# CS421 Fall 2009 Midterm 2

Thursday, November 5, 2009

| | |
|---|---|
| Name: | |
| NetID: | |

- You have **75 minutes** to complete this exam.
- This is a **closed-book** exam.. You are allowed one 3inch by 5 inch card of notes prepared by yourself. This card is **not to be shared**. All other materials, besides pens, pencils and erasers, are to be away.
- 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, 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 18 pages to the exam. Please verify that you have all 18 pages.
- Please write your name and NetID in the spaces above, and also at the top of every page.

| Problem | Possible Points | Points Earned |
|---|---|---|
| 1 | 18 | |
| 2 | 15 | |
| 3 | 12 | |
| 4 | 12 | |
| 5 | 10 | |
| 6 | 20 | |
| 7 | 13 | |
| PreTotal | 100 | |
| Extra Credit | 8 | |
| PostTotal | 108 | |

CS 421 Midterm 2 **Name:**_____

**1.** (18 points) Give a most general unifier for the following unification problem. Capital letters **A,B,C,D,E** denote variables of unification and lower case letters **f, p** and **s** denote constructors. Show all your work by listing the operation being performed at each step, and all the results of that operation. Your final result must be expressed as a single simultaneous substitution, not a composition of individual substitutions.

**{ p(f(A,C),E) = p(f(B,D), f(A,D)); s(E) = s(f(A,B)) }**

**Note:** The notation f o g will refer to the function that first does g and then applies f to the result. f o g (x) = f (g (x)),.

**Solution:**

Let $\varphi$ = Unify { p(f(A,C),E) = p(f(B,D), f(A,D)); s(E) = s(f(A,B)) }. Then

$\varphi$ = Unify { f(A,C) = f(B,D); E = f(A,D); s(E) = s(f(A,B)) } by Decompose

= Unify { A = B; C = D; E = f(A,D); s(E) = s(f(A,B)) } by Decompose

= Unify { A = B; C = D; E = f(A,D); E = f(A,B) } by Decompose

= {A → $\varphi_1$(B)} o $\varphi_1$ by Elminate (A)

where

$\varphi_1$ = Unify { C = D; E = f(B,D); E = f(B,B) }

= {C → $\varphi_2$(D)} o $\varphi_2$ by Elminate (C)

where

$\varphi_2$ = Unify { E = f(B,D); E = f(B,B) }

= {E → $\varphi_3$(f(B,D))} o $\varphi_3$ by Elminate (E)

where

$\varphi_3$ = Unify { f(B,B) = f(B,D) }

= Unify { B = B; B = D} by Decompose

= Unify { B = D} by Delete

= {B → $\varphi_4$(D)} o $\varphi_4$ by Eliminate (B)

where $\varphi_4$ = Unify { } = identity substitution.

Therefore,

$\varphi_3$ = {B → D}. $\varphi_2$ = {E → f(D,D); B → D}, $\varphi_1$ = { C → D; E → f(D,D); B → D } and

$\varphi$ = { A → D; C → D; E → f(D,D); B → D }

CS 421 Midterm 2 **Name:**_____

2. (15 pts total) Adding to the code fragment given below, give an implementation of the following rule for application:

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2}{\Gamma \vdash e_1\ e_2 : \tau \mid \{\tau_1 = \tau_2 \rightarrow \tau\} \cup C_1 \cup C_2}$$

The (slightly abbreviated) types for object language constructs are as follows:

```
type constTy = {name : string; arity : int}
type expType = TyVar of int | TyConst of (constTy * expType list)
type env = (string * expType) list
type judgment = { gamma:env; exp:exp; expType:expType }
type proof = {antecedents : proof list; conclusion : judgment}
type exp = … | ConstExp of const | … | AppExp of exp * exp | …
```

The type of **gather_ty_constraints is**

**gather_ty_constraints** : judgment -> (proof * (expType * expType) list) option

You may use **fresh()** to generate new numbers for fresh type variables, and
**mk_fun_ty: expType -> expType -> expType**
to make the type of functions from one type to another.

```
let rec gather_ty_constraints judgment =
   let {gamma = gamma; exp = exp; expType = tau} = judgment in
   match exp
   with ConstExp c ->
    let c_ty = const_signature c in
      Some ({antecedents = []; conclusion = judgment}, [(tau, c_ty)])
   | AppExp (e1, e2) ->
(* You add clauses here *)
```

