Problem:
The idea of this exercise is to provide a formal semantics for a small snippet of the Forth programming language. This is a nice primer for the basics of Forth:

http://galileo.phys.virginia.edu/classes/551.jvn.fall01/primer.htm

Forth is a high-level, stack-based programming language. Here is a very simple Forth program:

```
1 2 + .
```

Informal interpretation of this program is: push 1 on the stack, push 2 on the stack, add the top two numbers on the stack and push the result back onto the stack, and DOT means pop the top of the stack and print it.

Forth also supports variable and function definitions:

```
variable x 3 x ! x x @ x @ + .
```

Informal interpretation: declare variable `x` in the `dictionary`, push 3 on the stack, push the name of the variable on the stack, `!` pops the top and the variable name off the stack, and binds the value to the variable in the dictionary, push the name of the variable on the stack, `@` pops the name of the variable off the stack on pushes its bound value on the stack, push the name of the variable on the stack, pop the variable name and push the bound value on the stack. Here is an example with a function and a while loop:

```
: loop begin dup 0 > while dup . 1 − repeat ; 10 loop
```

This program defines the function loop which expects a positive integer on the stack as an argument. It then `duplicates` the integer on the stack and tests whether it is greater than 0. While the loop counter on the stack is greater than 0 we print out the loop counter and then subtract 1 from it. We end the function definition with `;`. We then push 10 on the stack and call the function loop.

The following is the grammar of the snippet of Forth we are considering. The grammar is a bit more abstract than actual Forth because we have to make the Prolog parser happy. Notice that the grammar enforces that function and variable declarations can only happen at the program level (global level); no nested scopes allowed! Also notice that in Forth the binding environment is called the `dictionary`. Here we model a list of Forth commands as an actual Prolog list of command terms.

```
PROG ::= [ CMDS ]
       | [ ]

CMDS ::= CMD
       | CMD , CMDS
```
CMD ::= DICTIONARY
    | COMPUTE

DICTIONARY ::= variable(x)
    | define(f,BODY)

BODY ::= [ BODYCMDS ]
    | []

BODYCMDS ::= COMPUTE
    | COMPUTE , BODYCMDS

COMPUTE ::= VAL
    | add
    | sub
    | mult
    | and
    | or
    | invert
    | eq
    | le
    | dot
    | bang
    | at
    | dup
    | if(BODYCMDS)
    | while(BODYCMDS)
    | call(f)

VAL ::= x | n | true | false

with the following informal semantics:

variable(x) - declares a new variable x in the dictionary, leaves the stack untouched.

define(f,C) - declares a function f with body C in the dictionary; leaves the stack untouched. Any arguments
the function uses are assumed to have been pushed on the stack. Any return values from the function
are assumed to be pushed on the stack by the function.

x - push the name of the variable x onto the stack.

n,true,false - push the value of the constant onto 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.

invert - apply the logical not to the top of the stack.

dot - print the top of the stack to the terminal output and pop the top of the stack.

bang - assume that the top of the stack is a variable name, assume the second to top of stack is a value,
store the value in the dictionary at location of the variable name, pop both values off the stack.

at - assume the top of the stack is a variable name, pop the variable name and push the value of the variable
in the dictionary onto the stack.
duplicate - duplicate the top of the stack.

if(C) - if the top of the stack is true, pop the stack and execute the command C; if the top of the stack if false, pop the stack and continue execution.

while(C) - if the top of the stack is true, pop the stack and execute C then loop again; if the top of the stack if false, pop the stack and continue execution.

call(f) - execute the body of the function f.

Answer the following questions:
(a) (20 points) Implement a Prolog semantics for this language.
(b) (20 points) Show that it works with some examples. In particular, show that your definition works with the following programs,

\[1,2,\text{add},\text{dot}\]

\[\text{variable}(x),3,x,\text{bang},x,\text{at},x,\text{at},\text{add},\text{dot}\]

\[10,\text{dup},0,\text{eq},\text{invert},\text{while}([\text{dup},\text{dot},1,\text{sub},\text{dup},0,\text{eq},\text{invert}])\]

\[\text{define}(\text{inc},[1,\text{add}]),\text{variable}(x),10,\text{call}(\text{inc}),x,\text{bang},x,\text{at},\text{dot}\]

(c) (10 points) Write a proof score that shows that addition is commutative, that is, that

\[\text{[a,b,add]} \sim \text{[b,a,add]}\]

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.