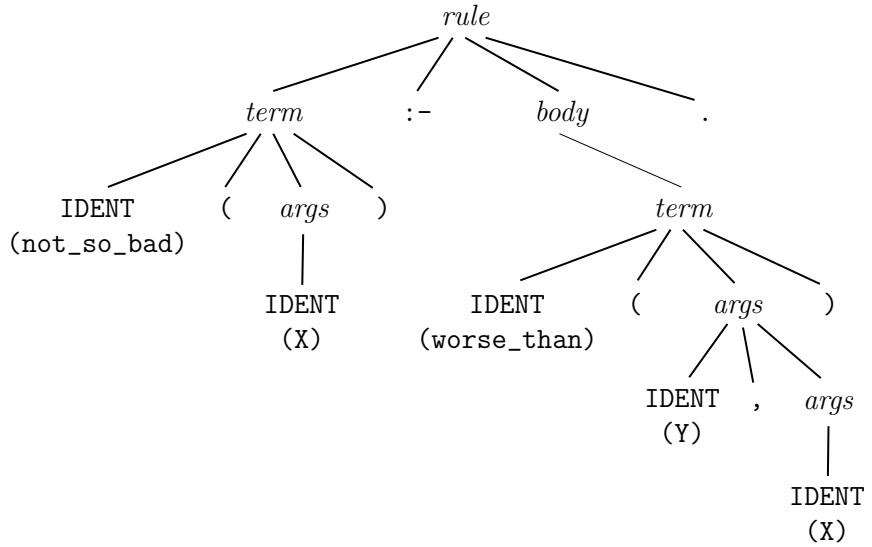
   (b) (Extra credit, up to 5 points) Express the same thing using Horn clauses.

   $B \leftarrow A \wedge C$
   $B \leftarrow A \wedge D$

4. (8 points) Consider the following partial grammar for Prolog rules:

   rule ⟶ *term* `:-` *body* `.`
   term ⟶ `IDENT (` *args* `)`
   args ⟶ `IDENT`
   args ⟶ `IDENT ,` *args*
   body ⟶ *term*
   body ⟶ *term* `,` *body*

   Give a parse tree for the string `not_so_bad(X) :- worse_than(Y,X).`



5. (5 points) Restate the Prolog rule (example token string) of the previous question in predicate logic, using quantifiers.

   $(\forall X)[not\_so\_bad(X) \leftarrow (\exists Y)[worse\_than(Y, X)]]$ *or, for all X, X is not so bad if there exists a Y that is worse.*

6. (4 points) Which of the following Java constructs *cannot* be recognized with a DFA? (Circle all that cannot.)

   - arithmetic expressions
   - floating point constants
   - class declarations
   - character strings containing escape sequences

   Arithmetic expressions and class declarations.

7. (5 points) What does it mean when a recursive descent parsing subroutine returns without doing anything (no matches, no recursive calls)?

   It means we predicted an epsilon production.

8. (5 points) Informally, FIRST(A) is defined to be the set of tokens that can begin some string of tokens derived from A. Compute FIRST(*stmt*) for the following grammar fragment:

   stmt $\longrightarrow$ *label_opt loop*
   stmt $\longrightarrow$ `IDENT :=` *expr*
   label_opt $\longrightarrow$ `NUMBER :`
   label_opt $\longrightarrow \epsilon$
   loop $\longrightarrow$ `WHILE` *condition* { *stmt_list* }
   loop $\longrightarrow$ `DO` { *stmt_list* } `UNTIL` *condition*

   {*NUMBER, IDENT, WHILE, DO*}.

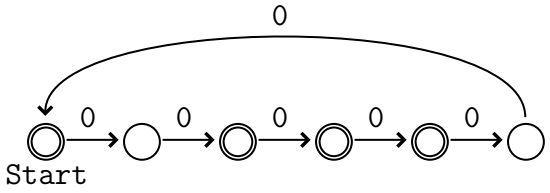
9. (5 points) We could use a relation R with the scheme (Tail, Head) to represent the edges of a directed graph. Give a relational algebra query to return the neighbors of node V.

