# Solutions for Sample Questions for Midterm 2 (CS 421 Fall 2019)

On the actual midterm, you will have plenty of space to put your answers.
Some of these questions may be reused for the exam.

1. Write the definition of an OCAML variant type **reg_exp** to express abstract syntax trees for regular expressions over a base character set of booleans. Thus, a boolean is a **reg_exp,** epsilon is a **reg_exp**, a parenthesized **reg_exp** is a **reg_exp**, the concatenation of two **reg_exp**'s is a **reg_exp,** the "choice" of two **reg_exp**'s is a **reg_exp**, and the Kleene star of a **reg_exp** is a **reg_exp**.

**Solution:**
```
type reg_exp =
  Char of bool
 | Epsilon
 | Paren of reg_exp
 | Concat of (reg_exp * reg_exp)
 | Choice of (reg_exp * reg_exp)
 | Kleene_star of reg_exp
```

2. Given the following OCAML datatype:

**type int_seq = Null | Snoc of (int_seq * int)**

write a tail-recursive function in OCAML **all_pos : int_seq -> bool** that returns **true** if every integer in the input **int_seq** to which **all_pos** is applied is strictly greater than 0 and **false** otherwise. Thus **all_pos (Snoc(Snoc(Snoc(Null, 3), 5), 7))** should returns **true,** but ) **all_pos (Snoc(Null, -1))** and **all_pos (Snoc(Snoc(Null, 3),0))** should both return **false.**

**Solution:**
```
let rec all_pos s =
   (match s with Null -> true
     | Snoc(seq, x) -> if x <= 0 then false else all_pos seq);;
```

3. Given a polymorphic type derivation for {} |- let id = fun x -> x in id id true : bool

**Solution:**

Let Γ = {id : ∀ ‘a. ‘a -> ‘a}

```
                             Instance: ‘a → bool -> bool
                         Var ----------------------------------          Instance: ‘a → bool
                               Γ |-  id : (bool -> bool) ->      Var---------------------------
                                          bool -> bool               Γ |-  id : bool -> bool          Const
  Var ---------------------     App-----------------------------------------------------------    -------------------
      { x : ‘a }  |-  x : ‘a               Γ |-  id id : bool -> bool                               Γ |-  true : bool
 Fun -----------------------      App--------------------------------------------------------------------------------
       { } |- fun x -> x                  {id : ∀ ‘a. ‘a -> ‘a} |-  id id true
   Let -------------------------------------------------------------------------------------
                         { } |- let id = fun x -> x in id id true : bool
```

4. Write the clause for **gather_exp_ty_substitution** for a function expression implementing the rule:

$$\frac{[x : \tau_1] + \Gamma \mid\!\!- \, e : \tau_2 \mid \sigma}{\Gamma \mid\!\!- \, (\textbf{fun } x \texttt{ -> } e) : \tau \mid \underline{\text{unify}\{(}\sigma(\tau), \sigma(\tau_1 \texttt{->} \tau_2))\} \circ \sigma}$$

Refer to MP6 for the details of the types. You should assume that all other clauses for **gather_exp_ty_substitution** have been provided.

**Solution:**

```
let rec gather_exp_ty_substitution gamma exp tau =
   let judgment = ExpJudgment(gamma, exp, tau) in
        match exp
        with . . .
        | FunExp(x,e) ->
          let tau1 = fresh() in
          let tau2 = fresh() in
          (match gather_exp_ty_substitution
                (ins_env gamma x (polyTy_of_monoTy tau1)) e tau2
           with None -> None
           | Some (pf, sigma) ->
             (match unify [(monoTy_lift_subst sigma tau,
                        monoTy_lift_subst sigma (mk_fun_ty tau1 tau2))]
              with None -> None
              | Some sigma1 ->
          Some(Proof([pf],judgment), subst_compose sigma1 sigma)))
```

5. Give a (most general) unifier for the following unification instance. Capital letters denote variables of unification. Show your work by listing the operation performed in each step of the unification and the result of that step.

