CS421 Fall 2012 Midterm 2

Name:	
NetID:	

- You have **75 minutes** to complete this exam.
- This is a **closed-book** exam. You are allowed one 3×5 inch (or smaller) card of notes (both sides may be used). This card is **not shared**. All other materials (e.g., calculators and cell phones), except writing utensils are prohibited.
- Do not share anything with other students. Do not talk to other students. Do not look at another students 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, you may seek clarification from myself or one of the TAs. You must use a whisper, or write your question out. Speaking out aloud is not allowed.
- Including this cover sheet and rules at the end, there are 17 pages to the exam, including two blank pages for workspace. Please verify that you have all 17 pages.
- Please write your name and NetID in the spaces above, and also in the provided space at the top of every sheet.

Question	Points	Bonus Points	Score
1	13	0	
2	8	0	
3	13	0	
4	15	0	
5	15	0	
6	24	0	
7	12	0	
8	0	10	
Total:	100	10	

Problem 1. (13 points)

Use the unification algorithm described in class and in MP7 to give a most general unifier for the following set of equations (unification problem), if one exists, or to say why if one does not. In this problem, we use = as the separator for constraints. The uppercase letters X, Y, Z, and W denote variables of unification. The lowercase letters f, g, and h are term constructors of arity 2, 3, and 1 respectively (*i.e.* take two, three or one argument(s), respectively). Show all your work by listing the operations performed in each step of the unification and the result of that step.

$$\mathsf{Unify}\{(f(X, g(Y, Z, h(W))) = f(g(Z, Y, Z), X))\}$$

Solution:

Rule Resulting Equations / Substitution

Given

$\mathsf{Unify}\{(f(X, g(Y, Z, h(W))) = f(g(Z, Y, Z), X))\}$

by Decompose $(f(X, g(Y, Z, h(W))) = f(g(Z, Y, Z), X))$

$= \mathsf{Unify}\{(X = g(Z, Y, Z)); (g(Y, Z, h(W)) = X)\}$

by Eliminate $(X = g(Z, Y, Z))$

$= \mathsf{Unify}\{(g(Y, Z, h(W)) = g(Z, Y, Z))\} \circ \{X \mapsto g(Z, Y, Z)\}$

by Decompose $(g(Y, Z, h(W)) = g(Z, Y, Z))$

$= \mathsf{Unify}\{(Y = Z); (Z = Y); (h(W) = Z)\} \circ \{X \mapsto g(Z, Y, Z)\}$

by Eliminate $(Y = Z)$

$= \mathsf{Unify}\{(Z = Z); (h(W) = Z)\} \circ \{Y \mapsto Z\} \circ \{X \mapsto g(Z, Y, Z)\}$

$= \mathsf{Unify}\{(Z = Z); (h(W) = Z)\} \circ \{X \mapsto g(Z, Z, Z); Y \mapsto Z\}$

by Delete $(Z = Z)$

$= \mathsf{Unify}\{(h(W) = Z)\} \circ \{X \mapsto g(Z, Z, Z); Y \mapsto Z\}$

by Orient $(h(W) = Z)$

$= \mathsf{Unify}\{(Z = h(W))\} \circ \{X \mapsto g(Z, Z, Z); Y \mapsto Z\}$

by Eliminate $(Z = h(W))$

$= \mathsf{Unify}\{\} \circ \{Z \mapsto h(W)\} \circ \{X \mapsto g(Z, Z, Z); Y \mapsto Z\}$

$= \{X \mapsto g(h(W), h(W), h(W)); Y \mapsto h(W); Z \mapsto h(W)\}$

Problem 2. (8 points)

Recall that in MP6 and MP7 we used the following OCaml types to represent the types of MicroML, the language we have been implementing since MP5:

```
type typeVar = int
type monoTy = TyVar of typeVar | TyConst of (string * monoTy list)
```

Further recall that we represented substitutions of `monoTy`s for `typeVar`s by the type `(typeVar * monoTy) list`. The first component of a pair is the index (or "name") of a type variable. The second is the type that should be substituted for that type variable. If an entry for a type variable index does not exist in the list, the identity substitution should be assumed for that type variable (i.e. the variable is substituted with itself).

