Problem Set 2

Fall 23

Due: Sunday, September 17th.

1. LL Parsing

   For the following sub-problems, consider the following context-free grammar:

$$
\begin{aligned}
S &\rightarrow T\$ && (1)\\
T &\rightarrow aAa && (2)\\
T &\rightarrow bBb && (3)\\
A &\rightarrow T && (4)\\
A &\rightarrow d && (5)\\
B &\rightarrow T && (6)\\
B &\rightarrow BdB && (7)
\end{aligned}
$$

   (a) What are the terminals and non-terminals of this language?

   (b) Show the derivation of the string *badadbadabb*\$ starting from S (specify which production you used at each step), and give the parse tree according to that derivation.

   (c) Give the first and follow sets for each of the non-terminals of the grammar.

   (d) What are the predict sets for each production?

   (e) Give the parse table for the grammar. The parse table is just a different representation of the predict set. Each non-terminal gets a row, and each terminal gets a column. Each entry is either production(s) or an error. Following the idea of recursive descent parsing, if a production p is in the cell for non-terminal V_n and terminal V_t, it means that in the function for matching V_n, if the first token is V_t, the matching should follow production p.

   (f) Is this an LL(1) grammar? Why or why not?

   (g) Suppose we change rule 5 to $A \rightarrow \lambda$. Is the resulting grammar LL(1)? Why or why not?

   (h) Is the grammar LL(k) for any k?

Solution:

(a) The non-terminals are any of the symbols that appear on the left hand side of productions: S, T, A, B. The terminals are any of the symbols that appear in completely-rewritten strings (i.e., strings with no non-terminals in them). These are all of the symbols that don't appear on the left hand side, except for λ, which is not a symbol that appears in strings: a, b, d, \$.

(b) Here is the derivation. I will use a left derivation for this (we will rewrite starting from the left-most non-terminal):

$$
\begin{aligned}
S &\Rightarrow T\$ && \textit{using } S \to T\$ \\
&\Rightarrow bBb\$ && \textit{using } T \to bBb \\
&\Rightarrow bBdBb\$ && \textit{using } B \to BdB \\
&\Rightarrow bTdBb\$ && \textit{using } B \to T \\
&\Rightarrow baAadBb\$ && \textit{using } T \to aAa \\
&\Rightarrow badadBb\$ && \textit{using } T \to d \\
&\Rightarrow badadTb\$ && \textit{using } B \to T \\
&\Rightarrow badadbBbb\$ && \textit{using } T \to BbB \\
&\Rightarrow badadbTbb\$ && \textit{using } B \to T \\
&\Rightarrow badadbaAabb\$ && \textit{using } T \to aAa \\
&\Rightarrow badadbadabb\$ && \textit{using } A \to d
\end{aligned}
$$

(c) We begin by computing the first sets for each non-terminal. To build up the first sets, we start by examining each production to determine which constraints that production creates on the various first sets. In the first pass over each production, we get:

$$
\begin{aligned}
&First(S) \supseteq First(T) - \lambda && \text{from production 1} \\
&First(T) \supseteq First(aAa) = \{a\} && \text{from production 2} \\
&First(T) \supseteq First(bBb) = \{b\} && \text{from production 3} \\
&First(A) \supseteq First(T) && \text{from production 4} \\
&First(A) \supseteq First(d) = \{d\} && \text{from production 5} \\
&First(B) \supseteq First(T) && \text{from production 6} \\
&First(B) \supseteq First(BdB) = First(B) && \text{from production 7}
\end{aligned}
$$

Solving these constraints to find the minimum assignment to each first set (using the method we discussed in class), we get an initial "guess" for the first sets of:

$$
\begin{aligned}
First(S) &= \{a, b\} \\
First(T) &= \{a, b\} \\
First(A) &= \{a, b, d\} \\
First(B) &= \{a, b\}
\end{aligned}
$$

Because none of these first sets include λ, when we go over the productions again, we don't have to add any new constraints. (If the first set for a non-terminal

contained λ, then for any production whose right-hand-side started with that non-terminal, we would have to look at the first set of the second symbol on the RHS).

We can now compute the follow sets. Follow sets begin by looking at the non-terminals that appear on the right hand side of rules.

$Follow(T) \supseteq First(\$) = \{\$\}$	from production 1
$Follow(T) \supseteq Follow(A)$	from production 4
$Follow(T) \supseteq Follow(B)$	from production 6
$Follow(A) \supseteq First(a) = \{a\}$	from production 2
$Follow(B) \supseteq First(b) = \{b\}$	from production 3
$Follow(B) \supseteq First(d) = \{d\}$	from production 7

When we solve these set constraints, we get:

$$Follow(T) = \{a, b, d, \$\}$$
$$Follow(A) = \{a\}$$
$$Follow(B) = \{b, d\}$$

(d) We compute the predict sets for each production using the method we discussed in class:

$$Predict(S \rightarrow T\$) = \{a, b\}$$
$$Predict(T \rightarrow aAa) = \{a\}$$
$$Predict(T \rightarrow bBb) = \{b\}$$
$$Predict(A \rightarrow T) = \{a, b\}$$
$$Predict(A \rightarrow d) = \{d\}$$
$$Predict(B \rightarrow T) = \{a, b\}$$
$$Predict(B \rightarrow BdB) = \{a, b\}$$

(e) Based on the predict set above, we get the following table:

	a	b	d	$
S	1	1		
T	2	3		
A	4	4	5	
B	6,7	6,7		

(f) Because there are are conflicts in the predict table (i.e., there exists multiple productions for a given non-terminal), this grammar is not LL(1).