{X = f(g(x),W); h(y) = Y;  f(Z,x) = f(Y,W)}

**Solution:**

Unify {X = f(g(x),W); h(y) = Y;  f(Z,x) = f(Y,W)}
= Unify {h(y) = Y; f(Z,x) = f(Y,W)} o {X → f(g(x),W)}     by eliminate  (X = f(g(x),W))
= Unify {Y = h(y); f(Z,x) = f(Y,W)} o {X → f(g(x),W)}     by orient  (h(y) = Y)
= Unify {f(Z,x) = f(h(y),W)} o {X → f(g(x),W), Y → h(y)} by eliminate  (Y = h(y))
= Unify {Z = h(y); x=W} o {X → f(g(x),W), Y → h(y)}     by decompose  (f(Z,x) = f(h(y),W))
= Unify {x = W} o {X → f(g(x),W), Y → h(y), Z → h(y)}   by eliminate  (Z = h(y))
= Unify {W = x} o {X → f(g(x),W), Y → h(y), Z → h(y)}   by orient  (x = W)
= Unify{} o {X → f(g(x),x), Y → h(y), Z → h(y), W → x}  by eliminate  (W = x)
Answer: {X → f(g(x),x), Y → h(y), Z → h(y), W → x}

6. For each of the following descriptions, give a regular expression over the alphabet {a,b,c}, and a regular grammar that generates the language described.

   a. The set of all strings over {**a**, **b**, **c**}, where each string has at most one **a**

      **Solution: (b ∨ c)*(a ∨ ε) (b ∨ c)***

      **<S> ::= b<S> | c<S> | a<NA> | ε**

      **<NA> ::= b<NA> | c<NA> | ε**

b. The set of all strings over {**a**, **b**, **c**}, where, in each string, every **b** is immediately followed by at least one **c**.

**Solution: (a ∨ c)*(bc(a ∨ c)*)***

**<S> ::= a<S> | c<S> | b<C> | ε**
**<C> ::= c<S>**

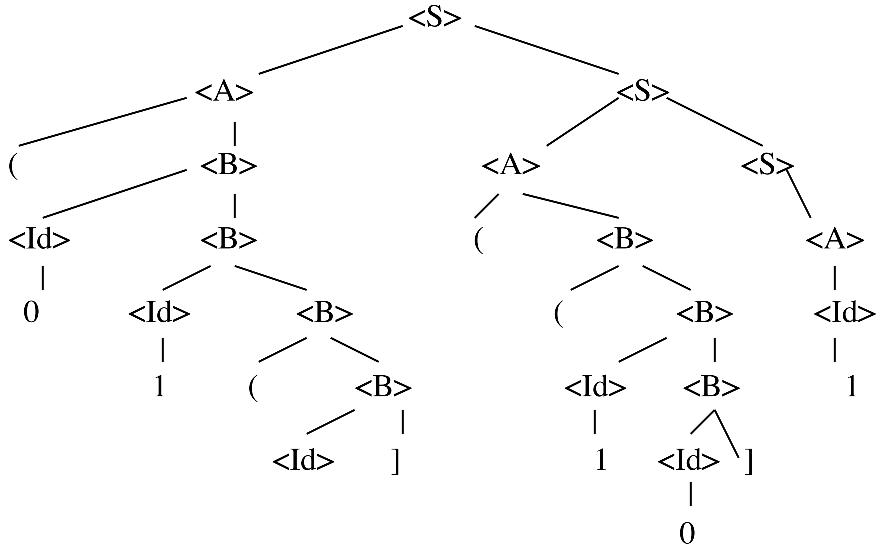
c. The set of all strings over {**a**, **b**, **c**}, where every string has length a multiple of four.

