CSEP501 : Compiler Construction: Homework 3

Karuna Sagar Krishna

November 22, 2023

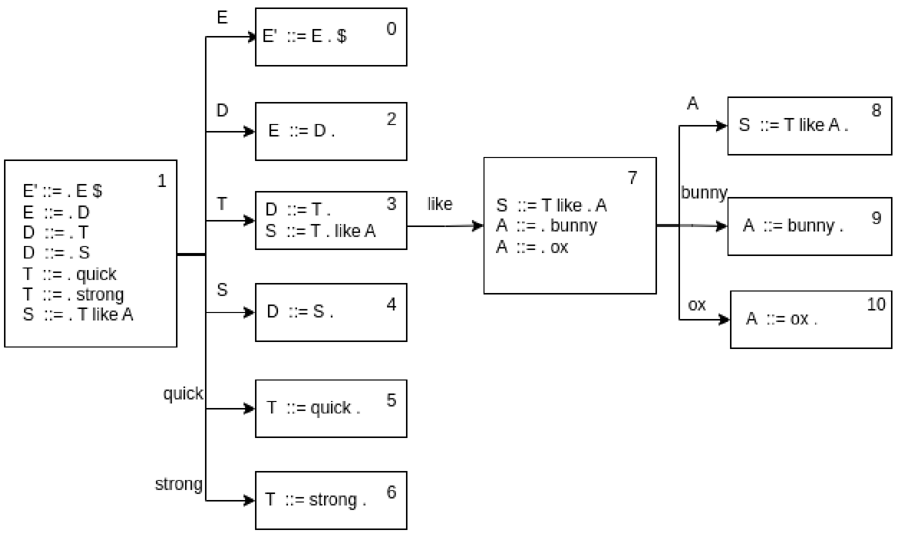
Question 1

1a

Adding production rule for E' and number all the production rules, we have:

rule #	production
0	E' ::= *E $*
1	E ::= *D*
2	D ::= *T*
3	D ::= *S*
4	S ::= *T like A*
5	T ::= *quick*
6	T ::= *strong*
7	A ::= *bunny*
8	A ::= *ox*

Using the closure, goto and LR(0) construction algorithms, we get the following DFA for viable prefixes:



The LR(0) parsing table built using the above DFA and parse table construction algorithms is as follows:

	like	quick	strong	bunny	ox	$	E	D	T	S	A
0						acc					
1		s5	s6				g0	g2	g3	g4	
2	r1	r1	r1	r1	r1	r1					
3	s7,r2	r2	r2	r2	r2	r2					
4	r3	r3	r3	r3	r3	r3					
5	r5	r5	r5	r5	r5	r5					
6	r6	r6	r6	r6	r6	r6					
7				s9	s10						g8
8	r4	r4	r4	r4	r4	r4					
9	r7	r7	r7	r7	r7	r7					
10	r8	r8	r8	r8	r8	r8					

1b

	nullable	FIRST	FOLLOW
E	no	{quick, strong}	{$}
D	no	{quick, strong}	{$}
T	no	{quick, strong}	{like, $}
S	no	{quick, strong}	{$}
A	no	{bunny, ox}	{$}

1c

	like	quick	strong	bunny	ox	\$	E	D	T	S	A
0						acc					
1		s5	s6				g0	g2	g3	g4	
2						r1					
3	s7					r2					
4						r3					
5	r5					r5					
6	r6					r6					
7				s9	s10						g8
8						r4					
9						r7					
10						r8					

1d

From the LR(0) table we can see that there is a shift reduce conflict at state 3 for terminal *like*. Because of this the grammar is not LR(0).

The grammar is SLR because we can resolve the shift reduce conflict by using the FIRST and FOLLOW sets as shown in the above SLR parse table.

Question 2

rule #	production
1	A ::= $s\ C\ n\ g$
2	A ::= ϵ
3	B ::= $C\ r$
4	B ::= t
5	C ::= $B\ i$
6	C ::= t

	nullable	FIRST	FOLLOW
A	yes	{s}	{\$}
B	no	{t}	{ı}
C	no	{t}	{r, n}

Lets consider non-terminal B which has two production rules $B ::= Cr$ and $B ::= t$. Here neither of the productions step into ϵ, so $FIRST(Cr) \cap FIRST(t)$ should be empty for the grammar to satisfy LL(1) property. Clearly, $FIRST(t) = \{t\}$ and $FIRST(C) \in FIRST(Cr)$, so we see $FIRST(C) \cap FIRST(t) \neq \emptyset$.

Similarly, for non-terminal C, $FIRST(Bi) \cap FIRST(t) = FIRST(B) \cap FIRST(t) \neq \emptyset$. So the grammar is not LL(1).

There is an indirect left recursion between B and C. So we substitute B in C production rules to make this is a direct left recursion. We can eliminate

B production rules since B is no longer referenced anywhere; note, this is not possible if we replaced C in B since it shows up in A production rules. We get the following grammar:

rule #	production
1	A ::= $s\ C\ n\ g$
2	A ::= ϵ
3	C ::= $C\ r\ i$
4	C ::= $t\ i$
5	C ::= t

Now we have direct left recursion in C, using the formal left recursion solution to eliminate it, we get:

rule #	production
1	A ::= $s\ C\ n\ g$
2	A ::= ϵ
3	C ::= $t\ i\ C'$
4	C ::= $t\ C'$
5	C' ::= $r\ i\ C'$
6	C' ::= ϵ

Here we see common prefix on RHS of C production rules. Using the left factor removal solution, we get:

rule #	production
1	A ::= $s\ C\ n\ g$
2	A ::= ϵ
3	C ::= $t\ C''$
3	C'' ::= $i\ C'$
4	C'' ::= C'
5	C' ::= $r\ i\ C'$
6	C' ::= ϵ

Recalculating the FIRST and FOLLOW sets, we get:

	nullable	FIRST	FOLLOW
A	yes	{s}	{\$}
C	no	{t}	{n}
C''	yes	{i, r}	{n}
C'	yes	{r}	{n}

Now, lets consider all non-terminals and verify the LL(1) property on them.

