CS421 Spring 2009 Midterm 1

Thursday, February 26, 2009

Name:	
NetID:	

- You have **90 minutes** to complete this exam.
- This is a **closed-book** exam.
- Do not share anything with other students. Do not talk to other students. Do not look at another student's exam. Do not expose your exam to easy viewing by other students. Violation of any of these rules will count as cheating.
- If you believe there is an error, or an ambiguous question, seek clarification from one of the TAs. You must use a whisper, or write your question out.
- Including this cover sheet, there are 10 pages to the exam. Please verify that you have all 10 pages.
- Please write your name and NetID in the spaces above, and at the top of every page.

Question	Possible points	Points earned	EC points
1	8		
2	8		
3	6		
4	8		
5	8		
6	8		
7	4		
8	8		
9	8		
10 (EC)	8		
11	10		
12	6		
13	10		
14	8		
15 (EC)	6		
Total	100 + 14		

1. (8 pts) Give the types of the following OCaml functions:

a. let f (a,b,c) = a + c f: __int * 'a * int -> int__________________

b. let g (a,b) = b g: __'a * 'b -> 'b_________________________

c. let rec h a b = h: __'a list list -> 'a list -> bool_______________

```
if a = [b] then true else h a (tl b)
```

d. let rec flatten lst = match lst with flatten: ___'a list list -> 'a list_______________

```
    (x::xs)::ys -> x::(flatten (xs::ys))
  | []::ys      -> flatten ys
  | []          -> [];;
```

2. (8 pts) Consider this function: let sec arg = match arg with (x::y)::z -> y | a::b -> b
What would the output be if sec is applied on the following values? If there is a type mismatch, write "Error;" if there is a run-time error, write "Run-time error."

a) [1;2;3] _type error______

b) [[1;2;3];[4;5;6];[7;8;9]] __[2; 3]___________

c) [[1];[2];[3]] __[]________________

d) [[];[1;2;3]] __[]________________

3. (6 pts) Given the function

```
let rec zero lst = match lst with  [] -> []  | x::xs -> (x>0)::zero xs
```

What is the output of the application zero [8;0;-3;4]? __[true; false; false; true]________

4. (8 pts) Write a function concat: string list -> string -> string concatenates the strings in its first argument, separated by the second argument, e.g. concat ["CS"; "421"; "rocks"] " " = "CS 421 rocks". (concat [] = "".)

```
let rec concat sl s = match sl with
  [] -> "" | [s'] -> s' | s'::sl' -> s' ^ s ^ concat sl' s;;
```

5. (8 pts) Write a function f that returns a pair of the number of elements in a list and the last element. E.g. f [1;2;3;4;3;2] = (6,2). You may not use any library functions (such as length), and you may assume that the argument is a non-empty list.

```
let rec f lis = match lis with
                          [n] -> (1,n)
                        | n::ns -> let (a,b) = f ns
                                   in (a+1,b)
```

6. (8 pts) A string list represents an expression in "prefix notation" if it has one of these forms: either it contains a single number, or it contains the string "+" and two lists in prefix notation all joined together. Here is an example: ["+"; "3"; "+"; "5"; "6"]. That example evaluates to 14.

Write a function evalprefix: string list -> int * string list. This takes a string list and evaluates the prefix expression at the start of it, returning the value of that expression and the remainder of the list. E.g.,

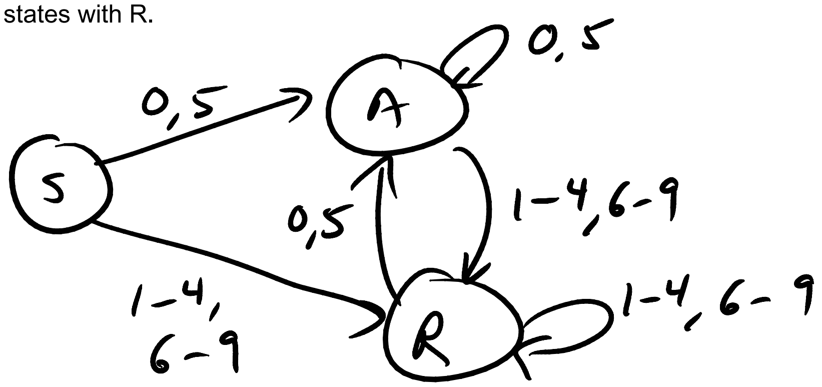
evalprefix ["+"; "3"; "+"; "5"; "6"] = (14, [])

evalprefix ["3"; "+"; "5"; "6"] = (3, ["+"; "5"; "6"])

You can assume all arguments are well-formed in the sense that they start with a prefix expression. (We will never supply arguments like the last one, but your function will need to handle them for the recursion to work out.) The function int_of_string converts a string containing decimal digits to an int.