   $\pi_{Head}(\sigma_{Tail=V}(\mathrm{R}))$.

10. Describe in English the languages generated or recognized by the following formalisms. Note: you will *not* get full credit just for saying how the formalism works. For example given the regular expression `(1|01*0)*` you would need to say "binary strings containing an even number of zeros", not "a one, or a pair of zeros separated by arbitrarily many ones, repeated arbitrarily many times".

    (a) (5 points) the regular expression `(a|b)* (b|c)*`.
    Strings of a's, b's, and c's in which all a's precede all c's.

(b) (5 points) the finite automaton



Strings of zeros whose length is evenly divisible by 2 or 3.

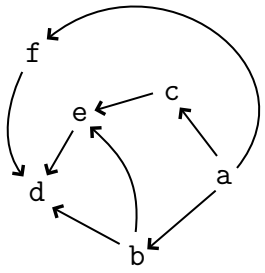
(c) (5 points) the context-free grammar

arrow $\longrightarrow$ `>>` *arrow* `>`
arrow $\longrightarrow$ `>>` *shaft* `>`
shaft $\longrightarrow$ `-`
shaft $\longrightarrow$ `-` *shaft* `-`

(Here the "greater than" and minus signs are tokens. The grammar generates strings that look like long arrow symbols. You need to say more precisely what sorts of arrows.)

Arrows in which there are twice as many tail feathers as barbs on the head, and a shaft of odd length.

11. Consider the following graph:



(a) (5 points) List the nodes in the order they would be discovered by depth first search, starting at node **a** (order neighbors alphabetically when necessary to break ties).

a b d e c f.

(b) (5 points) List the nodes in the order they would be discovered by breadth first search, again starting at **a** and ordering neighbors alphabetically.

a b c f d e.

(c) (Extra credit, up to 6 points) Give two different topological sorts of this graph.

There are eight possibilities:

a b c e f d	*a c b e f d*	*a f b c e d*
a b c f e d	*a c b f e d*	*a f c b e d*
a b f c e d	*a c f b e d*	

12. (5 points) Name and briefly describe a graph problem that is NP hard (i.e. for which the best known algorithm takes exponential time).

There are many. In class I mentioned

minimal coloring *– What is the smallest number of colors such that we can assign a color to every node in a given undirected graph and never have two neighbors colored the same?*

traveling salesman *– What is the cheapest cycle through a given weighted directed graph that visits every node?*

hamiltonian path *– Is there a path through a given directed graph that visits every node exactly once?*

maximal clique *– What is the largest completely-connected subgraph of a given graph?*

13. (5 points) Consider the following code, which is syntactically valid in both Java and C++:

```
// assume b.f == 1
my_class a = b;
a.f = 2;
// what is b.f now?
```

What is the answer to the commented question in Java? in C++? Explain.

In Java it is 2; in C++ it is 1. Variables of class type in Java are references to objects; variables of class type in C++ are named containers for objects. The initialization in the second line in Java makes `a` *refer to the same object* `b` *refers to; in C++ it makes* `a` *a copy of* `b`*.*

14. (5 points) Why do you usually have to apply a cast to the return values of methods of standard library collections (containers) in Java, but not in C++?

The C++ standard library uses templates to create type-specific containers. The Java standard library creates collections capable of holding objects of arbitrary (heterogeneous) types, all of which are returned as references of class `Object`*.*

15. (5 points) Give a reason why it might sometimes be better to implement an index for a relation with a balanced search tree instead of a hash table.

Possible answers: (1) Search trees support efficient range queries (e.g. find all tuples in which the key is larger than 100); hash tables do not. (2) Search trees are naturally dynamically resizable; changing the size of a hash table is an expensive ($O(n)$) operation. (3) If your hash function leads to many collisions for your data set, and for some reason you can't use a different hash function (unlikely but conceivable), then lookups in a tree could be faster ($O(\log n)$ instead of $O(n)$).