# Solutions for Sample Questions for Midterm 2 (CS 421 Fall 2013)

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. Given a polymorphic type derivation for
   {} |- let pair = fun x -> (x, x) in pair(pair 3) : ((int * int) * (int * int))

**Solution:**

Let $\Gamma_1$ = {x : ‘a}. $\Gamma_2$ = {pair : ∀’a. ‘a -> ‘a * ‘a}.
The infixed data construct , (comma) has type ∀’a ‘b. ‘a -> ‘b -> ‘a * ‘b

Let LeftTree =

Instance: ‘a → ‘a, ‘b → ‘a

Const ______________________ Var ________

$\Gamma_1$ |- (,) : ‘a -> ’a -> ‘a * ‘a $\Gamma_1$ |- x : ‘a

App ______________________________________ Var__________

$\Gamma_1$ |- (,) x : ‘a -> ‘a * ‘a $\Gamma_1$ |- x : ‘a

App ____________________________________________________

{x : ‘a}. |- (x, x) : ‘a * ‘a

Fun __________________________

{} |- fun x -> (x,x) : ‘a -> ‘a * ‘a

Let RightTree =

Var __Instance: ‘a → int____ Const _________

$\Gamma_2$ |- pair : int -> int * int $\Gamma_2$ |- 3 : int

Var __Instance: ‘a → int * int________________ App ____________________________________

$\Gamma_2$ |- pair : int * int -> ((int * int) * (int * int)) $\Gamma_2$ |- pair(3) : int * int

App ______________________________________________________________

{pair : ∀’a. ‘a -> ‘a * ‘a}|- pair(pair 3) : ((int * int) * (int * int))

Then the full proof is

LeftTree RightTree

_____________________________________________________

{} |- let pair = fun x -> (x, x) in pair(pair 3) : ((int * int) * (int * int))

2. 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:**

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

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

→ {Y = h(y); f(Z,x) = f(Y,W)} with {X → f(g(x),W)}      by orient  (h(y) = Y)
→ {f(Z,x) = f(h(y),W)} with {X → f(g(x),W), Y → h(y)} by eliminate  (Y = h(y))
→ {Z = h(y); x=W} with {X → f(g(x),W), Y → h(y)}      by decompose  (f(Z,x) = f(h(y),W))
→ {x = W} with {X → f(g(x),W), Y → h(y), Z → h(y)}    by eliminate  (Z = h(y))
→ {W = x} with {X → f(g(x),W), Y → h(y), Z → h(y)}    by orient  (x = W)
→{} with {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. 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>**

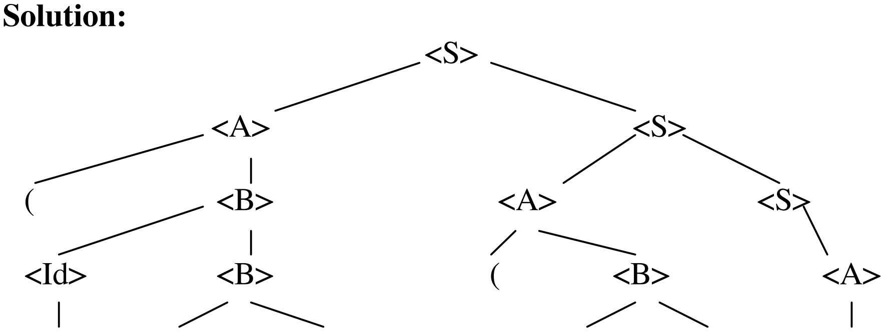
4. Consider the following grammar:

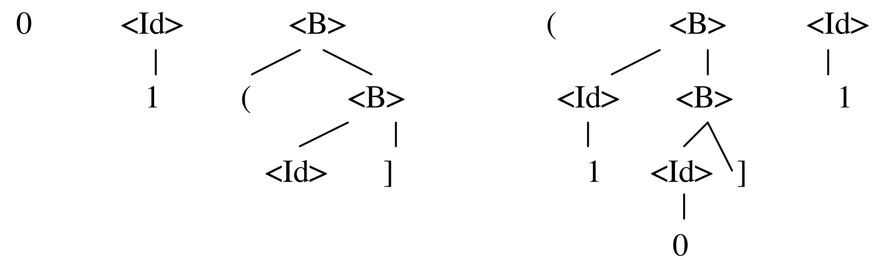
<S> ::= <A> | <A> <S>
<A> ::= <Id> | ( <B>
<B> ::=  <Id> ] |  <Id><B> | ( <B>
<Id> ::= 0 | 1

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

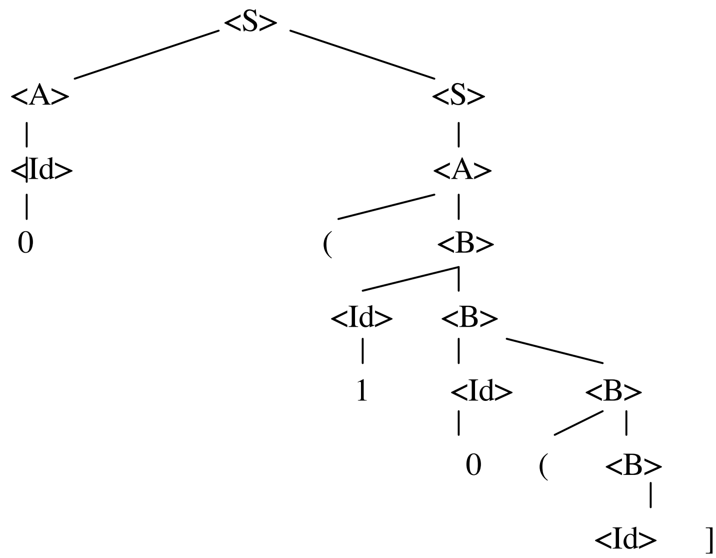
**Solution:**






b. 0 ( 1 0 ( 1 ]

