CMSC 330, Spring 2008, Midterm 2 Practice Problem Solutions

1. 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

2. 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:
      $S \rightarrow aSb \mid \varepsilon$
      $a^n b^n$, $n \geq 0$
   c. Describe the language accepted by the following grammar:
      $S \rightarrow bSb \mid A$ $\quad$ $A \rightarrow aA \mid \varepsilon$
      $b^n a^* b^n$, $n \geq 0$
   d. Describe the language accepted by the following grammar:
      $S \rightarrow AS \mid B$ $\quad$ $A \rightarrow aAc \mid Aa \mid \varepsilon$ $\quad$ $B \rightarrow bBb \mid \varepsilon$
      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 → S and S | S or S | (S) | true | 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 => AS => AaS => aS => aB => a
      S => AS => S => AS => AaS => aS => aB => a
      2e: S => S and S => S and S and S => true and S and S
      => true and true and S => true and true and true
      S => S and S => true and S => true and S and S
      => true and true and S => true and true and true

3. Creating Grammars
   a. Write a grammar for $a^x b^y$, where x = y
      S → aSb | ε
   b. Write a grammar for $a^x b^y$, where x > y
      S → aL **L → aL | aLb | ε**
   c. Write a grammar for $a^x b^y$, where x = 2y
      S → aaSb | ε
   d. Write a grammar for $a^x b^y a^z$, where z = x+y
      S → aSa | L **L → bLa | ε**
   e. Write a grammar for $a^x b^y a^z$, where z = x-y
      S → aSa | L **L → aLb | ε**
   f. Write a grammar for all strings of *a* and *b* that are palindromes.
      S → aSa | bSb | L **L → a | b | ε**
   g. Write a grammar for all strings of *a* and *b* that include the substring *baa*.
      S → LbaaL **L → aL | bL | ε** **// L = any**
   h. Write a grammar for all strings of *a* and *b* with an odd number of *a*'s and *b*'s.
      S → EaEbE | EbEaE **E → EaEaE | EbEbE | ε** **// E = even #s**
   i. Write a grammar for the "if" statement in OCaml
      S → if S then S else S | if S then S | expr
   j. Write a grammar for all lists in OCaml
      S → [] | [E] | E::S **E → elem | S** **// 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 -> RE -> Regular Grammar.
      Grammar for 3i is ambiguous. Multiple derivations for "if S then if S then S". It is possible to write an unambiguous grammar by restricting some S so that no unbalanced if statement can be produced.

4. Derivations, Parse Trees, Precedence and Associativity
   For the following grammar: S → S and S | true
   a. List all derivations for the string "true and true and true".
      i. **S => S̲ and S => S̲ and S and S => true and S̲ and S => true and true and S => true and true and true**
      ii. **S => S̲ and S => true and S => true and S̲ and S => true and true and S => true and true and true**
      iii. **S => S and S̲ => S and true => S and S̲ and true => S and true and true => true and true and true**
      iv. **S => S and S̲ => S and S and S̲ => S and S̲ and true => S and true and true => true and true and true**
      v. **S => S̲ and S => S̲ and S and S => true and S and S̲ => true and S and true => true and true and true**
      vi. **S => S̲ and S => S and S̲ and S => S̲ and true and S => true and true and S => true and true and true**

vii. **S => S and S => S and S and S => S and true and S => S and true and true => true and true and true**

viii. **S => S and S => S and S and S => S and S and true => true and S and true => true and true and true**

ix. **S => S and S => S and S and S => S and S and true => S and true and true => true and true and true**

x. **S => S and S => true and S => true and S and S => true and S and true => true and true and true**

xi. **S => S and S => S and true => S and S and true => true and S and true => true and true and true**

xii. **S => S and S => S and S and S => true and S and S => true and true and S => true and true and true**

xiii. **S => S and S => S and S and S => true and S and S => true and S and true => true and true and true**

xiv. **S => S and S => S and S and S =>S and true and S => true and true and S => true and true and true**

xv. **S => S and S => S and S and S => S and true and S => S and true and true => true and true and true**