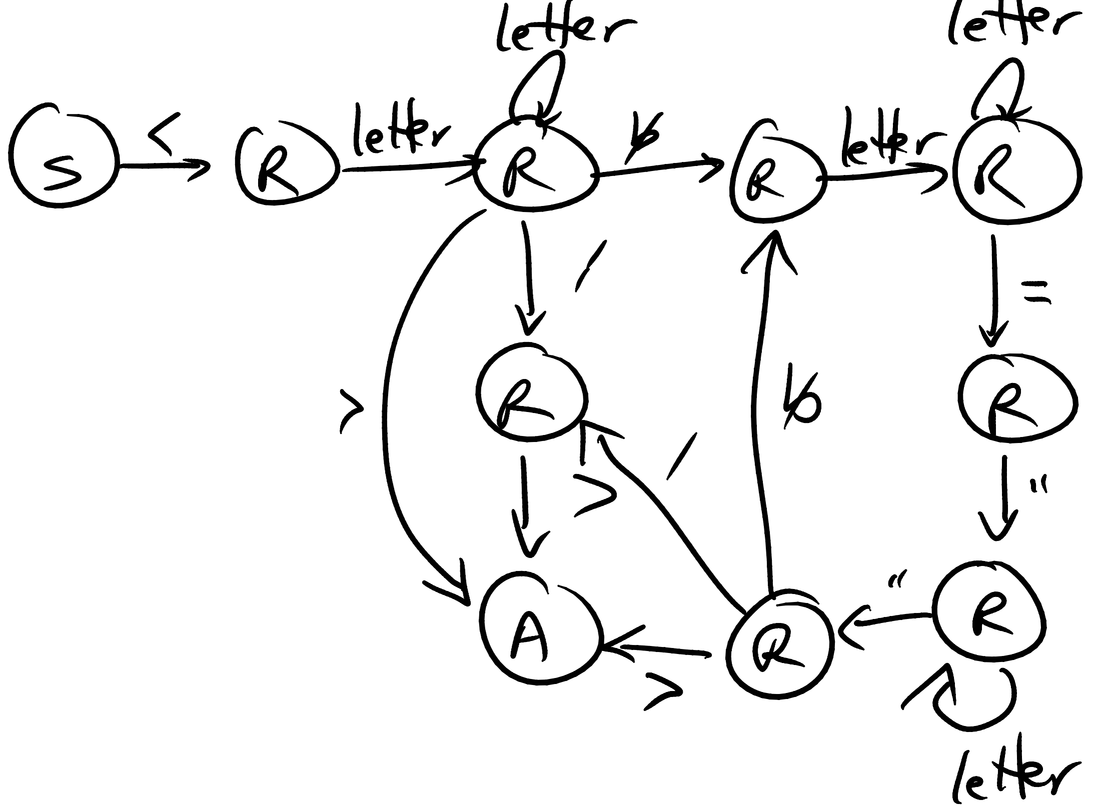
```
let rec evalprefix exp = match exp with
   "+" :: exp' -> let (v, exp'') = evalprefix exp'
                  in let (v', exp''') = evalprefix exp''
                    in (v+v', exp''')
 | n::exp' -> (int_of_string n, exp')
```

7. (4 pts) A number is a sequence of one or more digits (0 - 9). Give a DFA that recognizes numbers that are divisible by 5. Label the start state with S, accepting states with A, and rejecting states with R.



8. (8 pts) An XML node has the following format: < tag attribute=value … attribute=value > (or />). Assume tags and attributes consist only of letters, and values are strings beginning and ending with double quotes and containing only letters. Note that a node can end either with '>' or with '/>'. For purposes of this question assume that a single space separates the tag from the first attribute (if any), a single space separates one attribute from the next, and there are no other spaces. There can be any number of attributes, including zero (in which case there should be no spaces).

