

CMSC 330, Spring 2009, Quiz 3 Practice Problem Solutions

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.

No, since the return value for recursive call to insert cannot be used as the return value of the original call to insert. The return value is used to create a Node data type first, and the Node value is returned.

b. Implement min as a tail-recursive function. Raise an exception for an empty bst. Any reasonable exception is fine.

```
let rec min = function
  Empty -> (raise (Failure "min"))
  | Node (m, left, right) ->
    if (is_empty left) then m
    else min left
```

- c. Implement remove. The result should still be a binary search tree.

```
let rec remove n = function  
  Empty -> Empty  
  | Node (m, left, right) ->  
    if m = n then (  
      if (is_empty left) then right  
      else if (is_empty right) then left  
      else let x = min right in  
        Node(x, left, remove x right)  
      // OR  
      // else let x = max left in  
      //   Node(x, remove x left, right)  
    )  
    else if n < m then Node(m, (remove n left), right)  
    else Node(m, left, (remove n right))
```

- 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.

```
let rec fold f a n = match n with  
  Empty -> a  
  | Node (m, left, right) -> fold f (f (fold f a left) m) right
```

- e. Implement size using fold.

```
let size t = fold (fun a m -> a+1) 0 t
```

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: ? → ? *)
```

- a. What rule should have been applied to the productions for F?

Eliminate left recursion

(e.g., change $A \rightarrow A B \mid C$ to $A \rightarrow C N$
 $N \rightarrow B N \mid \text{epsilon}$)

- b. What productions for F & B would be created by applying the rule?

$$F \rightarrow U B$$

$$B \rightarrow ! B \mid \text{epsilon}$$

- c. What sentential form should appear in place of ? in comment 1?

$$! B$$

- d. What production should appear in place of ? in comment 2?

$$B \rightarrow ! B$$

- e. What production should appear in place of ? in comment 3?

$$B \rightarrow \text{epsilon}$$

- f. What production should appear in place of ? in comment 4?

$$F \rightarrow U B$$

3. Context Free Grammars
 - a. List the 4 components of a context free grammar.
Terminals, non-terminals, productions, start symbol
 - b. Describe the relationship between terminals, non-terminals, and productions.
Productions are rules for replacing a single non-terminal with a string of terminals and non-terminals
 - c. Define ambiguity.
Multiple left-most (or right-most) derivations for the same string
 - d. Describe the difference between scanning & parsing.
Scanning matches input to regular expressions to produce terminals, parsing matches terminals to grammars to create parse trees

4. Describing Grammars
 - a. Describe the language accepted by the following grammar:
 $S \rightarrow abS \mid a$
 $(ab)^*a$
 - b. Describe the language accepted by the following grammar:
 $S \rightarrow aSb \mid \epsilon$
 $a^n b^n, n \geq 0$
 - c. Describe the language accepted by the following grammar:
 $S \rightarrow bSb \mid A \quad A \rightarrow aA \mid \epsilon$
 $b^n a^* b^n, n \geq 0$
 - d. Describe the language accepted by the following grammar:
 $S \rightarrow AS \mid B \quad A \rightarrow aAc \mid Aa \mid \epsilon \quad B \rightarrow bBb \mid \epsilon$
Strings of a & c with same or fewer c's than a's and no prefix has more c's than a's, followed by an even number of b's
 - e. Describe the language accepted by the following grammar:
 $S \rightarrow S \text{ and } S \mid S \text{ or } S \mid (S) \mid \text{true} \mid \text{false}$
Boolean expressions of true & false separated by and & or, with some expressions enclosed in parentheses
 - f. Which of the previous grammars are left recursive?
2d, 2e
 - g. Which of the previous grammars are right recursive?
2a, 2c, 2d, 2e
 - h. Which of the previous grammars are ambiguous? Provide proof.
Examples of multiple left-most derivations for the same string
2d: $S \Rightarrow AS \Rightarrow AaS \Rightarrow aS \Rightarrow aB \Rightarrow a$
 $S \Rightarrow AS \Rightarrow S \Rightarrow AS \Rightarrow AaS \Rightarrow aS \Rightarrow aB \Rightarrow a$
2e: $S \Rightarrow S \text{ and } S \Rightarrow S \text{ and } S \text{ and } S \Rightarrow \text{true and } S \text{ and } S$
 $\Rightarrow \text{true and true and } S \Rightarrow \text{true and true and true}$
 $S \Rightarrow S \text{ and } S \Rightarrow \text{true and } S \Rightarrow \text{true and } S \text{ and } S$
 $\Rightarrow \text{true and true and } S \Rightarrow \text{true and true and true}$

5. Creating Grammars

- a. Write a grammar for $a^x b^y$, where $x = y$
 $S \rightarrow aSb \mid \epsilon$
- b. Write a grammar for $a^x b^y$, where $x > y$
 $S \rightarrow aL \quad L \rightarrow aL \mid aLb \mid \epsilon$
- c. Write a grammar for $a^x b^y$, where $x = 2y$
 $S \rightarrow aaSb \mid \epsilon$
- d. Write a grammar for $a^x b^y a^z$, where $z = x+y$
 $S \rightarrow aSa \mid L \quad L \rightarrow bLa \mid \epsilon$
- e. Write a grammar for $a^x b^y a^z$, where $z = x-y$
 $S \rightarrow aSa \mid L \quad L \rightarrow aLb \mid \epsilon$
- f. Write a grammar for all strings of a and b that are palindromes.
 $S \rightarrow aSa \mid bSb \mid L \quad L \rightarrow a \mid b \mid \epsilon$
- g. Write a grammar for all strings of a and b that include the substring baa .
 $S \rightarrow LbaaL \quad L \rightarrow aL \mid bL \mid \epsilon \quad // L = \text{any}$
- h. Write a grammar for all strings of a and b with an odd number of a 's and b 's.
 $S \rightarrow EaE bE \mid EbE aE \quad E \rightarrow EaE aE \mid EbE bE \mid \epsilon \mid SS \quad // E = \text{even \#s}$
- i. Write a grammar for the “if” statement in OCaml
 $S \rightarrow \text{if } E \text{ then } E \text{ else } E \mid \text{if } E \text{ then } E \quad E \rightarrow S \mid \text{expr}$
- j. Write a grammar for all lists in OCaml
 $S \rightarrow [] \mid [E] \mid E::S \quad E \rightarrow \text{elem} \mid S \quad // \text{Ignores types, allows lists of lists}$
- k. Which of your grammars are ambiguous? Can you come up with an unambiguous grammar that accepts the same language?

Grammar for 3h is ambiguous. An unambiguous grammar must exist since the language can be recognized by a deterministic finite automaton, and DFA \rightarrow RE \rightarrow Regular Grammar.

Grammar for 3i is ambiguous. Multiple derivations for “if expr then if expr then expr else expr”. It is possible to write an unambiguous grammar by restricting some S so that no unbalanced if statement can be produced.

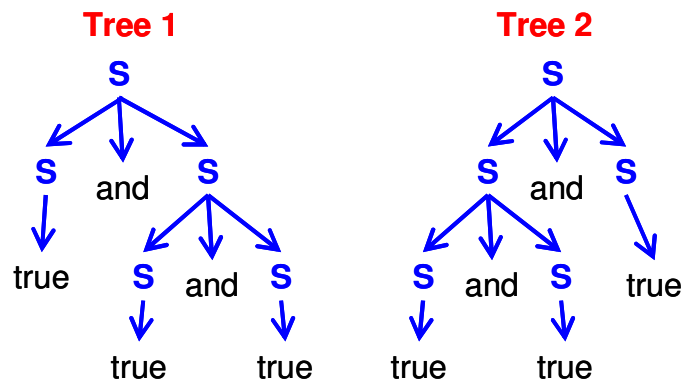
6. Derivations, Parse Trees, Precedence and Associativity

For the following grammar: $S \rightarrow S \text{ and } S \mid \text{true}$

- a. List 4 derivations for the string “true and true and true”.
 - i. $S \Rightarrow \underline{S} \text{ and } S \Rightarrow \underline{S} \text{ and } S \text{ and } S \Rightarrow \text{true and } \underline{S} \text{ and } S \Rightarrow \text{true and true and } S \Rightarrow \text{true and true and true}$