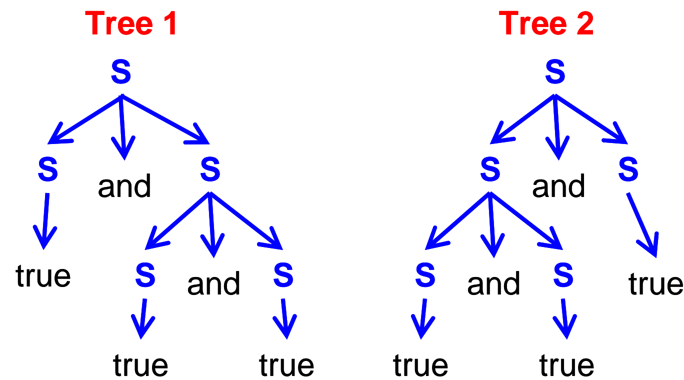
xvi. **S => S and S => S and S and S => S and S and true => true and S and true => true and true and true**

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

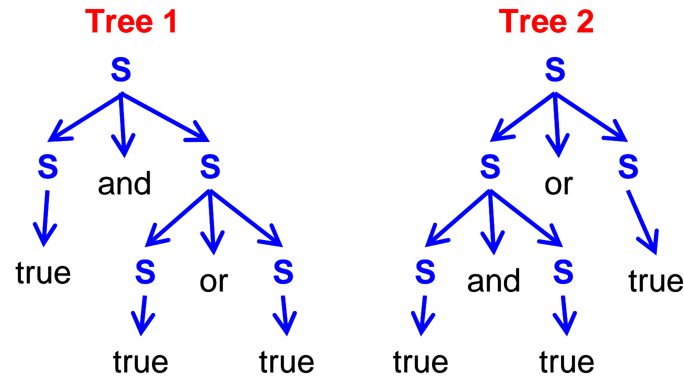


d. What is implied about the associativity of “and” for each parse tree?

Tree 1 => and is right-associative, Tree 2 => and is left-associative

For the following grammar: S → 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?

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 → S or S | L // **op closer to Start = lower precedence op**
L → true and L | true // **right recursive = right associative**

5. 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 → 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 → abS | acS | 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) ∩ First(acS) = { a } ∩ { a } = { a } ≠ ∅

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

S → aL | c **L → bS | 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 → Sa | Sc | 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 → c L   L → aL | cL | ε
    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

6. Automata
    a. Describe regular grammars.

Grammars where all productions are of the form X → a or X → 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

7. OCaml and Functional Programming
   a. Define functional programming
      Programs are expression evaluations
   b. Define imperative programming
      Programs change the value of variables
   c. Define iterative programming.
      Programs that use loop constructs (e.g., while, for)
   d. Define higher-order functions
      Functions can be passed as arguments and returned as results
   e. Describe the relationship between type inference and static types
      Variable has a fixed type that can be inferred by looking at how variable is used in the code
   f. Describe the properties of OCaml lists
      Entity containing 0 or more elements of the same type. Type of list is determined by type of element.
   g. Describe the properties of OCaml tuples
      Entity containing 2 or more elements of possibly different types. Type of tuple is determined by type and number of elements.
   h. Define pattern variables in OCaml
      Variables making up patterns used by "match"
   i. Describe the usage of "_" in OCaml

Pattern variable that can match anything but does not add binding

j. Describe polymorphism
   Function that can take different types for same formal parameter
k. Write a polymorphic OCaml function
   let f x = x // ‘a -> ‘a, x can be of any type
l. Describe variable binding
   A variable (symbol) is associated with a value in an expression (or environment)
m. Describe scope
   Portion of program where variable binding is visible
n. Describe lexical scoping
   Variable binding determined by nearest scope in text of program
o. Describe dynamic scoping
   Variable binding determined by nearest runtime function invocation
p. Describe environment
   Collection of variable bindings
q. Describe closure
   Function code + environment pair, may be invoked as function
r. Describe currying
   Functions consume one argument at a time, returning closures until all arguments are consumed