Write a DFA for XML nodes. Label the start state with S, accepting states with A, and rejecting states with R. (Note: we are not asking for a specification for XML *documents*, which have nesting and can only be specified with a context-free grammar; just for the nodes.)



9. (8 pts) Give an ocamllex specification for XML nodes as described in the previous question. You may do this as a single regular expression if you like, or as a more complicated specification. For the purposes of this question, you may be more liberal about spaces if you like; you can stick to the specification given in question 8, or you can allow more spaces; in particular, spaces after the tag, after each attribute, and before the closing bracket, are legal in XML. You can return zero as the value of the node.

```
let letter = ['A' - 'Z'] | ['a' - 'z']


rule tokenize = parse
  (* Add your regular expression here*)

        "<" letter+ (" " letter+ "=\"" letter* "\"")* (">"|"/>") { 0 }
```

10. (8 pts extra credit) Modify the ocamllex spec from question 9 so that it returns as its value a pair containing the tag and a list of key-value pairs. That is, it has type string * ((string * string) list).

```
let letter = ['A' - 'Z'] | ['a' - 'z']

rule tokenize = parse

        "<" (letter+ as tag)  { (tag, attributes lexbuf) }


and attributes = parse
        | " " (letter+ as attr) "=\"" (letter* as value) "\""
                                   {(attr,value)::attributes lexbuf}
         | (">"|"/>")          { [] }
```

11. (10 pts) This is a grammar for Ocaml int lists:

list $\rightarrow$ [ ] | [numbers]

numbers $\rightarrow$ int | int ; numbers

(a) Explain why this grammar is not LL(1).

For both non-terms, right-hand sides have overlapping FIRST sets.

(b) Here is a top-down (i.e. LL(1)) grammar for the same language:

list $\rightarrow$ [list2

list2 $\rightarrow$] | numbers]

numbers $\rightarrow$ int numbers2

numbers2 $\rightarrow$ ϵ | ; numbers

Write top-down parsing functions parseList and parseList2, both with type token list $\rightarrow$ (token list) option. You can assume that parseNumbers and parseNumbers2, with the same type, are provided. The set of tokens is defined by this type definition:

type token = Lbracket | Rbracket | Int | Semicolon

(Recall that the option type has definition: type ‘a option = None | Some of ‘a.)

```
let rec parseList tlis = if hd tlis = Lbracket
                         then parseList2 (tl tlis)
                         else None

and parseList2 tlis = if hd tlis = Rbracket
                      then Some (tl tlis)
                      else match parseNumbers tlis with
                           None -> None
                         | Some tlis' ->
                              if hd tlis' = Rbracket
                              then Some (tl tlis')
                              else None
```

12. (6 pts) A grammar for arithmetic expressions should have these properties, if possible:

(a) It should be unambiguous

(b) It should be LL(1)

(c) It should enforce left-associativity of + and *

(d) It should enforce precedence of * over +

None of the following grammars satisfies all of these criteria. For each grammar, list all the properties that it *fails* to satisfy (using letters a-d):

(1) E $\rightarrow$ id | E+id | E * id | (E) Fails: b, c, d

(2) E $\rightarrow$ id | id+E | id*E | (E) Fails: b, c, d

(3) E $\rightarrow$ T + E | T Fails: b, c
T $\rightarrow$ P * T | P
P $\rightarrow$ id | (E)

(4) E $\rightarrow$ id F | (E) Fails: c, d
F $\rightarrow$ ϵ | + E | * E

(5) E $\rightarrow$ E + T | T Fails: b
T $\rightarrow$ T * P | P
P $\rightarrow$ id | (E)

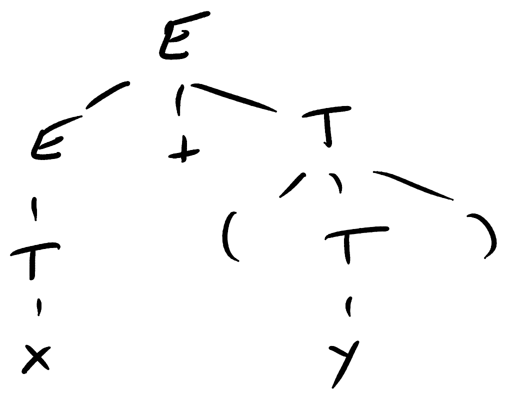
(6) E $\rightarrow$ T E' Fails: c
E' $\rightarrow$ ϵ | + E
T $\rightarrow$ P T'
T' $\rightarrow$ ϵ | * T
P $\rightarrow$ id | (E)

