There are two parts to this exam. Show all your work. The work has to be your own. No collaboration is allowed. However, you can use the course materials such as books and notes. If you do use work from the internet then you have to mark it clearly and there is the possibility that I will deduct points.

Part I: Problems.

1. (5 points) Given the natural deduction and Prolog semantics discussed in class, how do the state representations in the respective semantics differ? What is the effect of this difference on the style of semantic definition?

2. (5 points) What is meant by a well-founded relation and why is this important to the semantics of programming languages?

3. (5 points) For any choice of $be \in \text{Bexp}$, and $c, c' \in \text{Com}$, which of the following pairs of commands are semantically equivalent and which are not? Justify your answer informally in each case based on the natural deduction semantics discussed in class.
   1. `while be do c end ; c'` and `if be then while be do c end ; c else c end`
   2. `c ; while be do c end' and `if be then while be do c end else c end'
   3. `while be do if be then c else c' end end' and `while be do c c end'

4. Given the grammar $G$,
   
   $E \rightarrow E + E | E \ast E | a | b | c$

   (a) (5 points) What does the set $L(G)$ look like?
   (b) (10 points) Prove that for all $e \in L(G)$, $N(e) > O(e)$, where $N(e)$ is the number of names (e.g., $a$) and $O(e)$ is the number of operators (e.g., $+$) in expression $e$.
   (c) (10 points) If we extend our grammar with an additional operator,
   
   $E \rightarrow \sim E | E + E | E \ast E | a | b | c$

   why and where does your proof of $N(e) > O(e)$ break down?

5. (10 points) Given the natural deduction semantics discussed in class, assume that assignment and the while loop are not part of the specification, prove by structural induction that every program in this language is equivalent to the program $\text{skip}$, i.e. $c \sim \text{skip}$, for all $c \in \text{Com}$. 
Part II: programming language specification

1. Consider grammar for the simple stack machine programming language:

```
C ::= skip
    | push(x)
    | push(n)
    | push(true)
    | push(false)
    | pop
    | pop(x)
    | duplicate
    | put
    | get
    | add
    | sub
    | mult
    | eq
    | le
    | not
    | and
    | or
    | seq(C,C)
    | if(C,C)
    | whiledo(C)
```

with the following informal semantics:

- **skip** - leaves the state of the machine unchanged.
- **push(x)** - push the value of variable x onto the stack.
- **push(n)** - push the integer value n onto the stack.
- **push(true)/push(false)** - push a boolean constant on the stack.
- **pop** - discard the top of the stack.
- **pop(x)** - pop the value on the top of the stack into variable x.
- **duplicate** - duplicate the top of the stack.
- **put** - write the value on the top of the stack to the terminal output. Pop the stack.
- **get** - get a value from the terminal and push it on the stack.
- **add/sub/mult/eq/le/and/or** - apply the operator to the two values on the top of the stack, pop them off the stack, push the result back onto the stack. NOTE: top of the stack → right operand, second top of the stack → left operand.
- **not** - apply the logical not to the top of the stack.
- **seq(C,C)** - composition operator of two statements.
- **if(C1,C2)** - if the top of the stack is true, pop the stack and execute the first command C1; if the top of the stack if false, pop the stack and execute the second command C2.
- **whiledo(C)** - if the top of the stack is true, pop the stack and execute C seq whiledo(C); if the top of the stack if false, pop the stack.

A state in this machine is now a pair consisting of the binding environment and the machine stack. A pair in Prolog can be represented by the term structure ‘(Binding, Stack)’. A stack can be easily implemented in Prolog using the list term structure. For example:
is_empty([]).
pop_stack(Stack,Value,NewStack) :- Stack = [H|T], Value = H, NewStack = T.
push_stack(Value,Stack,NewStack) :- NewStack = [Value|Stack].

Note, that this definition does not allow you to pop empty stacks which is the usual behavior of stacks.

Answer the following questions:
(a) (20 points) Implement a Prolog semantics for this language.
(b) (10 points) Describe your state implementation and highlight/describe any difficulties or compromises you had to make in this implementation.
(c) (10 points) Show that your semantics works with some simple examples.
(d) (10 points) Write a proof score that shows that addition is commutative, that is, that
push(a) seq push(b) seq add  ~  push(b) seq push(a) seq add
where a and b are either variables or integer values.

Please hand in a hardcopy of your semantics as well as the proofs and any screen captures.