CMSC 330 Exam #2 practice questions #2– solutions Fall 2012

1. a. i. `let f x y = (x, x, y)`

   ii. `let f x y z = y (x z)`

   iii. `let rec f x = f x`

   Notice that because `f` never terminates, its result type could be anything.

   b. Shadowing occurs when one name hides another name that would otherwise be in scope. For example, in OCaml, the inner `x` shadows the outer `x` in the following function:

```
let f x = (fun x -> x + x)
```

   Here's an example in C:

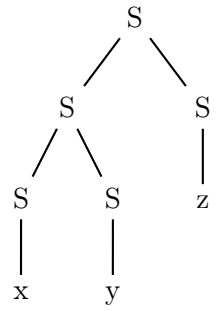
```
void f(int x) {
  int x = 3;

  x = x + x;
}
```

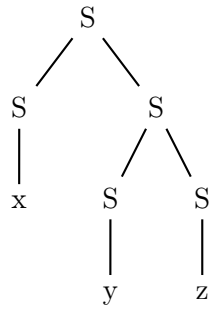
   c. Yes. e_1 `&&` e_2 is short-circuiting and won't evaluate e_2 if e_1 is false. But functions are call–by–value, so `and(`e_1`,` e_2`)` will always evaluate both e_1 and e_2. For example, suppose we define `int f(void) { printf("hello"); return 3; }`. Then `0 && f(x)` will not print anything, but `and(0, f(x))` will.

2. There are two sources of ambiguity in this grammar. One is the production S $\longrightarrow$ S S, the other is the production T $\longrightarrow$ T $\rightarrow$ T.

   Ambiguity could be demonstrated by showing two leftmost or two rightmost derivations for the same string (using either source of ambiguity), or two parse trees for the same string (which may have been quicker than writing two derivations).

   Here is one ambiguity, shown by having two parse trees for the same string "x y z":





   Two leftmost or two rightmost derivations could be shown for the same string, or for any other ambiguous string, as well. The two leftmost derivations for this string are:

   S $\Longrightarrow$ S S $\Longrightarrow$ S S S $\Longrightarrow$ x S S $\Longrightarrow$ x y S $\Longrightarrow$ x y z

   and

   S $\Longrightarrow$ S S $\Longrightarrow$ x S $\Longrightarrow$ x S S $\Longrightarrow$ x y S $\Longrightarrow$ x y z

3. Here is one solution:

```
(* returns the value corresponding to the first occurrence of n in
   association list *)
let rec lookup n = function
    [] -> raise Not_found  (* optional *)
  | ((k, v)::t) -> if n = k then v else lookup v t

let v = ref []

let eval_instr = function
    (s, IConst x) -> x::s
  | (x::y::s, IAdd) -> (x + y)::s
  | (x::s, Dup) -> x::x::s
  | (s, Nop) -> s
  | (x::s, Pop) -> s
  | (x::y::s, Swap) -> y::x::s
  | (s, ILoad n) -> (lookup n (!v))::s
  | (x::s, IStore n) ->
      begin
        v := (n, x)::(!v);  (* add new value of n *)
        s                   (* evaluate to result stack *)
      end

let rec eval = function
    (s, []) -> s
  | (s, i::is) -> eval (eval_instr (s, i), is)
```