**Solution: ((a ∨ b ∨ c) (a ∨ b ∨ c) (a ∨ b ∨ c) (a ∨ b ∨ c))***

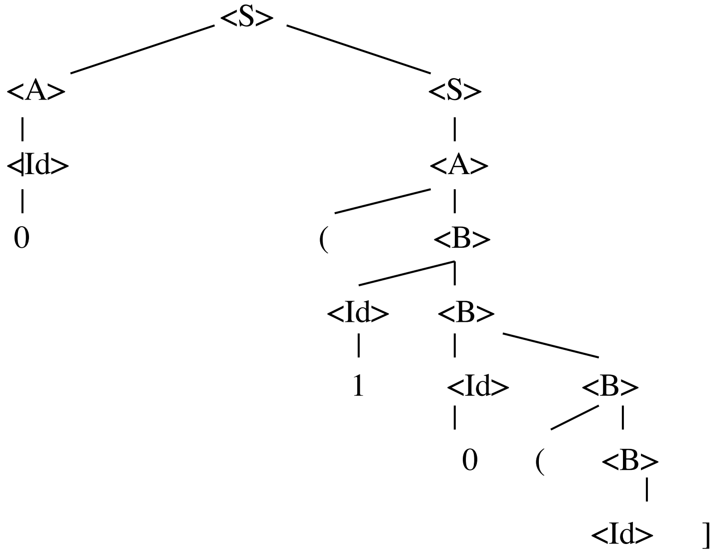
**<S> ::= a<TH> | b<TH> | c<TH> | ε**
**<TH> ::= a<TW> | b<TW> | c<TW>**
**<TW> ::= a<O> | b<O> | c<O>**
**<O> ::= a<S> | b<S> | c<S>**

7. Consider the following grammar:
   <S> ::= <A> | <A> <S>
   <A> ::= <Id> | ( <B>
   <B> ::= <Id> ] | <Id><B> | ( <B>
   <Id> ::= 0 | 1
   For each of the following strings, give a parse tree for the following expression as an <S>, if one exists, or write “No parse” otherwise:

   a. ( 0 1 ( 1 ] ( ( 1 0 ] 1
   **Solution:**



   b. 0 ( 1 0 ( 1 ]
   **Solution:**

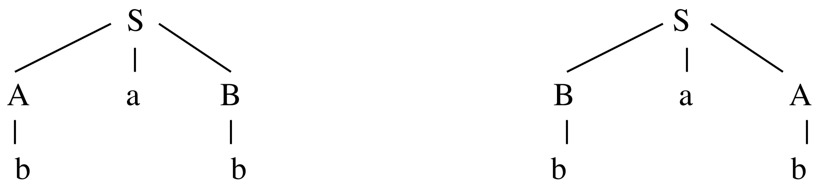


   c. ( 0 ( 1 0 1] 0 ]
   **Solution:** No parse tree

8. Demonstrate that the following grammar is ambiguous (Capitals are non-terminals, lowercase are terminals):

$$S \rightarrow A\ a\ B \mid B\ a\ A$$
$$A \rightarrow b \mid c$$
$$B \rightarrow a \mid b$$

**Solution:** String: bab



9. Write an unambiguous grammar generating the set of all strings over the alphabet {0, 1, +, **-**} , where + and – are infixed operators which both associate to the left and such that + binds more tightly than -.

**Solution:**

**<S> ::= <plus> | <S> - <plus>**
**<plus> :: <id> | <plus> + <id>**
**<id> ::= 0 | 1**

10. Write an ocamlyacc parser for the language

<S> = <V> | fun <V> -> <S> | <S><S>

where <V> is any identifier, application (<S><S>) associates to the left and has higher precedence than fun <V> -> <S> . You should include in your header one or more datatypes for the abstract syntax trees of the language. You declarations should give the constructors for the datatype **token** of the tokens input into the parser.

**Solution:**

```
%{

 type term = Var of string | Fun of (string * term) | App of (term * term)

%}

%token <string> IDENT
%token ARROW FUN EOF

%start main
```

```
%type <term> main

%%

main:
  term EOF      { $1 }

term:
  not_app_pos_fun                        { $1 }
| pos_app_no_fun not_app_pos_fun  { App($1, $2) }

pos_app_no_fun:
  pos_app_no_fun IDENT                 { App($1, Var $2) }
| IDENT                                       { Var $1 }

not_app_pos_fun:
  FUN IDENT ARROW term             { Fun($2,$4) }
| IDENT                                      { Var $1 }
```