(a) (4 points) Implement the function `subst_fun` that converts a list representing a substitution into a function that takes a `typeVar` and returns a `monoTy`.

```
# let subst_fun s = ...
val subst_fun : (typeVar * monoTy) list -> typeVar -> monoTy = <fun>
# let subst = subst_fun [(5, mk_fun_ty bool_ty (TyVar(2)))];;
val subst : typeVar -> monoTy = <fun>
# subst 1;;
- : monoTy = TyVar 1
# subst 5;;
- : monoTy = TyConst ("->", [TyConst ("bool", []); TyVar 2])
```

Solution:

```
let rec subst_fun subst m =
    match subst with [] -> TyVar m
    | (n,ty) :: more -> if n = m then ty else subst_fun more m
```

(b) (4 points) Implement the `monoTy_lift_subst` function that *lifts* a substitution ϕ to operate on `monoTy`s. A substitution ϕ, when lifted, replaces all the type variables occurring in its input type with the corresponding types.

```
# let rec monoTy_lift_subst s = ...
val monoTy_lift_subst : (typeVar * monoTy) list -> monoTy -> monoTy = <fun>
# monoTy_lift_subst [(5, mk_fun_ty bool_ty (TyVar(2)))]
             (TyConst ("->", [TyVar 1; TyVar 5]));;
- : monoTy =
TyConst ("->", [TyVar 1; TyConst ("->", [TyConst ("bool", []); TyVar 2])])
```

Solution:

```
let rec monoTy_lift_subst subst monoTy =
    match monoTy
    with TyVar m -> subst_fun subst m
    | TyConst(c, typelist) ->
      TyConst(c, List.map (monoTy_lift_subst subst) typelist)
```

Problem 3. (13 points)

Consider the set of all strings over the alphabet { [,], 0, 1, ; } (*i.e.* left square bracket, right square bracket, 0, 1 and semicolon) that describe lists of non-empty sequences of 0's and 1's, separated by semicolon, preceded by a left square bracket and followed by a right square bracket. This set of strings includes [ ]. Singleton lists (*e.g.* [010] contain no semicolons.

(a) (5 points) Write a regular expression describing the set given above. In writing a regular express describing this set of strings, you may use the notation for basic regular expressions (Kleene's notation), or you may use ocmallex syntax, but these are the only syntax allowed.

Solution:

$$[\,(\varepsilon \vee (((0 \vee 1)(0 \vee 1)^*\,;)^*\,(0 \vee 1)(0 \vee 1)^*))\,]$$

(b) (8 points) Write a right regular grammar describing the same set of strings.

Solution:

$$\begin{aligned}
<\mathit{list}>::= &\ \ [<\mathit{contents}> \\
<\mathit{contents}>::= &\ \ ] \\
| &\ \ 0\ <\mathit{contents}> \\
| &\ \ 1\ <\mathit{contents}> \\
| &\ \ 0\ <\mathit{semicolon}> \\
| &\ \ 1\ <\mathit{semicolon}> \\
<\mathit{semicolon}>::= &\ \ ;\ <\mathit{num\_and\_contents}> \\
| &\ \ ;\ <\mathit{num}> \\
<\mathit{num\_and\_contents}>::= &\ \ 0\ <\mathit{contents}> \\
| &\ \ 1\ <\mathit{contents}> \\
<\mathit{num}>::= &\ \ 0\ <\mathit{num}> \\
| &\ \ 1\ <\mathit{num}> \\
| &\ \ 0 \\
| &\ \ 1
\end{aligned}$$

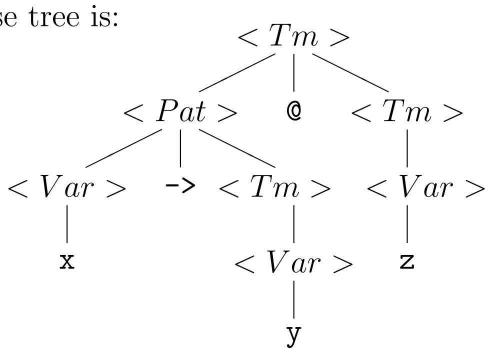
Problem 4. (15 points)

Given the following BNF grammar, for each of the following strings, give a parse tree for it, if it parses starting with $< Tm >$, or write None exists if it does not parse starting with $< Tm >$. The terminals for this grammar are { `@`, `->`, `b`, `d`, `x`, `y`, `z` }. The non-terminal are $< Tm >$, $< Pat >$, and $< Var >$.