8. OCaml Types & Type Inference

   Give the type of the following OCaml expressions:

   a. [] **// ‘a list**
   b. 1::[] **// int list**
   c. 1::2::[] **// int list**
   d. [1;2;3] **// int list**
   e. [[1];[1]] **// int list list**
   f. (1) **// int**
   g. (1,”bar”) **// int * string**
   h. ([1,2], [“foo”,”bar”]) **// (int * int) list * (string * string) list**
   i. [(1,2,”foo”);(3,4,”bar”)] **// (int * int * string) list**
   j. let f x = 1 **// ‘a -> int**
   k. let f (x) = x *. 3.14 **// float -> float**
   l. let f (x,y) = x **// ‘a * ‘b -> ‘a**
   m. let f (x,y) = x+y **// int * int -> int**
   n. let f (x,y) = (x,y) **// ‘a * ‘b -> ‘a * ‘b**
   o. let f (x,y) = [x,y] **// ‘a * ‘b -> (‘a * ‘b) list**
   p. let f x y = 1 **// ‘a -> ‘b -> int**
   q. let f x y = x*y **// int -> int -> int**
   r. let f x y = x::y **// ‘a -> ‘a list -> ‘a list**
   s. let f x = match x with [] -> 1 **// ‘a list -> int**
   t. let f x = match x with (y,z) -> y+z **// int * int -> int**
   u. let f (x::_) -> x **// ‘a list -> ‘a**
   v. let f (_::y) = y **// ‘a list -> ‘a list**

```
w.  let f (x::y::_) = x+y                          // int list -> int
x.  let f = fun x -> x + 1                         // int -> int
y.  let rec x = fun y -> x y                       // ‘a -> ‘b
z.  let rec f x = if (x = 0) then 1 else 1+f (x-1) // int -> int
aa. let f x y z = x+y+z in f 1 2 3                 // int
bb. let f x y z = x+y+z in f 1 2                   // int -> int
cc. let f x y z = x+y+z in f                       // int -> int -> int -> int
dd. let rec f x = match x with                     // ‘a list -> int
        [] -> 0
        | (_::t)  -> 1 + f t
ee. let rec f x = match x with                     // int list -> int
        [] -> 0
        | (h::t)  -> h + f t
ff. let rec f = function                           // int list -> int
        [] -> 0
        | (h::t)  -> h + (2*(f t))
gg. let rec func (f, l1, l2) = match l1 with   // (‘a -> ‘b) * ‘a list * ‘a list -> ‘b list
        [] -> []
        | (h1::t1) -> match l2 with
            [] -> [f h1]
            |(h2::t2) -> [f h1; f h2]
```

9. OCaml Types & Type Inference

Write an OCaml expression with the following types:

```
a.  int list                                  // [1]
b.  int * int                                 // (1,1)
c.  int -> int                                // let f x = x+1
d.  int * int -> int                          // let f (x,y) = x+y
e.  int -> int -> int                         // let f x y = x+y
f.  int -> int list -> int list               // let f x y = (x+1)::y
g.  int list list -> int list                 // let f (x::_) = 1::x
h.   ‘a -> ‘a                                 // let f x = x
i.  ‘a * ‘b -> ‘a                             // let f (x,y) = x
j.  ‘a -> ‘b -> ‘a                            // let f x y = x
k.  ‘a -> ‘b -> ‘b                            // let f x y = y
l.  ‘a list * ‘b list -> (‘a * ‘b) list       // let f (x::_,y::_) = [(x,y)]
m.  int -> (int -> int)                       // let f x y = x+y
n.  (int -> int) -> int                       // let f x = 1+(x 1)
o.  (int -> int) -> (int -> int) -> int       // let f x y = 1+(x 1)+(y 1)
p.  (’a -> ’b) * (’c * ’c -> ’a) * ’c -> ’b   // let f (x, y, z) = (x (y (z,z)))
```

10. OCaml Programs

What is the value of the following OCaml expressions? If an error exists, describe the error.

```
a.  2 ; 3              // 3
b.  2 ; 3 + 4          // 7
```

c. (2 ; 3) + 4 **// 7**

d. if 1<2 then 3 else 4 **// 3**

e. let x = 1 in 2 **// 2**

f. let x = 1 in x+1 **// 2**

g. let x = 1 in x ; x+1 **// 2**

h. let x = (1, 2) in x ; x+1
// error: x has type int*int but used with int

i. (let x = (1, 2) in x) ; x+1 **// error: unbound value x**

j. let x = 1 in let y = x in y **// 1**

k. let x = 1 let y = 2 in x+y **// syntax error: missing “in”**

l. let x = 1 in let x = x+1 in let x = x+1 in x **// 3**

m. let x = x in let x = x+1 in let x = x+1 in x **// error: unbound value x**

n. let rec x y = x in 1 **// error: x has type ‘a -> ‘b but used with ‘b**

o. let rec x y = y in 1 **// 1**

p. let rec x y = y in x 1 **// 1**

q. let x y = fun z -> z+1 in x **// fun y -> (fun z -> z+1)**

r. let x y = fun z -> z+1 in x 1 **// fun z -> z+1**

s. let x y = fun z -> z+1 in x 1 1 **// 2**

t. let x y = fun z -> x+1 in x 1 **// error: unbound value x**

u. let rec x y = fun z -> x+1 in x 1
// error: x has type ‘a -> ‘b -> ‘c but used with int

