Solutions for Sample Questions for Midterm 3 (CS 421 Fall 2025)

Some of these questions may be reused for the exam.

There are more questions here than would be on the actual midterm. They represent the ideas that would be the basis of questions you may be asked. The actual midterm will have 5 regular zones and one extra credit one. The balance will attempt to reflect the amount of time spent in lecture on a topic, broadly speaking, and the amount of time for the exam.

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

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

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)))
```

2. Give a (most general) unifier for the following unification instance. Capital letters denote variables of unification. Lowercase letters denote term constructors. The constructor f has arity 2, g and h have arity 1 and x and y have arity 0. 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 $\{h(y) = Y;\ f(Z,x) = f(Y,W)\} \circ \{X \rightarrow f(g(x),W)\}$ by eliminate $(X = f(g(x),W))$
= Unify $\{Y = h(y);\ f(Z,x) = f(Y,W)\} \circ \{X \rightarrow f(g(x),W)\}$ by orient $(h(y) = Y)$
= Unify $\{f(Z,x) = f(h(y),W)\} \circ \{X \rightarrow f(g(x),W),\ Y \rightarrow h(y)\}$ 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}

3. Write a function

unify_eliminate : typeVar -> monoTy -> (monoTy * monoTy) list ->
((monoTy * monoTy) list -> (typeVar * monoTy) list option) ->
(typeVar * monoTy) list option

where is **unify_eliminate** given the arguments **ns** is a typeVar, **t** is a monoTy, **rem_constraints** is a list of monoTy pairs describing a set of equational constraints on monoTys, and **unify_rem** is a function capable of returning a substitution capable of solving **rem_constraints** if a solution exists, but is not necessarily capable of solving any other set of constraints. When fully applied, **unify_eliminate** then returns a solution to the larger set of constraints: **(TyVar ns, t) :: rem_constraints**. You may assume **occurs: typeVar -> monoTy -> bool**, and **monoTy_lift_subst : (typeVar * monoTy) list -> monoTy -> monoTy**.

Solution:

```
let unify_eliminate n t rem_constraints unify =
 if (occurs n t)
 then None
 else
  let new_constraints =
   List.map
    (fun (t1,t2) ->
     (monoTy_lift_subst [(n,t)] t1, monoTy_lift_subst [(n,t)] t2))
    rem_constraints
  in
  (match unify new_constraints
   with None -> None
    | Some(phi) -> Some((n, monoTy_lift_subst phi t):: phi))
```

4. 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>

5. This problem is too big to be an exam question, but it can be cut down to a few different problems that would be possible. For example, just having strings is a plausible exam problem. I include the while problem because it poses several difficulties to challenge you, and because the end result is actually useful.

Write an ocamllex file, **basic_csv_lex.mll,** that translate a (simplified) comma separated values (.csv) file into a reversed list of reversed lists of entries, where an entry is an integer, a float, or a string. Entries are represented using the following type:

type csv_entry = INT of int | FLOAT of float | STRING of string

The file will contain newline ended rows of comma separated values, where a value is representation of an integer, float or string. An integer is represented as a nonempty sequence of digits, possibly preceded by a minus sign, where the leading digit is a 0 only if the integer is a 0 and 0 is not preceded by a minus sign. Floats are similar, but with a decimal point. So a float may start with a minus sign but then must have a nonempty sequence of digits, starting with 0 only if the integer part is 0, followed by a period, follow by a nonempty sequence of digits that does not end in a 0. (So, 1. , 1.0, .9 and -.5 are some examples of things that are not legal.) There are two representations of strings. Strings only use printable characters, as described in MP8, but including \ as an ordinary character. One type of string representation is any (possibly empty) sequence of printable characters not including a double quotes or comma. Its value should be the string that contains that sequence of characters. If the same sequence also represents an integer of a float, it should be translated as the integer or float, not a string. The second type of string representation must begin with and end with a single double quotes. Inside commas may freely be used. Since double quotes end the string, they can't appear inside the string without special support. This time, a double quote inside a string is represents by a pair of double quotes. So """" represents a string that has one character which is a double quotes. On an exam, you would be given some of the contents described here as starter code. A sample contents for a csv file is

""""",Does,""""""""",this ,""", ?","""work"""
\n,4,-0.4,33,0.1,0.2

Be sure to put only ASCII characters into your test file, or use test_csv.csv.

Solution: (see basic_csv_lec.mll in exams)