$< Tm >$::= $< Pat >$ `@` $< Tm >$ | $< Var >$
$< Pat >$::= $< Var >$ `->` $< Tm >$ | $< Pat >$ `->` `b` $< Tm >$ `d`
$< Var >$::= `x` | `y` | `z`

(a) (3 points) `x -> y @ z`

Solution: The parse tree is:



Given the following BNF grammar, for each of the following strings, give a parse tree for it, if it parses starting with $< Tm >$, or write None exists if it does not parse starting with $< Tm >$. The terminals for this grammar are { `@`, `->`, `b`, `d`, `x`, `y`, `z` }. The non-terminal are $< Tm >$, $< Pat >$, and $< Var >$.

$$
\begin{array}{ll}
< Tm > ::= & < Pat > \texttt{ @ } < Tm > \mid < Var > \\
< Pat > ::= & < Var > \texttt{ -> } < Tm > \mid < Pat > \texttt{ -> b } < Tm > \texttt{ d} \\
< Var > ::= & \texttt{x} \mid \texttt{y} \mid \texttt{z}
\end{array}
$$

(b) (5 points) `x -> y -> b y -> x @ z -> x @ y d`

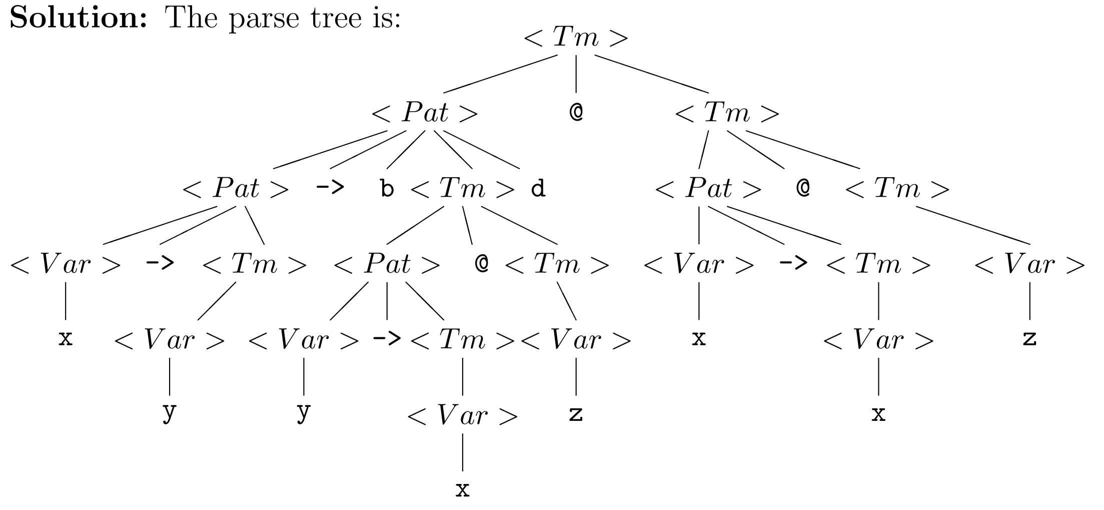
Solution: None exists. Every $< Tm >$ must end in a $< Var >$ (*i.e.* one of `x`, `y`, or `z`), but this string ends in a `d`.

Given the following BNF grammar, for each of the following strings, give a parse tree for it, if it parses starting with $< Tm >$, or write None exists if it does not parse starting with $< Tm >$. The terminals for this grammar are { `@`, `->`, `b`, `d`, `x`, `y`, `z` }. The non-terminal are $< Tm >$, $< Pat >$, and $< Var >$.