v. let rec x y = fun z -> x+y in x 1
// error: x has type ‘a -> ‘b -> ‘c but used with int

w. let rec x y = fun z -> x y in x 1
// error: x has type ‘a -> ‘b but used with ‘b

x. let rec x y = fun z -> x z in x 1
// error: x has type ‘a -> ‘b but used with ‘b

y. let x y = y 1 in 1 **// 1**

z. let x y = y 1 in x **// fun y -> (y 1)**

aa. let x y = y 1 in x 1 **// error: 1 has type int but used with int -> ‘a**

bb. let x y = y 1 in x fun z -> z + 1 **// syntax error at “x fun”**

cc. let x y = y 1 in x (fun z -> z + 1) **// 2**

dd. let a = 1 in let f x y z = x+y+z+a in f 1 2 3 **// 7**

ee. let a = 1 in let f x y z = x+y+z+a in f 1 2 -3
// error: (f 1 2) has type int -> int but used with int

11. OCaml Programming

a. Write an OCaml function named *fib* that takes an int *x*, and returns the Fibonacci number for *x*. Recall that fib(0) = 0, fib(1) = 1, fib(2) = 1, fib(3) = 2.

```
let rec fib x =
        if (x = 0) then 0
        else if (x = 1) then 1
        else (fib (x-1) + fib (x-2))
;;
```

b. Write an OCaml function named *concat* which takes 2 lists and returns the concatenated list.

```
let rec concat x y = match x with
        [] -> y
        | (h::t) -> h::(concat t y)
;;
```

c. Write an OCaml function named *map_odd* which takes a function *f* and a list *lst*, applies the function to every other element of the list, starting with the first element, and returns the result in a new list. Use map_odd and fib applied to the list [1;2;3;4;5;6;7] to calculate the Fibonacci numbers for 1, 3, 5, and 7.

```
let rec map_odd f l = match l with
        [] -> []
        | (x1::[]) -> [f x1]
        | (x1::x2::t) -> (f x1)::(map_odd f t)
;;
map_odd fib [1;2;3;4;5;6;7]
;;
```

d. Given the *fold* function, write an OCaml function named *all_true* which may be applied to a list of booleans *lst* so that it returns true only if all elements of *lst* are true.

```
let rec fold f a l = match l with
        [] -> a
        | (h::t) -> fold f (f a h) t
;;
let all_true lst = fold (fun a x -> (x = true) && (a = true)) true lst
;;
(* all_true [true;true;true] = true *)
(* all_true [true;false;true] = false *)
```

e. Write an OCaml function named *paths_blocked f m n b* that computes the number of paths from (*m,n*) to (1,1) that pass through exactly *b* blocked intersections. So *paths_blocked f m n 0* should yield the same result as *paths.* The number of blocked intersections (on the path) increases by 1 every time the path passes through a blocked intersection. Thus leaving (m,n) or arriving at (1,1) will not count as passing through a blocked intersection even if (m,n) and (1,1) are blocked.

(* can't provide answer until project deadline is past ☺ *)

f. Write an OCaml function named nth which has a tuple of an int named n and a list as a parameter and which returns the n'th element of the list. The list's first element is considered to be element number 1. For example:

nth (1, [2; 4; 6; 8; 10]) would return 2
nth (2, [2; 4; 6; 8; 10]) would return 4
nth (3, ["hi"; "ciao"; "bye"]) would return "bye"

You can assume the list will always have an n'th element to be returned, and you can also assume that n > 0. It doesn't matter if your function would generate any incomplete match warnings.

```
let rec nth (n,lst) = match lst with
        | (h::t) -> if (n=1) then h else nth (n-1,t)
;;
```