For non-terminal A, there are 2 production rules and since one of the production steps to ϵ, we check for $FIRST(sCng) \cap FOLLOW(A) = \emptyset$. Clearly, $FIRST(sCng) = \{s\}$, so the LL(1) property holds.

For non-terminal C, there is only one production rule hence it is deterministic. Hence LL(1) property holds trivially.

For non-terminal C'', there are 2 production rules and neither of them steps into ϵ. So the LL(1) property to verify is $FIRST(iC') \cap FIRST(C') = \emptyset$. $FIRST(iC') = \{i\}$ and $FIRST(C') = \{r\}$, so the LL(1) property holds.

For non-terminal C', there are 2 production rules and one of them steps into ϵ. So we should verify $FIRST(riC') \cap FOLLOW(C') = \emptyset$. $FIRST(riC') = \{r\}$ and $FOLLOW(C') = \{n\}$, so the LL(1) property holds.

Since the LL(1) property holds for every non-terminal, the new transformed grammar is LL(1).

Question 3

rule #	production
1	S ::= *S ; S*
2	S ::= *id := E*
3	S ::= *print (L)*
4	E ::= *id*
5	E ::= *num*
6	E ::= *E + E*
7	E ::= *(S , E)*
8	L ::= *E*
9	L ::= *L , E*

We have seen rule 6 in class, where we saw that it was part of ambiguous grammar. Specifically, it doesn't clarify the associativity of the + operator. Similarly, rule 1 has a similar structure to rule 6, so it must have the same ambiguity. This can be seen by the following 2 left most derivation leading to the same target sentence $id := num; id := num; id := num$:

$$\begin{aligned}
S &\to S; S \\
&\to S; S; S \\
&\to id := E; S; S \\
&\to id := num; S; S \\
&\to id := num; id := E; S \\
&\to id := num; id := num; S \\
&\to id := num; id := num; id := E \\
&\to id := num; id := num; id := num
\end{aligned}$$

$$
\begin{aligned}
S &\rightarrow S;S\\
&\rightarrow id := E;S\\
&\rightarrow id := num;S\\
&\rightarrow id := num;S;S\\
&\rightarrow id := num;id := E;S\\
&\rightarrow id := num;id := num;S\\
&\rightarrow id := num;id := num;id := E\\
&\rightarrow id := num;id := num;id := num
\end{aligned}
$$

Since the same target sentence/string has 2 different left most derivations, the grammar is ambiguous.

Lets assume, we want to have left associativity for ; and + operators. To achieve this we want to have only left recursion so that the parse tree leans on the left thereby honoring left associativity eg. consider $id + id + id$, with left recursion we would be forced to evaluate the left + first i.e. $(id + id) + id$. This implies a production rule that looks like $E ::= E + E$ gets transformed to $E ::= E + T|T$, similar to what we saw with classic expression grammar in class (lecture 2). Applying this to the above grammar, leads to:

rule #	production
1	S ::= *S ; S′*
2	S ::= *S′*
3	*S′* ::= *id := E*
4	*S′* ::= *print (L)*
5	E ::= *E + E′*
6	E ::= *E′*
7	*E′* ::= *id*
8	*E′* ::= *num*
9	*E′* ::= *(S , E)*
10	L ::= *E*
11	L ::= *L , E*

This grammar doesn't seem to be ambiguous. However, this is not LL(1) grammar since it has left recursion on S, E and L. Using the formal left recursion solution, we transform the grammar to:

rule #	production
1	S ::= S' S''
2	S'' ::= *;* S' S''
3	S'' ::= ϵ
4	S' ::= *id := E*
5	S' ::= *print (L)*
6	E ::= E' E''
7	E'' ::= + E' E''
8	E'' ::= ϵ
9	E' ::= *id*
10	E' ::= *num*
11	E' ::= *(S , E)*
12	L ::= *E* L''
13	L'' ::= *, E* L''
14	L'' ::= ϵ

This transformed grammar does not have left recursion or common prefix on RHS of production rules. Calculating the FIRST and FOLLOW sets, we get:

	nullable	FIRST	FOLLOW
S	no	{id print}	{, $}
S''	yes	{;}	{, $}
S'	no	{id print}	{; , $}
E	no	{id num (}	{,) ; $}
E''	yes	{+}	{,) ; $}
E'	no	{id num (}	{, +) ; $}
L	no	{id num (}	{)}
L''	yes	{,}	{)}

Now, lets consider all non-terminals and verify the LL(1) property on them.

There is only one production rule for non-terminals S, E and L. So the LL(1) property holds trivially.

S''	since one of the production rules steps into ϵ, we see that $FIRST(;S'S'') \cap FOLLOW(S'') = \{;\} \cap FOLLOW(S'') = \emptyset$. Hence the LL(1) property holds
S'	$FIRST(id := E) \cap FIRST(print(L)) = \{id\} \cap \{print\} = \emptyset$
E''	since one of the production rules steps into ϵ, we see that $FIRST(+E'E'') \cap FOLLOW(E'') = \{+\} \cap FOLLOW(E'') = \emptyset$.
E'	$FIRST(id) \cap FIRST(num) = \emptyset$
L''	since one of the production rules steps into ϵ, we see that $FIRST(,EL'') \cap FOLLOW(L'') = \{,\} \cap FOLLOW(L'') = \emptyset$.

Hence the grammar is LL(1).