$< Tm >$::= $< Pat >$ `@` $< Tm >$ | $< Var >$
$< Pat >$::= $< Var >$ `->` $< Tm >$ | $< Pat >$ `->` `b` $< Tm >$ `d`
$< Var >$::= `x` | `y` | `z`

(c) (7 points) `x -> y -> b y -> x @ z d @ x -> x @ z`

Solution: The parse tree is:



< Tm >
< Pat > @ < Tm >
< Pat > -> b < Tm > d
< Pat > @ < Tm >
< Var > -> < Tm > < Pat > @ < Tm > < Var > -> < Tm > < Var >
x < Var > < Var > -> < Tm > < Var > x < Var > z
y y < Var > z x
x

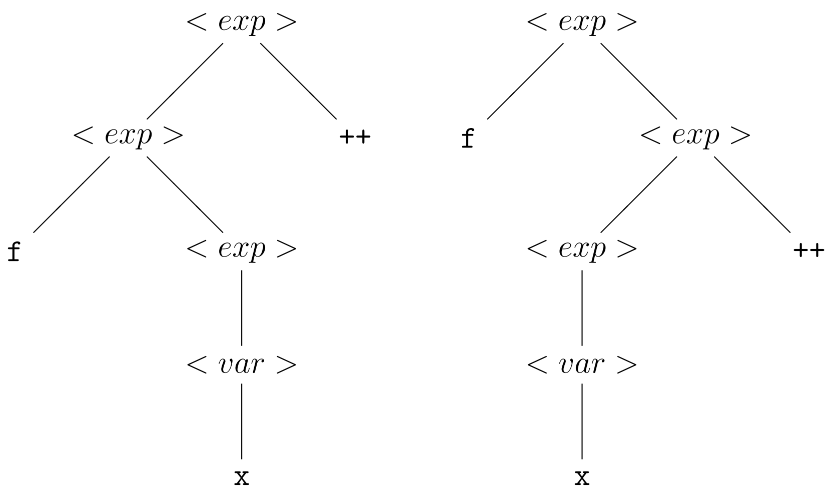
Problem 5. (15 points)

Consider the following grammar over the terminal alphabet { `f`, `(`, `)`, `++`, `x`, , `y`, `z` }.

$$
\begin{array}{ll}
< exp > ::= & \texttt{f} \; < exp > \;|\; < exp > \; \texttt{++} \;|\; \texttt{(} < exp > \texttt{)} \;|\; < var > \\
< var > ::= & \texttt{x} \;|\; \texttt{y} \;|\; \texttt{z}
\end{array}
$$

(a) (5 points) Show that this grammar is ambiguous (using the definition of an ambiguous grammar).

Solution: The term `f x ++` has two parse trees.



(b) (10 points) Disambiguate this grammar by writing a new grammar with start symbol $< exp >$ accepting the same language accepted by $< exp >$ above, and such that `++` has higher precedence than `f`.

Solution:

$$
\begin{array}{ll}
< exp > ::= & \texttt{f} \; < exp > \;|\; < no\_f > \\
< no\_f > ::= & < no\_f > \; \texttt{++} \;|\; (< exp >) \;|\; \texttt{x} \;|\; \texttt{y} \;|\; \texttt{z}
\end{array}
$$

Workspace

Problem 6. (24 points)

Consider the following grammar:

$$< term >=:: \texttt{+} \ < term > < term > \ | \ \tilde{} \ < term > \ | \ 0 \ | \ 1$$

(a) (3 points) Write an Ocaml data type `token` for the tokens that a lexer would generate as input to a parser for this grammar.

Solution:

```
type token = PLUS | NEG | ZERO | ONE
```

(b) (6 points) Write an Ocaml data type `term` to parse tree generated by $< term >$.

Solution:

```
type term = Plus of term * term | Neg of term | Zero | One
```

(c) (15 points) Consider the following grammar:

$$< term >::= \texttt{+} < term >< term > \mid \tilde{} < term > \mid 0 \mid 1$$

Using the types you gave in parts a) and b), write an Ocaml recursive descent parser `parse: token list -> term` that, given a list of tokens, returns `term` representing a $< term >$ parse tree. You should use `raise (Failure no parse)` for cases where no parse exists.

Solution:

```
let rec term tokens =
    match tokens
     with PLUS :: tokens_after_PLUS ->
      (match term tokens_after_PLUS
        with (term1, tokens_after_term1) ->
         (match term tokens_after_term1
           with (term2, tokens_after_term2) ->
            (Plus (term1, term2),tokens_after_term2)))
     | NEG :: tokens_after_NEG ->
       (match term tokens_after_NEG
         with (term, tokens_after_term) ->
         (Neg term, tokens_after_term))
     | ZERO :: toks -> (Zero, toks)
     | ONE :: toks -> (One, toks)
     | [] -> raise (Failure "no parse")

let parse tokens =
     match term tokens
      with (term, []) -> term
      | _ -> raise (Failure "no parse")
```