```
{
type csv_entry = INT of int | FLOAT of float | STRING of string
(* csv_entries gives back a reverse list of reserved lists of csv_entries *)
}
let nzdigit       = ['1' - '9']
let digit         = ['0' - '9']
let lowercase     = ['a' - 'z']
let uppercase     = ['A' - 'Z']
let letter        = uppercase | lowercase
let natural       = '0' | (nzdigit (digit *))
let integer       = natural | '-' natural
let float         = integer '.' (((digit *) nzdigit) | '0')

let noncomma_string_char
              =  letter | digit | '_' | '\'' | '-'
                | ' ' | '~' | '`' | '!' | '@' | '#'
                | '$' | '%' | '^' | '&' | '*' | '(' | ')'
```

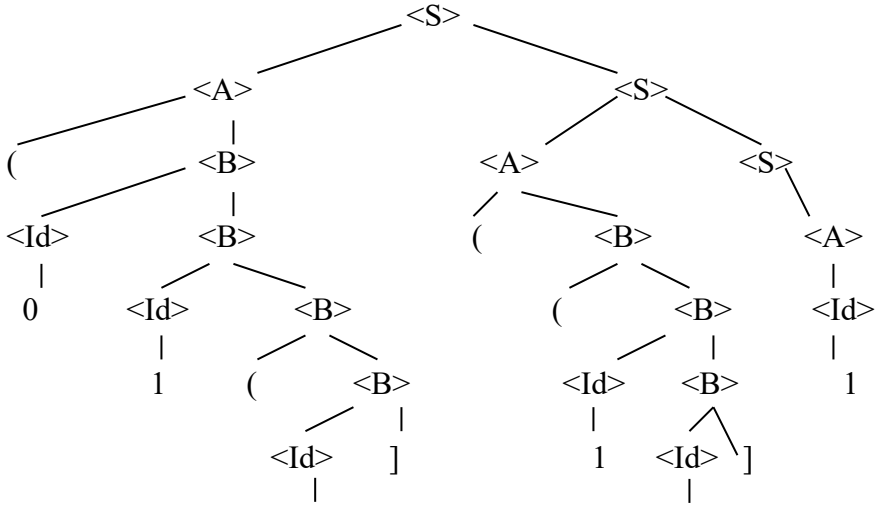
```
                | '+' | '=' | '{' | '[' | '}' | '\'' | ']' | '|' | '\\'
                | ':' | ';' | '<' | '>' | '.' | '?' | '/'
let comma = ','
let newline = '\n' | '\r'
let string_char     = noncomma_string_char | comma

rule csv_entries rev_entries rev_lines = parse
  | (float as flt) comma
     { csv_entries
        ((FLOAT (float_of_string flt)) :: rev_entries)
        rev_lines
        lexbuf }
  | (float as flt) newline
     { csv_entries
        []
        (((FLOAT (float_of_string flt)) :: rev_entries) :: rev_lines)
        lexbuf }
  | (float as flt) eof
     { ((FLOAT (float_of_string flt)) :: rev_entries) :: rev_lines }
  | (integer as int) comma
     { csv_entries
        ((INT (int_of_string int)) :: rev_entries)
        rev_lines
        lexbuf }
  | (integer as int) newline
     { csv_entries
        []
        (((INT (int_of_string int)) :: rev_entries) :: rev_lines)
        lexbuf }
  | (integer as int) eof
     { ((INT (int_of_string int)) :: rev_entries) :: rev_lines }
  | comma
     { csv_entries ((STRING "") :: rev_entries) rev_lines lexbuf }
  | newline
     { csv_entries [] (((STRING "") :: rev_entries) :: rev_lines) lexbuf }
  | eof
     { ((STRING "") :: rev_entries) :: rev_lines }