 - ii. $S \Rightarrow \underline{S} \text{ and } S \Rightarrow \text{true and } S \Rightarrow \text{true and } \underline{S} \text{ and } S \Rightarrow \text{true and true and } S \Rightarrow \text{true and true and true}$
 - iii. $S \Rightarrow S \text{ and } \underline{S} \Rightarrow S \text{ and true} \Rightarrow S \text{ and } \underline{S} \text{ and true} \Rightarrow S \text{ and true and true} \Rightarrow \text{true and true and true}$
 - iv. $S \Rightarrow S \text{ and } \underline{S} \Rightarrow S \text{ and } S \text{ and } \underline{S} \Rightarrow S \text{ and } \underline{S} \text{ and true} \Rightarrow S \text{ and true and true} \Rightarrow \text{true and true and true}$
 - v. $S \Rightarrow \underline{S} \text{ and } S \Rightarrow \underline{S} \text{ and } S \text{ and } S \Rightarrow \text{true and } S \text{ and } \underline{S} \Rightarrow \text{true and } S \text{ and true} \Rightarrow \text{true and true and true}$
 - vi. $S \Rightarrow \underline{S} \text{ and } S \Rightarrow S \text{ and } \underline{S} \text{ and } S \Rightarrow \underline{S} \text{ and true and } S \Rightarrow \text{true and true and } S \Rightarrow \text{true and true and true}$

- vii. $S \Rightarrow \underline{S}$ and $S \Rightarrow S$ and \underline{S} and $S \Rightarrow S$ and true and $\underline{S} \Rightarrow S$ and true and true \Rightarrow true and true and true
- viii. $S \Rightarrow \underline{S}$ and $S \Rightarrow S$ and S and $\underline{S} \Rightarrow \underline{S}$ and S and true \Rightarrow true and S and true \Rightarrow true and true and true
- ix. $S \Rightarrow \underline{S}$ and $S \Rightarrow S$ and S and $\underline{S} \Rightarrow S$ and \underline{S} and true $\Rightarrow S$ and true and true \Rightarrow true and true and true
- x. $S \Rightarrow \underline{S}$ and $S \Rightarrow$ true and $S \Rightarrow$ true and S and $\underline{S} \Rightarrow$ true and S and true \Rightarrow true and true and true
- xi. $S \Rightarrow S$ and $\underline{S} \Rightarrow S$ and true $\Rightarrow \underline{S}$ and S and true \Rightarrow true and S and true \Rightarrow true and true and true
- xii. $S \Rightarrow S$ and $\underline{S} \Rightarrow \underline{S}$ and S and $S \Rightarrow$ true and \underline{S} and $S \Rightarrow$ true and true and $S \Rightarrow$ true and true and true
- xiii. $S \Rightarrow S$ and $\underline{S} \Rightarrow \underline{S}$ and S and $S \Rightarrow$ true and S and $\underline{S} \Rightarrow$ true and S and true \Rightarrow true and true and true
- xiv. $S \Rightarrow S$ and $\underline{S} \Rightarrow S$ and \underline{S} and $S \Rightarrow \underline{S}$ and true and $S \Rightarrow$ true and true and $S \Rightarrow$ true and true and true
- xv. $S \Rightarrow S$ and $\underline{S} \Rightarrow S$ and \underline{S} and $S \Rightarrow S$ and true and $\underline{S} \Rightarrow S$ and true and true \Rightarrow true and true and true
- xvi. $S \Rightarrow S$ and $\underline{S} \Rightarrow S$ and S and $\underline{S} \Rightarrow \underline{S}$ and S and true \Rightarrow true and S and true \Rightarrow true and true and true

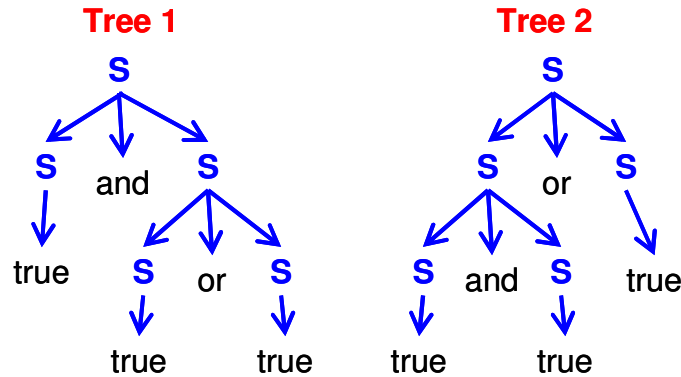
- b. Label each derivation as left-most, right-most, or neither.
 - i and ii are left-most derivations, iii and iv are right-most derivations, remaining derivations are neither**
- c. List the parse tree for each derivation
