

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
```

- Is insert tail recursive? Explain why or why not.
- Implement min as a tail-recursive function. Raise an exception for an empty bst. Any reasonable exception is fine.
- Implement remove. The result should still be a binary search tree.
- 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.
- 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

$$S \rightarrow T + S \mid T$$

directly instead of first applying left factoring:

$$S \rightarrow T A \quad A \rightarrow + S \mid \text{epsilon}$$

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

$$F \rightarrow U \mid U$$

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

$$F \rightarrow ? \quad B \rightarrow ?$$

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: ? → ? *)
```

- What rule should have been applied to the productions for F?
- What productions for F & B would be created by applying the rule?
- What sentential form should appear in place of ? in comment 1?
- What production should appear in place of ? in comment 2?
- What production should appear in place of ? in comment 3?
- What production should appear in place of ? in comment 4?

3. Context Free Grammars

- List the 4 components of a context free grammar.
- Describe the relationship between terminals, non-terminals, and productions.
- Define ambiguity.
- Describe the difference between scanning & parsing.

4. Describing Grammars

- Describe the language accepted by the following grammar:

$$S \rightarrow abS \mid a$$

- b. Describe the language accepted by the following grammar:
 $S \rightarrow aSb \mid \varepsilon$
- c. Describe the language accepted by the following grammar:
 $S \rightarrow bSb \mid A$ $A \rightarrow aA \mid \varepsilon$
- d. Describe the language accepted by the following grammar:
 $S \rightarrow AS \mid B$ $A \rightarrow aAc \mid Aa \mid \varepsilon$ $B \rightarrow bBb \mid \varepsilon$
- e. Describe the language accepted by the following grammar:
 $S \rightarrow S \text{ and } S \mid S \text{ or } S \mid (S) \mid \text{true} \mid \text{false}$
- 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: $S \rightarrow S \text{ and } S \mid \text{true}$
- List 4 derivations for the string "true and true and true".
 - Label each derivation as left-most, right-most, or neither.
 - List the parse tree for each derivation
 - What is implied about the associativity of "and" for each parse tree?
- For the following grammar: $S \rightarrow S \text{ and } S \mid S \text{ or } S \mid \text{true}$
- List all parse trees for the string "true and true or true"
 - What is implied about the precedence/associativity of "and" and "or" for each parse tree?
 - 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
- $S \rightarrow S + a \mid b$
 - $S \rightarrow S + a \mid S + b \mid c$
 - $S \rightarrow S + a \mid S + b \mid \varepsilon$
 - $S \rightarrow a b c \mid a c$

- e. $S \rightarrow a b c | a b b$
- f. $S \rightarrow a b c | a b$
- g. $S \rightarrow a a | a b | a c$
- h. $S \rightarrow a a | a b | a$
- i. $S \rightarrow a a | a b | \epsilon$
- j. $S \rightarrow a S c | a S b | b$
- k. $S \rightarrow a S c | a S b | a$
- l. $S \rightarrow a S c | a S | a$

7. Parsing

For the problem, assume the term “predictive parser” refers to a top-down, recursive descent, non-backtracking predictive parser.

- a. Consider the following grammar: $S \rightarrow S$ and $S | S$ or $S | (S) | \text{true} | \text{false}$
 - i. Compute First sets for each production and nonterminal
 - ii. Explain why the grammar cannot be parsed by a predictive parser
- b. Consider the following grammar: $S \rightarrow abS | acS | c$
 - i. Compute First sets for each production and nonterminal
 - ii. Show why the grammar cannot be parsed by a predictive parser.
 - iii. Rewrite the grammar so it can be parsed by a predictive parser.
 - iv. Write a predictive parser for the rewritten grammar.
- c. Consider the following grammar: $S \rightarrow Sa | Sc | c$
 - i. Show why the grammar cannot be parsed by a predictive parser.
 - ii. Rewrite the grammar so it can be parsed by a predictive parser.
 - iii. Write a recursive descent parser for your new grammar
- d. Describe an abstract syntax tree (AST)

8. Automata

- a. Describe regular grammars.
- b. Describe the relationship between regular grammars and regular expressions.
- c. Name features needed by automata to recognize
 - i. Regular languages (i.e., languages recognized by regular grammars)
 - ii. Context-free languages
 - iii. All binary numbers
 - iv. All binary numbers divisible by 2
 - v. All prime binary numbers
- d. Compare finite automata, pushdown automata, and Turing machines
- e. Describe computability
- f. Describe a Turing test