CMSC 330 Exam #2 practice questions #2 Fall 2012

Do not open this exam until you are told. Read these instructions:

1. This is a closed book exam. **No notes or other aids are allowed.**
2. **You must turn in your exam immediately when time is called at the end.**
3. This exam contains 6 pages, including this one. **Make sure you have all the pages.** Each question's point value is next to its number. **Write your name on the top of all pages before starting the exam.**
4. In order to be eligible for as much partial credit as possible, show all of your work for each problem, and **clearly indicate** your answers. Credit **cannot** be given for illegible answers.
5. If you finish at least 15 minutes early, bring your exam to the front when you are finished; otherwise, wait until the end of the exam to turn it in. Please be as quiet as possible.
6. If you have a question, raise your hand. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn't need to do this at all, so be careful when making assumptions.
7. If you need scratch paper during the exam, please raise your hand. Scratch paper must be turned in with your exam, with your name and ID number written on it. Scratch paper **will not** be graded.
8. Small syntax errors will be ignored in any code you have to write on this exam, as long as the concepts are correct.
9. The Campus Senate has adopted a policy asking students to include the following handwritten statement on each examination and assignment in every course: "*I pledge on my honor that I have not given or received any unauthorized assistance on this examination.*" Therefore, **just before turning in your exam**, you are requested to write this pledge **in full** and **sign it** below:

Good luck!

1. Short answer:

   a. Below is the output of the OCaml interpreter after we've typed in various definitions of the function `f`. In each case, write a definition of a function `f` that has that type.

      i. `val f : 'a -> 'b -> 'a * 'a * 'b = <fun>`

      ii. `val f : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c = <fun>`

      iii. `val f : 'a -> 'b = <fun>`

   b. Define what it means for one name to *shadow* another, and give an example of shadowing in the language of your choice.

   c. Suppose in C we define the following function:

```
int and(int x, int y) {
  return x && y;
}
```

   Is it ever possible that calling `and(`e_1`, `e_2`)` behaves differently than evaluating e_1 `&&` e_2 directly, where e_1 and e_2 are C expressions? Explain, and give an example if so.

2. Consider the short grammar below that generates expressions in something that is called the simply–typed lambda calculus (which we will not explain further here):

   $$\begin{aligned} \mathrm{S} &\longrightarrow \mathrm{U} \mid \mathrm{V} \mid \lambda \mathrm{V}:\mathrm{T}.\mathrm{S} \mid \mathrm{S}\ \mathrm{S} \\ \mathrm{T} &\longrightarrow \texttt{int} \mid \mathrm{T} \to \mathrm{T} \\ \mathrm{U} &\longrightarrow \texttt{1} \mid \texttt{2} \mid \texttt{3} \\ \mathrm{V} &\longrightarrow \texttt{x} \mid \texttt{y} \mid \texttt{z} \end{aligned}$$

   We assume that the only integers are just `1`, `2`, and `3`, and that the only variable names are just `x`, `y`, and `z`.

   Prove that this grammar is ambiguous.

3. Evaluating Java bytecode: In this problem you will write an OCaml function to evaluate Java bytecodes. The Java Virtual Machine, which executes bytecodes, is *stack–based*, so that bytecode instructions operate on a stack, pushing and popping values, which for this problem will only be integers. Java bytecodes can also access a set of *local variables*, which are numbered.

   The following table describes the eight opcodes you will implement for this problem. The right–most column is a succinct picture of what the instructions do. Stacks are written with the top of the stack to the left and ... used to indicate the portion of the stack that doesn't change. V is the local variable lookup table, and $V(n)$ refers to the value which variable number n maps to in V. The $\Rightarrow$ means that the initial stack's contents are shown to the left of the arrow, and the stack's contents after the instruction is executed are shown to the right of the arrow. So, for example, the notation "$\ldots \Rightarrow V(n), \ldots$" in the 2nd to last row of the table means "look up n in the table V, and push the result onto the stack." The last row's notation means "remove the top value from the stack, and store whatever that was in $V(n)$."

Instruction	Description	Example
`iconst` x	Push the value x onto the stack	$\ldots \Rightarrow x, \ldots$
`iadd`	Add two ints on top of the stack	$x, y, \ldots \Rightarrow x + y, \ldots$
`dup`	Duplicate the value on the top of the stack	$x, \ldots \Rightarrow x, x, \ldots$
`nop`	Do nothing	$\ldots \Rightarrow \ldots$
`pop`	Pop the top value off of the stack and discard it	$x, \ldots \Rightarrow \ldots$
`swap`	Swap the top two operand stack values	$x, y, \ldots \Rightarrow y, x, \ldots$
`iload` n	Push the value in local variable #n onto the stack	$\ldots \Rightarrow V(n), \ldots$
`istore` n	Pop the top value off of the stack and store it in local variable #n	$x, \ldots \Rightarrow \ldots \qquad V(n) := x$

   Below we give an OCaml data type `instr` representing Java bytecodes. You job is to write two functions. First, write:

```
eval_instr : int list * instr -> int list
```

   which, given a stack and an instruction, returns a new stack according to the behavior of the instruction. The stack is a list of integers with the head of the list as the top of the stack. Then write:

```
eval : int list * instr list -> int list
```

   which, given a stack and a list of instructions, returns a new stack which is the result of evaluating each of the instructions in turn, in order from the first instruction in the list to the end of the list. (You should find this second function much easier to write.)

   - It is an error to try to read a local variable that has not been written to, and your solution may do anything if an instruction tries to do this.
   - It is an error to try to pop from an empty stack, and your solution may do anything if an instruction tries to do this.
   - You **may not use OCaml library functions or arrays** for this question. In particular, if you choose to use associative lists to represent V, then you will need to give code that implements them.

   Here is the data type for instructions:

```
type instr =
    IConst of int
  | IAdd
  | Dup
  | Nop
  | Pop
  | Swap
  | ILoad of int
  | IStore of int
```

Here is a sample run of `eval`:

```
# let l = [IConst 3; IConst 4; IAdd];;
val l : instr list = [IConst 3; IConst 4; IAdd]
# eval ([], l);;
- : int list = [7]
```

Write your functions `eval_instr` and `eval`, as well as any other functions or declarations you may need, here: