Sample Questions for Midterm 2 (CS 421 Fall 2011)

On the actual midterm, you will have plenty of space to put your answers.
Some of these questions may be reused for the exam.

1. Give a (most general) unifier for the following unification instance. Capital letters denote variables of unification. Show your work by listing the operation performed in each step of the unification and the result of that step.

   {X = f(g(x),W); h(y) = Y; f(Z,x) = f(Y,W)}

2. For each of the following descriptions, give a regular expression over the alphabet {a,b,c}, and a regular grammar that generates the language described.
   a. The set of all strings over {**a**, **b**, **c**}, where each string has at most one **a**

   b. The set of all strings over {**a**, **b**, **c**}, where, in each string, every **b** is immediately followed by at least one **c**.

   c. The set of all strings over {**a**, **b**, **c**}, where every string has length a multiple of four.

3. Consider the following grammar:

   <S> ::= <A> | <A> <S>
   <A> ::= <Id> | (<B>
   <B> ::= <Id>] | <Id><B> | (<B>
   <Id> ::= 0 | 1

   For each of the following strings, give a parse tree for the following expression as an <S>, if one exists, or write "No parse" otherwise:

   a. (0 1 (1] ((1 0] 1
   b. 0 (1 0 (1]
   c. (0 (1 0 1] 0]

4. Demonstrate that the following grammar is ambiguous (Capitals are non-terminals, lowercase are terminals):

   S ::= A a B | B a A
   A ::= b | c
   B ::= a | b

5. Write an unambiguous grammar generating the set of all strings over the alphabet {0, 1, +, -}, where + and – are infixed operators which both associate to the left and such that + binds more tightly than -.

6. Write a recursive descent parser for the following grammar:,

   <S> ::= <N> % <S> | <N>
   <N> ::= g <N> | a | b

   You should include a datatype **token** of tokens input into the parser, one or more datatypes representing the parse trees produced by parsing (the abstract syntax trees), and the

function(s) to produce the abstract syntax trees. Your parser should take a list of tokens as input and generate an abstract syntax tree corresponding to the parse of the input token list.

7. Why don't we ever get shift/shift conflicts in LR parsing?

8. Consider the following grammar with terminals ***, f, x,*** and **y**, and **eol** for "end of line", and non-termimals **S, E** and **N** and productions

```
(P0)  S => E eol
(P1)  E => E * N
(P2)  E => N
(P3)  N => f N
(P4)  N => x
(P5)  N => y
```

The following are the Action and Goto tables generated by YACC for the above grammar:

	ACTION					GOTO		
STATE	*****	**f**	**x**	**y**	**eol**	**S**	**E**	**N**
1		s3	s4	s5		2	6	7
2					acc			
3		s3	s4	s5				8
4	r4	r4	r4	r4	r4			
5	r5	r5	r5	r5	r5			
6	s9				acc			
7	r2	r2	r2	r2	r2			
8	r3	r3	r3	r3	r3			
9		s3	s4	s5				10
10	r1	r1	r1	r1	r1			

where *si* is shift and stack state *i*, *rj* is reduce using production *Pj*, *acc* is accept. The blank cells should be considered as labeled with error. The empty "character" represents end of input. Describe how the sentence **fx*y<eol>** would be parsed with an LR parser using this table. For each step of the process give the parser action (shift/reduce), input and stack state.

9. Describe a complete evaluation of (**{x=5; y=2}, if x > 3 then y:= x+y else y:=3 fi)** using each of structural operational semantics (aka natural semantics) and transition semantics, as given in class.
10. If we added to the Simple Imperative Programming Language described in class a post-fix ++ operator applied to identifiers in expressions, where I++ returned the value of I in the input memory, but had the side effect of incrementing that value in memory, what would the rules for **if_then_else_** become in each of structural operational semantics and transition semantics?