**Solution:**




c. ( 0 ( 1 0 1] 0 ]

**Solution:** No parse tree

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

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

**Solution:** String: bab

6. 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**

7. Write a recursive descent parser for the following grammar:,
<S> ::= <N> % <S> | <N>
<N> ::= g <N> | a | b
You should include a datatype **token** of tokens input into the parser, one or more datatypes representing the parse trees produced by parsing (the abstract syntax trees), and the function(s) to produce the abstract syntax trees. Your parser should take a list of tokens as input and generate an abstract syntax tree corresponding to the parse of the input token list.
**Solution:**

```
type  token = ATk | BTk | GTk | PercentTk
type  s = Percent of (n * s) | N_as_s  n
and n = G of n | A | B

let rec s_parse tokens  =
   match n_parse tokens with (n, tokens_after_n) ->
       (match tokens_after_n with PercentTk::tokens_after_percent ->
            (match s_parse tokens_after_percent
             with (s, tokens_after_s) -> (Percent (n,s), tokens_after_s))
          | _ -> (N_as_s n, tokens_after_n))
and n_parse tokens =
   match tokens
   with GTk::tokens_after_g ->
       (match n_parse tokens_after_g
           with (n, tokens_after_n) -> (G n, tokens_after_n))
     | ATk::tokens_after_a -> (A, tokens_after_a)
     | BTk::tokens_after_b -> (B, tokens_after_b)

let parse tokens =
   match s_parse tokens
   with (s, []) -> s
     | _ -> raise (Failure "No parse")
```

8. Why don't we ever get shift/shift conflicts in LR parsing?
**Solution: The shift action means, when in a given state prescribing the shift, to remove the token from the top of the token stream and place it on top of the stack and move to the new state prescribed for the given state and the moved token. There is only one token stream, only one stack and the state to which to go is entirely determined by the given**

**state and the token moved. Thus, there is only one way to execute a shift so we never have two different shifts between which to choose.**

9. Consider the following grammar with terminals ***, f, x,** and **y**, and **eol** for “end of line”, and non-termimals **S, E** and **N** and productions

```
(P0)  S => E eol
(P1)  E => E * N
(P2)  E => N
(P3)  N => f N
(P4)  N => x
(P5)  N => y
```

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

| | **ACTION** | | | | | **GOTO** | | |
|---|---|---|---|---|---|---|---|---|
| **STATE** | ***** | **f** | **x** | **y** | **eol** | **S** | **E** | **N** |
| **1** | | s3 | s4 | s5 | | 2 | 6 | 7 |
| **2** | | | | | acc | | | |
| **3** | | s3 | s4 | s5 | | | | 8 |
| **4** | r4 | r4 | r4 | r4 | r4 | | | |
| **5** | r5 | r5 | r5 | r5 | r5 | | | |
| **6** | s9 | | | | acc | | | |
| **7** | r2 | r2 | r2 | r2 | r2 | | | |
| **8** | r3 | r3 | r3 | r3 | r3 | | | |
| **9** | | s3 | s4 | s5 | | | | 10 |
| **10** | r1 | r1 | r1 | r1 | r1 | | | |

where *si* is shift and stack state *i*, *rj* is reduce using production *Pj*, *acc* is accept. The blank cells should be considered as labeled with error. The empty “character” represents end of input. Describe how the sentence **fx*y<eol>** would be parsed with an LR parser using this table. For each step of the process give the parser action (shift/reduce), input and stack state.

**Solution: In the table below, the top of the stack is on the right**

| **Curr State** | **Current Stack** : | **Curr String** | **Action** |
|---|---|---|---|
| | | fx*y<eol> | Init stack and go to state 1 |
| st1 | st1 | fx* y<eol> | Shift f to stack, go to state 3 |
| st3 | st1 : f : st3 | x*y<eol> | Shift x, go to state 4 |
| st4 | st1 : f : st3 : x : st4 | *y<eol> | Reduce by prod 4: N => x, ie remove st4 and x from the stack, temporarily putting us in st3, push N and st8 (because GOTO(st3, N) = st8 onto stack go to state 8 |
| st8 | st1 : f : st3 : N : st8 | *y<eol> | Reduce by prod 3: N => f N, go |

| | | | |
|---|---|---|---|
| | | | to state 7 |
| st7 | st1 : N : st7 | *y<eol> | Reduce by prod 2: E=>N, go to state 6 |
| st6 | st1 : E : st6 | *y<eol> | Shift *, go to state 9 |
| st9 | st1 : E : st7 : * : st9 | y<eol> | Shift y, go to state 5 |
| st5 | st1 : E : st7 : * : st9 : y st5 | <eol> | Reduce by prod 5: N=>y, go to state 10 |
| st10 | st1 : E : st7 : * : st9 : N : st10 | <eol> | Reduce by prod 1: E=>E*N, go to state 6 |
| st6 | st1 : E : st6 | <eol> | Accept (prod 0: S => E<eol>) |