HW 8 – Parse Trees, Ambiguous Grammars, LR and Recursive Descent Parsing

CS 421 – Fall 2012
Revision 1.0

Assigned Tuesday, October 30, 2012
Due Tuesday, November 6, 2012, 11:59 PM
Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

2 Turn-In Procedure

Answer the problems below, save your work as a PDF (either scanned if handwritten or converted from a program), and hand in the PDF. Your file should be named `hw8.pdf`.

3 Objectives and Background

The purpose of this HW is to test your understanding of

- How to create a parse tree for a given string with a given grammar
- How to disambiguate a grammar
- How to write a recursive descent parser for an LL(1) grammar

Another purpose of HW8 is to provide you with experience answering non-programming written questions of the kind you may experience on the second midterm and final.
Caution: It is strongly advised that you know how to do these problems before the second midterm.

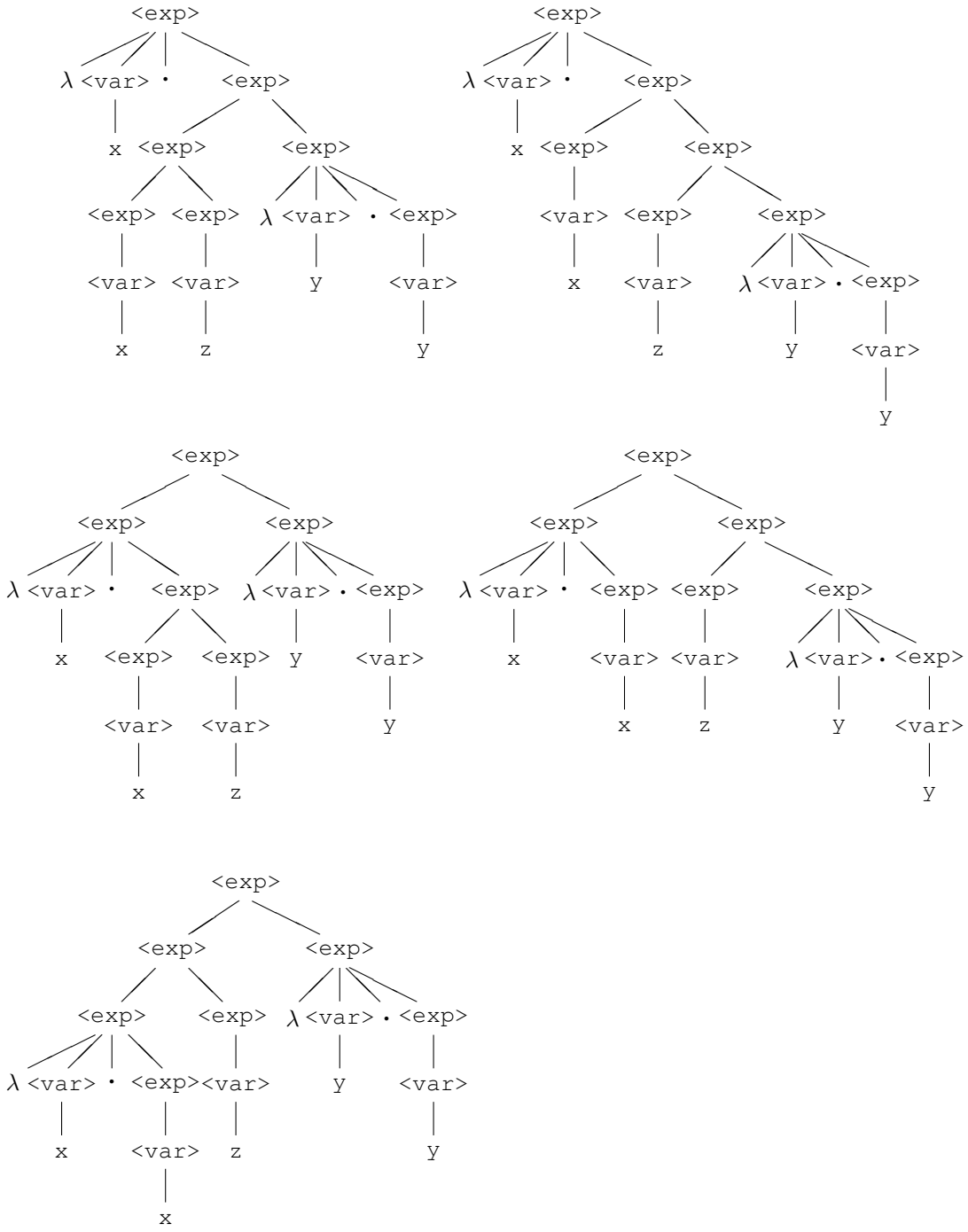
4 Problems

1. (23 points) Consider the following grammar over the alphabet $\{\lambda,\ .,\ x,\ y,\ z,\ (,\ )\}$:

$$\begin{array}{lcl} < \texttt{exp} > & ::= & < \texttt{var} > \mid \lambda\ < \texttt{var} > \ .\ < \texttt{exp} > \mid < \texttt{exp} >< \texttt{exp} > \mid (< \texttt{exp} >) \\ < \texttt{var} > & ::= & \texttt{x} \mid \texttt{y} \mid \texttt{z} \end{array}$$

   a. (9 points) Show that the above grammar is ambiguous by showing at least three distinct parse trees for the string λx.x z λy.y