**Solution:**

```
| AppExp (e1, e2) ->
     let (tau1, tau2) = (fresh(),fresh()) in
     (match
       (gather_ty_constraints {gamma = gamma; exp = e1; expType = tau1},
        gather_ty_constraints {gamma = gamma; exp = e2; expType = tau2})
      with (Some(e1_pf, e1_const), Some(e2_pf, e2_const))
         -> Some({antecedents = [e1_pf; e2_pf]; conclusion = judgment},
               (tau1, mk_fun_ty tau2 tau) :: (e1_const @ e2_const))
     | _ -> None)
```

I accidentally referred to tau as expType in rgw ConstExp clauses, and I have corrected that here.

CS 421 Midterm 2 **Name:**_____

**More space for Problem 2.**

CS 421 Midterm 2 **Name:**_____

**3.** (12 pts total) Give a regular expression and a regular grammar generating each of the following languages over the alphabet {0, 1}. You should use the notation given in class: Regular expressions over an alphabet Σ are strings over Σ together with the five extra characters **(**, **)**, *****, **∨**, and **ε**. **No other symbols should occur in your regular expression, and they will not be accepted**.

a. (6 pts) The set of all strings ending in **1** in which all 0's are consecutive, and there is at least one **0**..

**Solution:** 1* 00*1*1

<S> ::= 1<S> | 0<T>
<T> ::= 0<T> | 1<U>
<U> ::= ε | 1<U>

b. (6 pts) The set of all strings with an even number of 1's.

**Soultion:** (0*10*1)*0*

<S> ::= ε | 0<S> | 1<N>
<N> ::= 0<N> | 1<S>

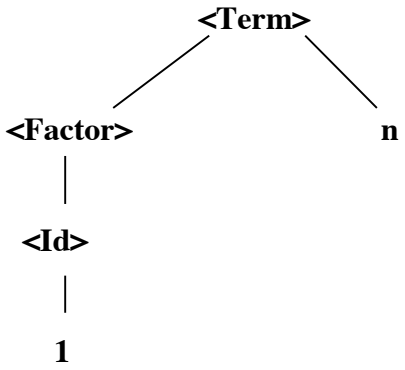
CS 421 Midterm 2 **Name:**_____

**4.** (12 points) Given the following BNF grammar, for each of the following strings, give a parse tree for it if it parses starting with **<Term>,** or write "None exists" if it does not parse starting with **<Term>.**

**<Term> ::= p <Exp> | <Factor> n**
**<Exp> ::= <Id> % <Factor> | <Factor>**
**<Factor> ::= <Term> | <Id>**
**<Id> ::= 0 | 1**

a. (2 pts) **1 n**

**Solution:**



CS 421 Midterm 2
**Name:**_____

**4.** (cont) (12 pts total) Given the following BNF grammar, for each of the following strings, give a parse tree for it if it parses starting with **<Term>,** or write "None exists" if it does not parse starting with **<Term>.**

**<Term> ::= p <Exp> | <Factor> n**
**<Exp> ::= <Id> % <Factor> | <Factor>**
**<Factor> ::= <Term> | <Id>**
**<Id> ::= 0 | 1**