(g) Rebuilding the first and follow sets for the non-terminals, we get:

$$First(S) = \{a, b\}$$
$$First(T) = \{a, b\}$$
$$First(A) = \{a, b, \lambda\}$$
$$First(B) = \{a, b\}$$

$$Follow(T) = \{a, b, d, \$\}$$
$$Follow(A) = \{a\}$$
$$Follow(B) = \{b, d\}$$

Putting them together, we get the following predict sets (note that now we sometimes have to worry about follow sets!):

$$Predict(S \to T\$) = \{a, b\}$$
$$Predict(T \to aAa) = \{a\}$$
$$Predict(T \to bBb) = \{b\}$$
$$Predict(A \to T) = \{a, b\}$$
$$Predict(A \to \lambda) = \{a\}$$
$$Predict(B \to T) = \{a, b\}$$
$$Predict(B \to BdB) = \{a, b\}$$

Now we have even more prediction conflicts. If we're trying to predict what to rewrite an A into, and we see an a, we don't know whether to rewrite it into T or into λ. Therefore, the resulting grammar is not LL(1).

(h) The grammar is not LL(k) for any k. Note that the letter a's are always generated in pairs using production 1. Therefore, to determine whether to rewrite A into T or into λ, the parser needs to compare the number of a's it has read and the number of a's remaining. If there are more a's in the remaining input, A should be rewritten into T; otherwise rewritten to λ. Nonetheless, the length of the input could be arbitrarily long. For any integer k, consider the input of length $2k + 1$: $a^{2k}\$$. After reading the first k a's and rewriting A to T for k times, the parser won't be able to determine how to rewrite the next A.

2. LR Parsing (for in-person courses only)

for the following sub-problems, consider the following grammar:

$$
\begin{aligned}
S &\rightarrow aP\$ && (8)\\
E &\rightarrow aP && (9)\\
E &\rightarrow Eb && (10)\\
E &\rightarrow b && (11)\\
P &\rightarrow Ea && (12)
\end{aligned}
$$

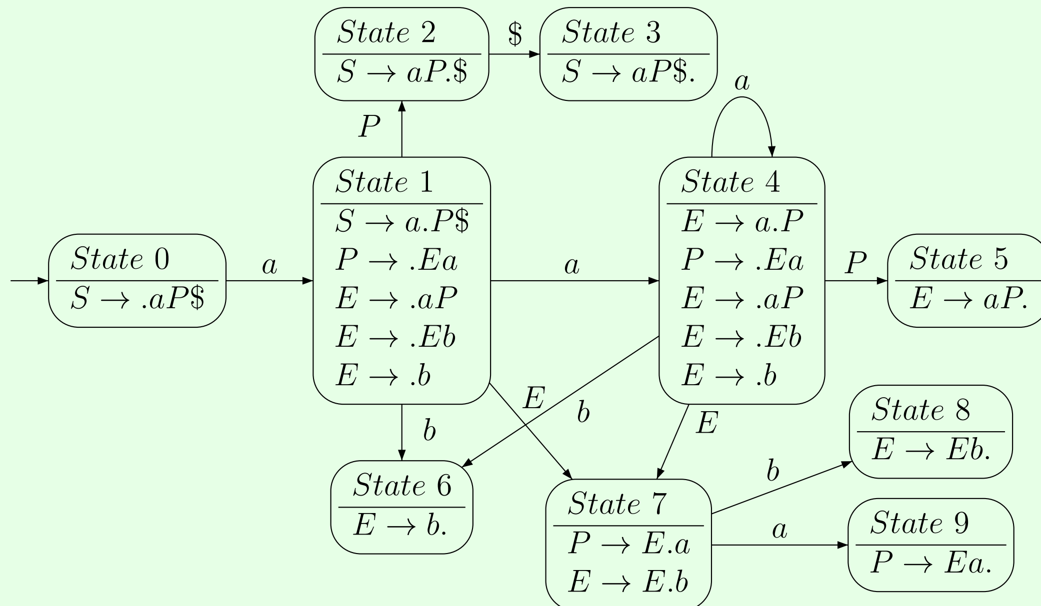
(a) Build the CFSM for this grammar.

(b) Build the goto and action tables for this grammar. Is it an LR(0) grammar? Why or why not?

