

CMSC330 Spring 2008 Midterm #2 Solutions

1. (14 pts) Context Free Grammars & Automata
 - a. (2 pts) Explain how context free grammars are used for programming languages.
CGFs are used to precisely specify syntax of programming languages
 - b. (2 pts) Describe the relationship between derivations and sentential forms.
Sentential forms are the strings produced by derivations
 - c. (2 pts) Describe the language accepted by the grammar: $S \rightarrow aaSb \mid aSb \mid \epsilon$
 $a^x b^y$, where $2y \geq x \geq y$, and $x, y \geq 0$
 - d. (4 pts) Write a grammar for $a^x b^y a^z$, where $z = 2x - y$, for $x, y, z \geq 0$
 $S \rightarrow aSaa \mid aLba \mid L \quad L \rightarrow aLbb \mid \epsilon$
 - e. (2 pts) Name features needed by automata to recognize all binary numbers with more 1's than 0's.
DFA and stack (since language recognizable with simple CFG)
 - f. (2 pts) Explain why a finite automaton with 2 stacks can recognize many more languages than a finite automaton with 1 stack.
2 stacks can simulate a tape, yielding a Turing machine

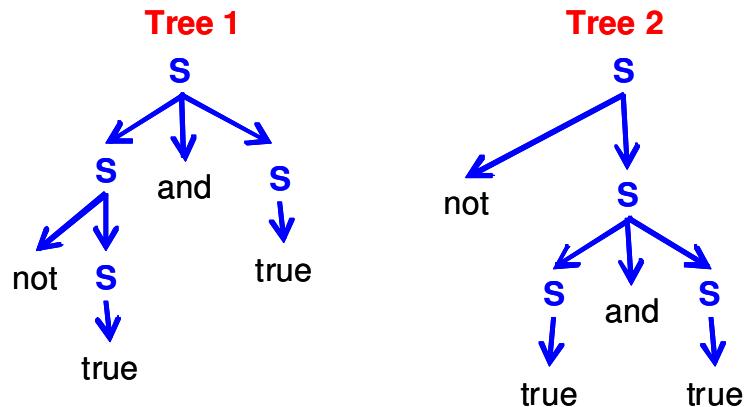
2. (14 pts) Derivations, Parse Trees, Precedence and Associativity

For the following grammar: $S \rightarrow S \text{ and } S \mid \text{not } S \mid \text{true} \mid \text{false}$

- a. (4 pts) List all left-most derivations for the string "not true and true"

D1: $S \Rightarrow S \text{ and } S \Rightarrow \text{not } S \text{ and } S \Rightarrow \text{not true and } S \Rightarrow \text{not true and true}$

D2: $S \Rightarrow \text{not } S \Rightarrow \text{not } S \text{ and } S \Rightarrow \text{not true and } S \Rightarrow \text{not true and true}$
- b. (2 pts) Draw the parse tree for one of the left-most derivations above.



where D1 \rightarrow Tree 1, D2 \rightarrow Tree 2

- c. (6 pts) Rewrite the grammar so that "and" is left associative and has lower precedence than "not".
 $S \rightarrow S \text{ and } L \mid L \quad L \rightarrow \text{not } L \mid \text{true} \mid \text{false}$
- d. (2 pts) Is your rewritten grammar ambiguous?
No

3. (16 pts) Parsing

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

a. (10 pts) Consider the following grammar: $S \rightarrow Ac \mid b$ $A \rightarrow aS \mid \epsilon$

i. (4 pts) Compute First sets for each production and nonterminal

First(aS) = { a }

First(ϵ) = { ϵ }

First(A) = First(aS) \cup First(ϵ) = { a, ϵ }

First(Ac) = { First(A) - ϵ } \cup First(c) = { a } \cup { c } = { a, c }

First(b) = { b }

First(S) = First(Ac) \cup First(b) = { a, b, c }

ii. (4 pts) Write a predictive parser for the grammar

```

parse_S() {
    if ((lookahead == "a") || // S  $\rightarrow$  Ac
        (lookahead == "c")) {
        parse_A();
        match("c");
    }
    else if (lookahead == "b") { // S  $\rightarrow$  b
        match("b");
    }
    else error();
}

parse_A() {
    If (lookahead == "a") { // A  $\rightarrow$  aS
        match("a");
        parse_S();
    }
    else ; // A  $\rightarrow$   $\epsilon$ 
}

```

iii. (2 pts) Use your parser to parse the string “abc”. Show the sequence of calls in the parse, and what symbols remain at each point.