 - Tree 1 = ii, iii, x, xi, Tree 2 = rest**



- d. What is implied about the associativity of “and” for each parse tree?
 - Tree 1 => and is right-associative, Tree 2 => and is left-associative**

For the following grammar: $S \rightarrow S \text{ and } S \mid S \text{ or } S \mid \text{true}$

- 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?

Tree 1 => or has higher precedence than and

Tree 2 => and has higher precedence than or

g. Rewrite the grammar so that “and” has higher precedence than “or” and is right associative

$S \rightarrow S \text{ or } S \mid L$

// op closer to Start = lower precedence op

$L \rightarrow \text{true and } L \mid \text{true}$

// right recursive = right associative

7. Left factoring & eliminating left recursion

Rewrite the following grammars so they can be parsed by a predictive parser by eliminating left recursion and applying left factoring where necessary

a. $S \rightarrow S + a \mid b$

↓

$S \rightarrow b L$

$L \rightarrow + a L \mid \epsilon$

b. $S \rightarrow S + a \mid S + b \mid c$

↓

$S \rightarrow c L$

$L \rightarrow + a L \mid + b L \mid \epsilon$

↓

$S \rightarrow c L$

$L \rightarrow + M \mid \epsilon$

$M \rightarrow a L \mid b L$

c. $S \rightarrow S + a \mid S + b \mid \epsilon$

↓

$S \rightarrow L$

$L \rightarrow + a L \mid + b L \mid \epsilon$

↓

$S \rightarrow L$

$L \rightarrow + M \mid \epsilon$

$M \rightarrow a L \mid b L$

d. $S \rightarrow a b c \mid a c$

↓

$S \rightarrow a L$

$L \rightarrow b c \mid c$

e. $S \rightarrow a b c | a b b$

\downarrow
 $S \rightarrow a b L$
 $L \rightarrow c | b$

f. $S \rightarrow a b c | a b$

\downarrow
 $S \rightarrow a b L$
 $L \rightarrow c | \epsilon$

g. $S \rightarrow a a | a b | a c$

\downarrow
 $S \rightarrow a L$
 $L \rightarrow a | b | c$

h. $S \rightarrow a a | a b | a$

\downarrow
 $S \rightarrow a L$
 $L \rightarrow a | b | \epsilon$

i. $S \rightarrow a a | a b | \epsilon$

\downarrow
 $S \rightarrow a L | \epsilon$
 $L \rightarrow a | b$

j. $S \rightarrow a S c | a S b | b$

\downarrow
 $S \rightarrow a S L | b$
 $L \rightarrow c | b$

k. $S \rightarrow a S c | a S b | a$

\downarrow
 $S \rightarrow a L$
 $L \rightarrow S c | S b | \epsilon$

\downarrow
 $S \rightarrow a L$
 $L \rightarrow S M | \epsilon$
 $M \rightarrow c | b$

l. $S \rightarrow a S c | a S | a$

\downarrow
 $S \rightarrow a L$
 $L \rightarrow S c | S | \epsilon$

\downarrow
 $S \rightarrow a L$
 $L \rightarrow S M | \epsilon$
 $M \rightarrow c | \epsilon$

8. 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 $S | (S) | true | false$
 - i. Compute First sets for each production and nonterminal

First(true) = { “true” }

First(false) = { “false” }

First(S) = { “(“ }

First(S and S) = First(S or S) = First(S) = { “(“, “true”, “false” }

- ii. Explain why the grammar cannot be parsed by a predictive parser

First sets of productions intersect, grammar is left recursive

- b. Consider the following grammar: $S \rightarrow abS \mid acS \mid c$

- i. Compute First sets for each production and nonterminal

First(abS) = { a }

First(acS) = { a }

First(c) = { c }

First(S) = { a, c }

- ii. Show why the grammar cannot be parsed by a predictive parser.

First sets of productions overlap

First(abS) \cap First(acS) = { a } \cap { a } = { a } \neq \emptyset

- iii. Rewrite the grammar so it can be parsed by a predictive parser.

$S \rightarrow aL \mid c$ $L \rightarrow bS \mid cS$

- iv. Write a predictive parser for the rewritten grammar.