Problem 7. (12 points)

Given the following grammar over nonterminal `<m>`, `<e>` and `<t>`, and terminals `z`, `o`, `l`, `r`, `p` and `eof`, with start symbol `<m>`:

$P0: \ <\texttt{m}>::=<\texttt{e}>\texttt{eof}$
$P1: \ <\texttt{e}>::=<\texttt{t}>$
$P2: \ <\texttt{e}>::=<\texttt{t}>\texttt{p}<\texttt{e}>$
$P3: \ <\texttt{t}>::=\texttt{z}$
$P4: \ <\texttt{t}>::=\texttt{o}$
$P5: \ <\texttt{t}>::=\texttt{l}<\texttt{e}>\texttt{r}$

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

	Action							**Goto**		
State	`z`	`o`	`l`	`r`	`p`	`[eof]`		`<m>`	`<e>`	`<t>`
st1	**s3**	**s4**	**s5**	**err**	**err**	**err**			**st2**	**st7**
st2	**err**	**err**	**err**	**err**	**err**	**a**				
st3	**r3**	**r3**	**r3**	**r3**	**r3**	**r3**				
st4	**r4**	**r4**	**r4**	**r4**	**r4**	**r4**				
st5	**s3**	**s4**	**s5**	**err**	**err**	**err**			**st8**	**st7**
st6	**err**	**err**	**err**	**err**	**err**	**a**				
st7	**err**	**err**	**err**	**r1**	**s9**	**r1**				
st8	**err**	**err**	**err**	**s10**	**err**	**err**				
st9	**s3**	**s4**	**s5**	**err**	**err**	**err**			**st11**	**st7**
st10	**r5**	**r5**	**r5**	**r5**	**r5**	**r5**				
st11	**r2**	**r2**	**r2**	**r2**	**r2**	**r2**				

where **st***i* is state *i*, **s***i* abbreviates **shift** *i*, **r***i* abbreviates **reduce** *i*, **a** abbreviates **accept** and `[eof]`] means we have reached the end of input, describe how the string `lzpor[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 5 cells in the first two rows to get you started. Thereafter, there are more blank lines than you should need to fill in the rest.

Solution:

Stack	**Current String**	**Action to be taken**
Empty	`lzpor[eof]`	Initialize stack, go to state 1
st1	`lzpor[eof]`	shift `l`, go to state 5
st1::l::st5	`zpor[eof]`	shift `z`, go to state 3
st1::l::st5::z::st3	`por[eof]`	reduce by rule 3, go to state 7
st1::l::st5::`<t>`::st7	`por[eof]`	shift `p`, go to state 9
st1::l::st5::`<t>`::st7::p::st9	`or[eof]`	shift `o`, go to state 4
st1::l::st5::`<t>`::st7::p::st9::o::st4	`r[eof]`	reduce by rule 4, go to state 7
st1::l::st5::`<t>`::st7::p::st9::`<t>`::st7:	`r[eof]`	reduce by rule 1, go to state 11
st1::l::st5::`<t>`::st7::p::st9::`<e>`::st11	`r[eof]`	reduce by rule 2, go to state 8
st1::l::st5::`<e>`::st8	`r[eof]`	shift `r`, go to state 10
st1::l::st5::`<e>`::st8::r::st10	`[eof]`	reduce by rule 5, go to state 7
st1::`<t>`::st7	`[eof]`	reduce by rule 1, go to state 2
st1::`<e>`::st2	`[eof]`	accept

8. (10 points (bonus)) Disambiguate the following grammar with start symbol < *comm* >:

```
< comm >::=  < var > <= < exp >
             |if < bool > then < comm > else < comm >
             |if < bool > then < comm >
< exp >::=   < var > | 0 | 1 | inc < exp > | dec < exp >
< var >::=   x | y | z
< bool >::=  true | false | < var > = < var >
```

Solution:

```
< comm >::=              |if < bool > then < comm >
                         | < no_missing_else >
< no_missing_else >::=   < var > <= < exp >
                         |if < bool > then < no_missing_else > else < comm >
< exp >::=               < var > | 0 | 1 | inc < exp > | dec < exp >
< var >::=               x | y | z
< bool >::=              true | false | < var > = < var >
```

Workspace