13. (10 pts) For this question, we will use the following grammar:

E $\to$ E + T | T

T $\to$ id | (T)

(a) Give the parse tree for sentence x+(y)



(b) Based on this parse tree, show the entire shift-reduce parse of this sentence, giving every shift and reduce action. For each reduce action, say what production is being reduced. For the stack, show only the top node of the trees (as we did in class). We have filled in the outline of this parse, showing that the first action is shift, and the last action is accept; the number of rows is exactly the correct number.

Action	Stack	Input
Shift		x+(y)
R T→id	x	+(y)
R E→T	T	+(y)
sh	E	+(y)
Sh	E+	(y)
Sh	E+(	y)
RT→id	E+(y	)
Sh	E+(T	)
R T→(T)	E+(T)	
R E→E+T	E+T	
Acc	E	

14. (8 pts) The following grammar is ambiguous.

```
        Exp → Exp or Exp
              | if Exp then Exp else Exp
              | not Exp
              | true
              | false
```

Two sentences that have more than one parse are "not true or false" (could be interpreted as "(not true) or false", or "not (true or false)"), and "if true then false else true or false" (could be interpreted as "(if true then false else true) or false" or "if true then false else (true or false)").

Give an ocamlyacc specification that resolves the ambiguity as follows:

- not has higher precedence than or.
- or is left assoc.
- or has higher precedence than an if-expression.

You should include all required declarations. The semantic actions should return abstract syntax trees of type ast:

type ast = Or of ast * ast | If of ast * ast * ast | Not of ast | True | False

```
%{
  type ast = Or of ast * ast | If of ast * ast * ast |
        Not of ast | True | False
%}

%token true false not if then else or
%nonassoc else
%left or
%nonassoc not
%start Exp
%type <ast> Exp
%%
Exp :
     Exp or Exp     { Or($1, $3) }
   | if Exp then Exp else Exp { If($2, $4, $6) }
   | not Exp        { Not $2 }
   | true           { True }
   | false          { False }
```

15. (6 pts Extra credit) No ambiguous grammar is LALR(1) – every ambiguous grammar has at least one conflict. The following grammars are ambiguous. For each one, do the following: (1) Find a sentence that has two parse trees. (2) Show the two parse trees. (3) Based on those parse trees, show a stack/lookahead configuration where there are two different actions – either a shift and a reduce, or two different reduces – that would lead to the two parse trees shown, and say which action leads to which tree.

(a) P → a | P P

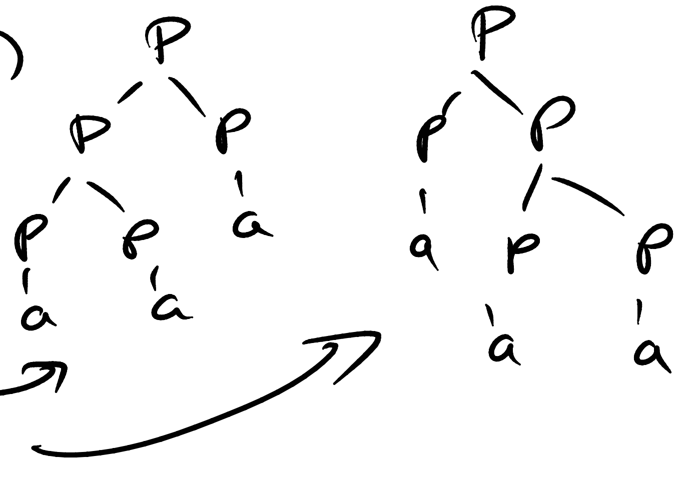
(1) aaa

(3) Stack: P P
Lookahead: a

Red P→PP gives

Shift gives

(2)



(b) S → T | U
T → T c | A
A → a A b | a b
U → a U | B
B → b B c | b c

(1) aabbcc

(2)
Stack: a a b
Lookahead: b

Red A→b

Shift

(2)