b. (5 pts) **p 0 % 1 n 0 1 n n**

**Solution:** None exists

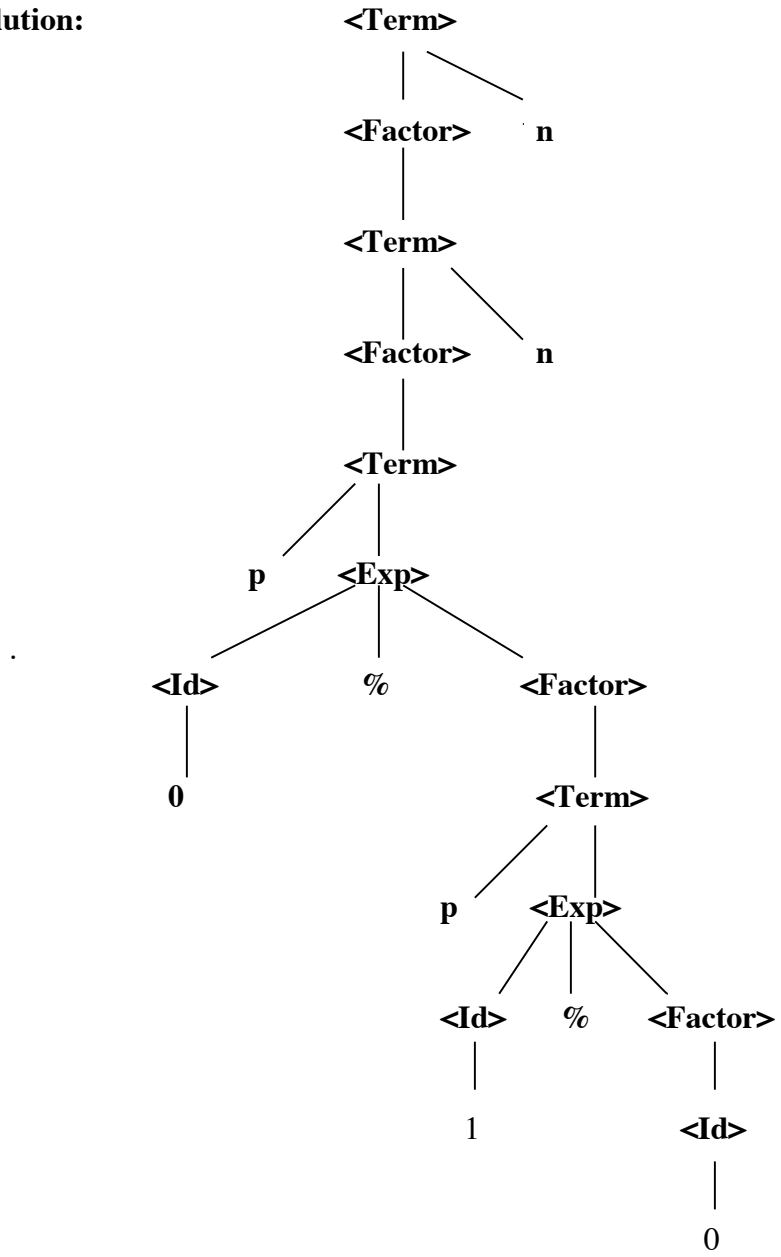
CS 421 Midterm 2 **Name:**_____

**4.** (cont) (12 pts total) Given the following BNF grammar, for each of the following strings, give a parse tree for it if it parses starting with **<Term>,** or write "None exists" if it does not parse starting with **<Term>.**

**<Term> ::= p <Exp> | <Factor> n**
**<Exp> ::= <Id> % <Factor> | <Factor>**
**<Factor> ::= <Term> | <Id>**
**<Id> ::= 0 | 1**

c. (5 pts) **p 0 % p 1 % 0 n n**

**Solution:**



CS 421 Midterm 2 **Name:**_____

**5.** (20 points) Consider the following grammar:

**<expr> ::= <atom> | p <atom> <expr>**

**<atom> ::= 0 | 1 | (<expr>)**

**a.** (4 points) Write an Ocaml data type **token** for the tokens that lexer would generate as input to a parser for this grammar.

**Solution:** type token = Ztk | Otk | Ptk | LPtk |RPtk ;;

b. (5 points) Write an Ocaml data types **expr** and **atom** to represent parse trees for each of the syntactic categories in the given grammar.

**Solution:**

type expr = AtomExp of atom | PExp of (atom * expr)
and atom = ZeroAt | OneAt | ParenAt of expr ;;

CS 421 Midterm 2 **Name:**_____

**6.** (cont) Consider the following grammar:

**<expr> ::= <atom> | p <atom> <expr>**
**<atom> ::= 0 | 1 | (<expr>)**

**c. (11 points)**Using the types you gave in parts **a.** and **b.**, write an Ocaml recursive descent parser **parse: token list -> expr** that, given a list of **token**s, returns an **expr** representing an **<expr>** parse tree. You should use

**raise (Failure "no parse")**

for cases where no parse exists.