   Solution: There are five possible parse trees for λx.x z λy.y with the given grammar. Yours should be three

among them.



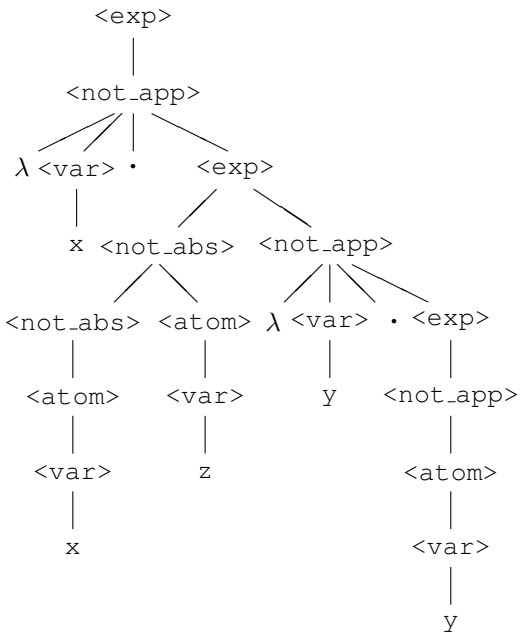
b. (9 points) Write a new grammar accepting the same language that is unambiguous, and such that application $< \texttt{exp} > < \texttt{exp} >$ has higher precedence than abstraction $\lambda < \texttt{var} > . < \texttt{exp} >$, and such that application associates to the left.

Solution:

$$
\begin{array}{rcl}
< \texttt{exp} > & ::= & < \texttt{not\_abs} >< \texttt{not\_app} > \mid < \texttt{not\_app} > \\
< \texttt{not\_abs} > & ::= & < \texttt{not\_abs} >< \texttt{atom} > \mid < \texttt{atom} > \\
< \texttt{not\_app} > & ::= & \lambda < \texttt{var} > \texttt{.} < \texttt{exp} > \mid < \texttt{atom} > \\
< \texttt{atom} > & ::= & < \texttt{var} > \mid \texttt{(} < \texttt{exp} > \texttt{)} \\
< \texttt{var} > & ::= & \texttt{x} \mid \texttt{y} \mid \texttt{z}
\end{array}
$$

c. (5 points) Give the parse tree for λx.x z λy.y using the grammar you gave in the previous part of this problem.

Solution:



2. (17 points) Given the following grammar over nonterminal <m>, <e> and <t>, and terminals z, o, l, r, p and eof, with start symbol <m>:

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

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	**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. You will need to add more rows.

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