  | ((noncomma_string_char +) as str) comma
     { csv_entries ((STRING str) :: rev_entries) rev_lines lexbuf }
  | ((noncomma_string_char +) as str) newline
     { csv_entries [] (((STRING str) :: rev_entries) :: rev_lines) lexbuf }
  | ((noncomma_string_char +) as str) eof
     { ((STRING str) :: rev_entries) :: rev_lines }
  | '\"'
     { string "" rev_entries rev_lines lexbuf }
```

```
and string str rev_entries rev_lines = parse
  | (string_char+ as s)
     { string (str ^ s) rev_entries rev_lines lexbuf }
  | "\"\""
     { string (str ^ "\"") rev_entries rev_lines lexbuf }
  | "\"" comma
     { csv_entries ((STRING str) :: rev_entries) rev_lines lexbuf  }
  | "\"" newline
     { csv_entries [] (((STRING str) :: rev_entries) :: rev_lines) lexbuf  }
  | "\"" eof
     { ((STRING str) :: rev_entries) :: rev_lines }


{

let csv_lex lexbuf = csv_entries [] [] lexbuf

let get_csv_string s = csv_lex (Lexing.from_string s)

let get_csv_file file = csv_entries [] [] (Lexing.from_channel (open_in file))

}
```

6. 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

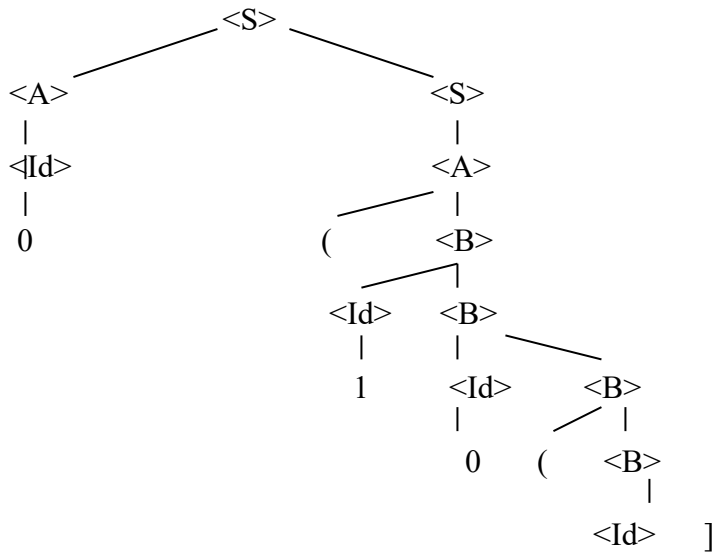
   Solution:



1 0

b. 0 (1 0 (1]

Solution:



c. (0 (1 0 1] 0]

Solution: No parse tree

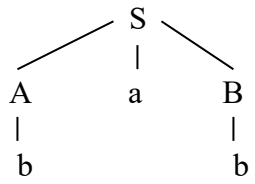
7. Demonstrate that the following grammar is ambiguous (Capitals are non-terminals, lowercase are terminals):

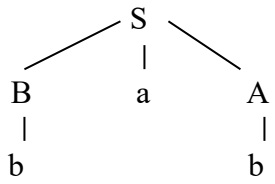
$S \rightarrow A\ a\ B \mid B\ a\ A$

$A \rightarrow b \mid c$

$B \rightarrow a \mid b$

Solution: String: bab





8. 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 –, but where – is also a unary operator that binds more tightly than either + or – used as binary operators.

Solution:

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