**Solution:**

```
let rec expr tks =
  (match tks
    with Ptk::tks_after_p ->
     (match atom tks_after_p with (atom_pt, tks_after_atom) ->
       (match expr tks_after_atom with (expr_pt, tks_after_expr) ->
         (PExp(atom_pt, expr_pt), tks_after_expr)))
   | _ ->
     (match atom tks with (atom_pt, tks_after_atom) ->
       (AtomExp(atom_pt), tks_after_atom))

and atom tks =
   match tks
    with Ztk::tks_after_0 -> (ZeroAt, tks_after_0)
   | Otk::tks_after_1 -> (OneAt, tks_after_1)
   | LPtk::tks_after_lp ->
     (match expr tks_after_lp with (expr_pt, tks_after_expr) ->
       (match tks_after_expr with RPtk::tks_after_rp ->
         (ParenAt(expr_pt), tks_after_rp)
        | _ -> raise (Failure "no parse")))
   | _ -> raise (Failure "no parse");;

let parse tks = match expr tks with (expr_pt, []) -> expr_pr
                                  | _ -> raise (Failure "no parse");;
```

CS 421 Midterm 2 **Name:**_____

6. (13 points) Given the following grammar over nonterminal **<m>**, **<e>** and **<term>**, and terminals **z, o, l, r, p** and **eof**, with start symbol **<m>**:

P0: **<m> ::= <e> eof**
P1: **<e> ::= <t>**
P2: **<e>::= <t> p <e>**
P3: **<t>::= z**
P4: **<t>::= o**
P5: **<t>::= l <e> r**

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

| State | Action | | | | | | | Goto | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **z** | **o** | **l** | **r** | **p** | **[eof]** | | **<m>** | **<e>** | **<t>** |
| st1 | **s 3** | **s 4** | **s 5** | **err** | **err** | **err** | | | **st2** | **st7** |
| **st2** | **err** | **err** | **err** | **err** | **err** | **a** | | | | |
| **st3** | **r 3** | **r 3** | **r 3** | **r 3** | **r 3** | **r 3** | | | | |
| **st4** | **r 4** | **r 4** | **r 4** | **r 4** | **r 4** | **r 4** | | | | |
| **st5** | **s 3** | **s 4** | **s 5** | **err** | **err** | **err** | | | **st8** | **st7** |
| **st6** | **err** | **err** | **err** | **err** | **err** | **a** | | | | |
| **st7** | **err** | **err** | **err** | **r 1** | **s 9** | **r 1** | | | | |
| **st8** | **err** | **err** | **err** | **s 10** | **err** | **err** | | | | |
| **st9** | **s 3** | **s 4** | **s 5** | **err** | **err** | **err** | | | **st11** | **st7** |
| **st10** | **r 5** | **r 5** | **r 5** | **r 5** | **r 5** | **r 5** | | | | |
| **st11** | **r 2** | **r 2** | **r 2** | **r 2** | **r 2** | **r 2** | | | | |

where st*i* is state *i*, **s *i*** means **shift *i***, **r *i*** means **reduce *i*,** **a** means **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 to get started. **Caution:** There are strictly more rows than you will need, so do not expect to fill them all.

CS 421 Midterm 2 **Name:**_____

| Stack | Current String | Action |
|---|---|---|
| *Empty* | **lzpor[eof]** | Initialize stack, go to state 1 |
| st1 | **lzpor[eof]** | Shift and go to state 5 |
| st1 :: l :: st5 | **zpor[eof]** | Shift and go to state 3 |
| st1 :: l :: st5 :: z :: st3 | **por[eof]** | Reduce by P3, go to state 7 |
| st1 :: l :: st5 :: <t>:: st7 | **por[eof]** | Shift and go to state 9 |
| st1 :: l :: st5 :: <t>:: st7 :: p :: st9 | **or[eof]** | Shift and go to state 4 |
| st1 :: l :: st5 :: <t>:: st7 :: p :: st9 :: o :: st4 | **r[eof]** | Reduce by P4, go to state 7 |
| st1 :: l :: st5 :: <t>:: st7 :: p :: st9 :: <t> :: st7 | **r[eof]** | Reduce by P1, go to state 11 |
| st1 :: l :: st5 :: <t>:: st7 :: p :: st9 :: <e> :: st11 | **r[eof]** | Reduce by P2, go to state 8 |
| st1 :: l :: st5 :: <e>:: st8 | **r[eof]** | Shift and go to state 10 |
| st1 :: l :: st5 :: <e>:: st8 :: r :: st10 | **[eof]** | Reduce by P5, go to state 7 |
| st1 :: <t> :: st7 | **[eof]** | Reduce by P1, go to state 2 |
| st1 :: <e> :: st2 | **[eof]** | accept |
| | | |
| | | |

