CMSC 330, Spring 2009, Quiz 3 Practice Problems

1. OCaml Polymorphic Types

Consider a OCaml module Bst that implements a binary search tree:

```
module Bst = struct
  type bst =
    Empty
    | Node of int * bst * bst

  let empty = Empty              (* empty binary search tree          *)

  let is_empty = function        (* return true for empty bst         *)
    Empty -> true
  | Node (_, _, _) -> false

let rec insert n = function      (* insert n into binary search tree  *)
   Empty -> Node (n, Empty, Empty)
 | Node (m, left, right) ->
    if m = n then Node (m, left, right)
    else if n < m then Node(m, (insert n left), right)
    else Node(m, left, (insert n right))

(* Implement the following functions
          val min : bst -> int
          val remove : int -> bst -> bst
          val fold : ('a -> int -> 'a) -> 'a -> bst -> 'a
          val size : bst -> int
        *)
   let rec min =                 (* return smallest value in bst      *)
   let rec remove n t =          (* tree with n removed               *)
   let rec fold f a t =          (* apply f to nodes of t in inorder  *)
   let size t =                  (* # of non-empty nodes in t         *)
  end
```

a. Is insert tail recursive? Explain why or why not.
b. Implement min as a tail-recursive function. Raise an exception for an empty bst. Any reasonable exception is fine.
c. Implement remove. The result should still be a binary search tree.
d. Implement fold as an inorder traversal of the tree so that the code

```
List.rev (fold (fun a m -> m::a) [] t)
```

will produce an (ordered) list of values in the binary search tree.
e. Implement size using fold.

2. Recursive Descent Parser in OCaml

The example OCaml recursive descent parser 15-parseArith_fact.ml employs a number of shortcuts. For instance, the function parseS handles the grammar rules for

S → T + S | T

directly instead of first applying left factoring:

S → T A A → + S | epsilon

However, we can still identify where code corresponding to parseA was inserted directly in the code for parseS, in the comments below:

```
let rec parseS lr =                          (* parseS *)
  let x = parseT lr in                       (* S → T A *)
  match !lr with                             (* parseA *)
    | ('+'::t) ->                            (* if lookahead = First( + S ) *)
      lr := t;                               (* A → + S *)
      Sum (x,parseS lr)
    | _ -> x                                 (* A → epsilon *)
```

Similarly, the function parseF handles the grammar rules for

F -> U ! | U

directly instead of rewriting the grammar, creating the following productions:

F -> ? B -> ?

You must identify where code corresponding to parseB was inserted directly in the code for parseF in the comments below:

```
let rec parseF lr =                          (* parseF *)
  let rec fHelper lr tmp =
   match !lr with                            (* parseB *)
     | ('!'::t) ->                           (* 1: if lookahead = First( ? ) *)
       lr := t;                              (* 2: ? → ? *)
       Fact (fHelper lr tmp)
     | _ -> tmp                              (* 3: ? →  ? *)
  in let x = parseU lr in (fHelper lr x)     (* 4: ? →  ? *)
```

a. What rule should have been applied to the productions for F?
b. What productions for F & B would be created by applying the rule?
c. What sentential form should appear in place of ? in comment 1?
d. What production should appear in place of ? in comment 2?
e. What production should appear in place of ? in comment 3?
f. What production should appear in place of ? in comment 4?

3. Context Free Grammars
   a. List the 4 components of a context free grammar.
   b. Describe the relationship between terminals, non-terminals, and productions.
   c. Define ambiguity.
   d. Describe the difference between scanning & parsing.

4. Describing Grammars
   a. Describe the language accepted by the following grammar:

      S → abS | a

b. Describe the language accepted by the following grammar:
   $S \rightarrow aSb \mid \varepsilon$
c. Describe the language accepted by the following grammar:
   $S \rightarrow bSb \mid A$ $\quad$ $A \rightarrow aA \mid \varepsilon$
d. Describe the language accepted by the following grammar:
   $S \rightarrow AS \mid B$ $\quad$ $A \rightarrow aAc \mid Aa \mid \varepsilon$ $\quad$ $B \rightarrow bBb \mid \varepsilon$
e. Describe the language accepted by the following grammar:
   $S \rightarrow$ S and S | S or S | (S) | true | false
f. Which of the previous grammars are left recursive?
g. Which of the previous grammars are right recursive?
h. Which of the previous grammars are ambiguous? Provide proof.
i. Write a grammar for $a^x b^y$, where $x = y$
j. Write a grammar for $a^x b^y$, where $x > y$
k. Write a grammar for $a^x b^y$, where $x = 2y$
l. Write a grammar for $a^x b^y a^z$, where $z = x+y$
m. Write a grammar for $a^x b^y a^z$, where $z = x-y$
n. Write a grammar for all strings of *a* and *b* that are palindromes.
o. Write a grammar for all strings of *a* and *b* that include the substring *baa*.
p. Write a grammar for all strings of *a* and *b* with an odd number of *a*'s and *b*'s.
q. Write a grammar for the "if" statement in OCaml
r. Write a grammar for all lists in OCaml
s. Which of your grammars are ambiguous? Can you come up with an unambiguous grammar that accepts the same language?

5. Derivations, Parse Trees, Precedence and Associativity

   For the following grammar: $S \rightarrow$ S and S | true
   a. List 4 derivations for the string "true and true and true".
   b. Label each derivation as left-most, right-most, or neither.
   c. List the parse tree for each derivation
   d. What is implied about the associativity of "and" for each parse tree?

   For the following grammar: $S \rightarrow$ S and S | S or S | true
   e. List all parse trees for the string "true and true or true"
   f. What is implied about the precedence/associativity of "and" and "or" for each parse tree?
   g. Rewrite the grammar so that "and" has higher precedence than "or" and is right associative

6. Left factoring & eliminating left recursion

   Rewrite the following grammars so they can be parsed by a predicative parser by eliminating left recursion and applying left factoring where necessary
   a. $S \rightarrow S + a \mid b$
   b. $S \rightarrow S + a \mid S + b \mid c$
   c. $S \rightarrow S + a \mid S + b \mid \varepsilon$
   d. $S \rightarrow a\ b\ c \mid a\ c$