```
parse_S() {
    if (lookahead == “a”) {
        match(“a”); // S → aL
        parse_L();
    }
    else if (lookahead == “c”)
        match(“c”); // S → c
    else error();
}

parse_L() {
    if (lookahead == “b”) {
        match(“b”); // L → bS
        parse_S();
    }
    else if (lookahead == “c”) {
        match(“c”); // L → cS
        parse_S();
    }
    else error();
}
```

- c. Consider the following grammar: $S \rightarrow Sa \mid Sc \mid c$

- i. Show why the grammar cannot be parsed by a predictive parser.

First sets of productions intersect, grammar is left recursive

- ii. Rewrite the grammar so it can be parsed by a predictive parser.

$S \rightarrow cL$ $L \rightarrow aL \mid cL \mid \epsilon$

- iii. Write a recursive descent parser for your new grammar

```
parse_S() {
```

```

    if (lookahead == "c") {
        match("c"); // S → cL
        parse_L();
    }
    else error();
}
parse_L() {
    if (lookahead == "a") {
        match("a"); // L → aL
        parse_L();
    }
    else if (lookahead == "c") {
        match("c"); // L → cL
        parse_L();
    }
    else ; // L → ε
}

```

- d. Describe an abstract syntax tree (AST)

Compact representations of parse trees with only essential parts

9. Automata

- a. Describe regular grammars.

Grammars where all productions are of the form $X \rightarrow a$ or $X \rightarrow aY$

- b. Describe the relationship between regular grammars and regular expressions.

Regular grammars are exactly as powerful as regular expressions (and one can be converted to the other)

- c. Name features needed by automata to recognize

- i. Regular languages (i.e., languages recognized by regular grammars)

DFA (automaton with finite # of states and transitions)

- ii. Context-free languages

NFA and 1 stack

- iii. All binary numbers

DFA (binary #s can be recognized by RE)

- iv. All binary numbers divisible by 2

DFA (binary #s ending in 0 can be recognized by RE)

- v. All prime binary numbers

DFA and 1 tape (can write a program to compute prime #s)

- d. Compare finite automata, pushdown automata, and Turing machines

Pushdown automata are finite automata that can use 1 stack, Turing machines are finite automata that can use a tape (or 2 stacks). Turing machines > pushdown automata > finite automata in terms of computing power.

- e. Describe computability

Problem that can be solved by algorithm of finite length

- f. Describe a Turing test

When communicating by text, indistinguishable from human being