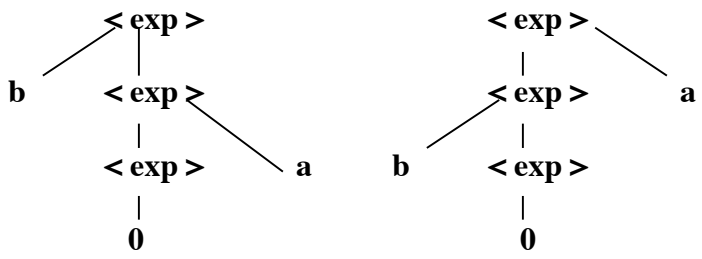
CS 421 Midterm 2 **Name:**_____

7.(10 pts) Consider the following extended BNF grammar over the alphabet of tokens **0,1,a,b,m**

**< exp > ::= 0 | 1| b <exp> | <exp> a | <exp> m <exp>**

a. (3 pts) Show that the given grammar is ambiguous.

**Solution: b0a** has two parses:




b. (7 pts) Write a new grammar generating the same language as the one given above, but that is unambiguous, and such that **a** has higher precedence than **b,** which in turn has higher precedence than **m**, and such that **m** associates to the left.

**Solution:**

**<exp> ::= <exp> m <nom> | <nom>**
**<nom> ::= b <nom> | <nob>**
**<nob> ::= 0 | 1 | <nob> a**

CS 421 Midterm 2 **Name:**_____

8. (Extra Credit) (8 pts total) The following grammar is ambiguous:

<exp> ::= true | false | if <exp> then <exp> | if <exp> then <exp> else <exp>

Given an unambiguous grammar generating the same language, such that **if_then_** has higher precedence than **if_then_else_**.

**Solution:**

This problem was somewhat ill-worded. The conflict is between the outer if_then / if_then_else and one occurring in the then clause. The conflict is intended to be resolved by giving the else to the inner if_then and leaving the outer one without an else.

**<exp> ::= <unmatched_final_then> | <matched_final_then>**

**<unmatched_final_then> ::= if <exp> then <exp>**

**| if <exp> then <matched_final_then> else <unmatched_final_then>**

**<matched_final_then> ::=**

**if <exp> then <matched_final_then> else <matched_final_then> | true | false**

If you seemed to be trying to do something that showed the right basic ideas we tried to give majority credit.

CS 421 Midterm 2 **Name:**_____

CS 421 Midterm 2 **Name:**_____

**Rules for type derivations:**

Constants:

$$\frac{}{\Gamma \vdash n : \text{int}} \quad \text{(assuming } n \text{ is an integer constant)}$$

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}} \qquad \frac{}{\Gamma \vdash \text{false} : \text{bool}}$$

Variables:

$$\frac{}{\Gamma \vdash x : \sigma} \quad \text{if } \Gamma(x) = \sigma$$

Primitive operators ( $\oplus \in \{ +, -, *, \ldots \}$):

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \oplus e_2 : \text{int}}$$

Relations ( $\sim \in \{ <, >, =, <=, >= \}$):

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \sim e_2 : \text{bool}}$$

Connectives :

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \,\&\&\, e_2 : \text{bool}} \qquad \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \,||\, e_2 : \text{bool}}$$

If_then_else rule:

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : \tau}$$

Application rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1\ e_2) : \tau_2}$$

fun rule:

$$\frac{[x : \tau_1] \cup \Gamma \vdash e : \tau_2}{\Gamma \vdash \text{fun } x \text{ -> } e : \tau_1 \to \tau_2}$$

let rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad [x : \tau_1] \cup \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\text{let } x = e_1 \text{ in } e_2) : \tau_2}$$

let rec rule:

$$\frac{[x : \tau_1] \cup \Gamma \vdash e_1 : \tau_1 \quad [x : \tau_1] \cup \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\text{let rec } x = e_1 \text{ in } e_2) : \tau_2}$$