(c) Show the steps taken by the parser when parsing the string: $aabaa\$$. Give the action and show the state stack and remaining input for each step of the parse. Shift actions should be of the form "Shift X" where X is the state you are shifting to, and Reduce actions should be of the form "Reduce R, goto X" where "R" is the rule being used to reduce, and "X" is the state the parser winds up in.

Solution:

(a) Here is the CFSM:



(b) Note that in this CFSM, there are no states where there are both Shift configurations and Reduce configurations. Every state either has all Shift configurations, or exactly one Reduce configuration. Hence, this machine has no Shift/Reduce or Reduce/Reduce conflicts. As a result, the grammar is an LR(0) grammar.

The action table is just the tabular representation of this machine. The goto and action tables is as follows:

State	a	b	$	E	P
0	1				
1	4	6		7	2
2			3		
3					
4	4	6		7	5
5					
6					
7	9	8			
8					
9					

State	*Action*
0	Shift
1	Shift
2	Shift
3	Accept
4	Shift
5	Reduce 9
6	Reduce 11
7	Shift
8	Reduce 10
9	Reduce 12

(c) Let's see how the following string is parsed by this machine.

$$aabaa\$$$

Recall that we parse by keeping a stack of states (starting in state 0). The state at the "top" of the stack is the state we are currently in. When we shift, we consume the next token off the input, and use the goto table to decide which table to go to; that state is pushed onto the stack. When we reduce, we "back up" as many steps as there are symbols on the right hand side of the rule we are reducing–we pop that many symbols off the stack. Then, from the state we end up in, we look at the goto table to decide which state to go to based on the symbol on the left hand side of the rule: we Reduce (according to a rule) and goto (the next state).

State stack	Remaining input	Action	Explanation
0	$aabaa\$$	Shift 1	Consume a from the input and shift
01	$abaa\$$	Shift 4	Consume a from the input and shift
014	$baa\$$	Shift 6	Consume b from the input and shift
0146	$aa\$$	Reduce 11, goto 7	Back up one (b) and replace with E
0147	$aa\$$	Shift 9	Consume a from the input and shift
01479	$a\$$	Reduce 12, goto 5	Back up two (Ea) and replace with P
0145	$a\$$	Reduce 9, goto 7	Back up one (aP) and replace with E
017	$a\$$	Shift 9	Consume a from the input and shift
0179	$	Reduce 12, goto 2	Back up two (Ea) and replace with P
012	$	Shift 3	Consume $ from the input and shift
0123		Accept	We have matched the string

3. Generating Code

For the following sub-problems, consider the following line of code:

$$y = z - 5 * 3 \qquad (1)$$

(a) Draw an AST for the above code. Assume that your AST has nodes for assignments, binary operations, variables, and literals. Don't forget the order of operations!

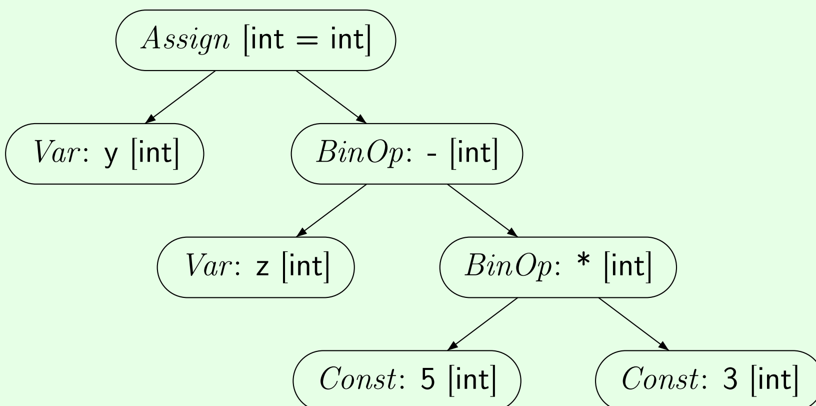
(b) Consider the following assembly instructions

Assembly	Description
$LD \quad X, \$Y$	Load the value stored in the variable X into the temporary Y
$LI \quad v, \$X$	Load the value v into the temporary X
$\$X = \$Y - \$Z$	Put the difference of temporaries Y and Z into X
$\$X = \$Y * \$Z$	Put the product of temporaries Y and Z into X
$ST \quad \$X, Y$	Store the value inside temporary X into variable Y

Use these instructions to generate assembly for the above code. You can use any letter to represent a temporary, just prefix it with $.

Solution:

(a) AST:



(b) Assembly:

1. $LI \quad 5, \$A$
2. $LI \quad 3, \$B$
3. $\$C = \$A * \$B$
4. $LD \quad z, \$D$
5. $\$E = \$D - \$C$
6. $ST \quad \$E, y$

NOTE: The order of operations for the initial *LI* statements is irrelevant, they just must both occur before the multiplication statement. Additionally, the *LD* statement's placement is also irrelevant so long as it occurs before the subtraction statement. Finally, the temporary names were chosen purely for convenience, they can be any other letters so long as the actual statements are correct.