Parse “abc”	Remaining
parse_S()	“abc”
parse_A()	“abc”
match(“a”)	“abc”
parse_S()	“bc”
match(“b”)	“bc”
match(“c”)	“c”
	“”

- b. (6 pts) Consider the following grammar: $S \rightarrow aSc \mid ab \mid a$
- (2 pts) Show why the grammar cannot be parsed by a predictive parser.
 $\text{First}(aSc) \cap \text{First}(ab) = \{ a \} \cap \{ a \} = \{ a \} \neq \emptyset$
First sets of productions for S overlap \rightarrow grammar not predictive
 - (4 pts) Rewrite the grammar so it can be parsed by a predictive parser, using the rules presented in class for left factoring & eliminating left recursion.
 $S \rightarrow aL \quad L \rightarrow Sc \mid b \mid \epsilon$

4. (8 pts) OCaml and Functional Programming

- (2 pts) Describe one advantage of functional programming
Programs easier to analyze than imperative programs. No aliasing
- (2 pts) Describe the difference between the usage of “;” and “,” in OCaml
Semicolon separates expressions, comma separates elements of a tuple
- (2 pts) Describe the relationship between type inference and polymorphic types
Type inference assigns polymorphic types to variables that have multiple possible types based on how they’re used in the code
- (2 pts) Describe the difference between function pointers and closures
Closures include both a function pointer and an environment

5. (10 pts) OCaml Types & Type Inference 1

Give the type of the following OCaml expressions:

- (2 pts) `[1, "bar"]` // **(int * string) list**
- (2 pts) `let rec f x = match x with
[] -> []
| (h::t) -> (h+1)::(f t)` // **int list -> int list**
- (2 pts) `let f (x::y) = [y; [x]]` // **'a list -> 'a list list**
- (4 pts) `let f x y z = y x` // **'a -> ('a -> 'b) -> 'c -> 'b**

6. (12 pts) OCaml Types & Type Inference 2

Write an OCaml expression with the following types:

- (2 pts) `string list list` // **[["foo"]]**
- (4 pts) `'a * ('b list) -> ('a * 'b) list` // **let f (x,y::_) = [x,y]**
- (6 pts) `(int -> 'a) -> (int -> 'a)` // **let f x y = x (y+1)**

7. (12 pts) OCaml Programs

What are the values of the following OCaml expressions? If an error exists, describe the error.

- (2 pts) `1 + 2 ; 3 + 4` // **7**
- (2 pts) `[1; "foo"]` // **mixed types in list, "foo" has type string but used with int**
- (2 pts) `let x = 1 in let y = x+2 in let x = y+3 in x+4` // **10**
- (3 pts) `let x y = fun z -> z+y in x 1 2` // **3**
- (3 pts) `let x y = fun z -> y z in x (fun x -> x+3) 4` // **7**

8. (26 pts) OCaml Programming

For the following problems, you may use helper functions, but no library functions.

You are given the curried version of the fold function:

```
let rec fold f a l = match l with
  [] -> a
  | (h::t) -> fold f (f a h) t
```

- a. (4 pts) Using the curried version of the *fold* function, write an OCaml function named *reverse* that when applied to a list *lst* returns the list in reverse order.

Example: `reverse [1;3;5;2;4] = [4;2;5;3;1]`

```
let reverse lst = fold (fun a h -> h::a) [] lst
```

- b. (10 pts) Using the curried version of the *fold* function, write an OCaml function named *filter* with type `((a -> bool) -> 'a list -> 'a list)` that takes two arguments: a predicate function *pred* with type `('a -> bool)`, and list *lst* with type `('a list)`. *filter* returns only the elements of *lst* that return true when evaluated by *pred*. The filtered elements must be returned in their order in *lst*. You may use the reverse function above.

Example: `filter (fun x -> (x > 2)) [1;3;5;2;4] = [3;5;4]`

```
let filter pred lst = reverse (fold (fun a h -> if (pred h) then h::a else a) [] lst)
```

- c. (12 pts) Write an OCaml function named *rev_map* which takes a function *f* and a list *lst*, applies *f* to every element *lst*, and returns the results in a new list in reverse order. You must implement *rev_map* as a single pass over the input list (i.e., you cannot first apply map, then reverse the result).

Example: `rev_map (fun x -> x+1) [1;3;5;2;4] = [5;3;6;4;2]`

```
let rev_map f lst = fold (fun a h -> (f h)::a) [] lst
```