Solutions for Sample Questions for Midterm 3 (CS 421 Spring 2023)

Some of these questions may be reused for the exam.

1. 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 $\{X = f(g(x),W);\ h(y) = Y;\ f(Z,x) = f(Y,W)\}$
$= \text{Unify } \{h(y) = Y;\ f(Z,x) = f(Y,W)\} \circ \{X \to f(g(x),W)\}$ by eliminate $(X = f(g(x),W))$
$= \text{Unify } \{Y = h(y);\ f(Z,x) = f(Y,W)\} \circ \{X \to f(g(x),W)\}$ by orient $(h(y) = Y)$
$= \text{Unify } \{f(Z,x) = f(h(y),W)\} \circ \{X \to f(g(x),W),\ Y \to h(y)\}$ by eliminate $(Y = h(y))$
$= \text{Unify } \{Z = h(y);\ x=W\} \circ \{X \to f(g(x),W),\ Y \to h(y)\}$ by decompose $(f(Z,x) = f(h(y),W))$
$= \text{Unify } \{x = W\} \circ \{X \to f(g(x),W),\ Y \to h(y),\ Z \to h(y)\}$ by eliminate $(Z = h(y))$
$= \text{Unify } \{W = x\} \circ \{X \to f(g(x),W),\ Y \to h(y),\ Z \to h(y)\}$ by orient $(x = W)$
$= \text{Unify}\{\} \circ \{X \to f(g(x),x),\ Y \to h(y),\ Z \to h(y),\ W \to x\}$ by eliminate $(W = x)$
Answer: $\{X \to f(g(x),x),\ Y \to h(y),\ Z \to h(y),\ W \to x\}$

2. 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))

3. 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>

4. 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 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))

}
```

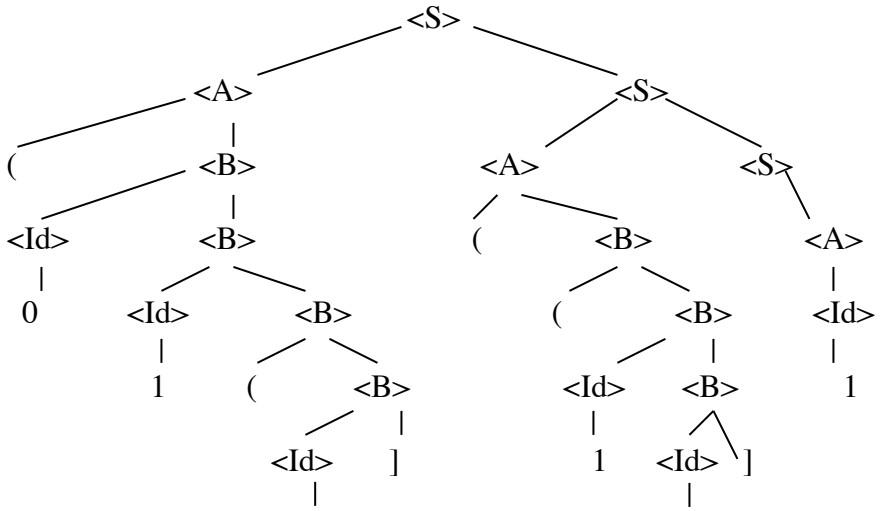
5. Consider the following grammar:

<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

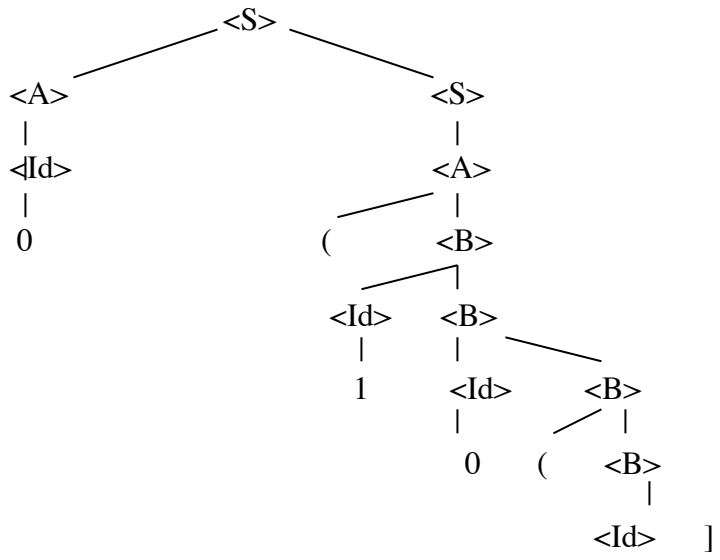
Solution:



1 0

b. 0 (1 0 (1]

Solution:



c. (0 (1 0 1] 0]

Solution: No parse tree

6. You are given the following grammar over nonterminal <s>, <b> and <e>, and terminals c, x, l, and eof, with start symbol <s>:

P0: <s> ::= <e>eof
P1: <e> ::= c<b><e><e>

P2: <e> ::= x
P3: <b> ::= <e>l

and Action and Goto tables generated by YACC for the above grammar:

	Action					Goto		
State	x	c	l	[eof]		<b>	<e>	<s>
st1	S3	S4	Err	Err			st2	
st2	Err	Err	Err	A				
st3	R2	R2	R2	R2				
st4	S3	S4	Err	Err		st6	st5	
st5	Err	Err	S7	Err				

st6	S3	Err	S8	Err		st6	st8	
st7	R3	R3	R3	R3				
st8	S3	S4	Err	Err			st9	
st9	R1	R1	R1	R1				

where sti is state i, Si abbreviates shift i, Ri abbreviates reduce i, A abbreviates accept, RErr abbreviates error and [eof] means we have reached the end of input. Describe how the string cxlxx[eof] would be parsed with an LR parser using these productions and tables by filling in the table on the next page. I have given you the first 7 cells in the first three rows to get you started. Thereafter, there are more blank lines than you should need to fill in the rest.

Solution:

Stack	Current unprocessed String	Action to be taken
Empty	cxlxx[eof]	Initialize stack, go to state 1
st1	cxlxx[eof]	Shift c, go to state 4
st1::c::st4	xlxx[eof]	Shift x, go to state 3
st1::c::st4::x::st3	lxx[eof]	Reduce by rule 2, pop back to st4, push <e>, go to state 5
st1::c::st4::<e>::st5	lxx[eof]	Shift l, go to state 7
st1::c::st4::<e>::st5::l::st7	xx[eof]	Reduce by rule 3, pop back to st4, push <b>, go to state 6
st1::c::st4::<b>::st6	xx[eof]	Shift x, go to state 3
st1::c::st4::<b>::st6::x::st3	x[eof]	Reduce by rule 2, pop back to st6, push <e>, go to state 8
st1::c::st4::<b>::st6::<e>::st8	x[eof]	Shift x, go to state 3
st1::c::st4::<b>::st6::<e>::st8::x::st3	[eof]	Reduce by rule 2, pop back to st8, push <e>, go to state 9
st1::c::st4::<b>::st6::<e>::st8::<e>::st9	[eof]	Reduce by rule 1, pop back to st1, push <e>, go to state 2
st1::<e>::st2	[eof]	Accept