Assignment 2

Ayush Kumar (170195)

12 February 2020

I pledge on my honor that I have not given or received any unauthorized assistance.

1. We have the given CFG,

$$S \rightarrow (L) \mid a$$
$$L \rightarrow L, S \mid LS \mid b$$

Clearly from the above grammar, we see that the production rules $L \rightarrow L, S$ and $L \rightarrow LS$ introduce **direct left-recursion** in L. Also, there is **no indirect left-recursion** in L or S. To remove the left-recursion, we will introduce a new non-terminal L' and will rewrite the two left-recursive rules as $L \rightarrow bL'$ and $L' \rightarrow , SL' \mid SL' \mid \epsilon$. The final grammar obtained is given below,

$$S \rightarrow (L) \mid a$$
$$L \rightarrow bL'$$
$$L' \rightarrow , SL' \mid SL' \mid \epsilon$$

Also, it seems there is no need to left-factor the grammar, since there are no common prefixes on the RHS of the production rules. Now, we deduce the FIRST and FOLLOW sets for the non-terminals S, L, L'. They are given by,

$$FIRST(S) = \{(a\}$$
$$FIRST(L) = \{b\}$$
$$FIRST(L') = \{, (a\epsilon\}$$
$$FOLLOW(S) = \{\$, (a)\}$$
$$FOLLOW(L) = \{)\}$$
$$FOLLOW(L') = \{)\}$$

Using the algorithm discussed in class, the predictive parsing table for the modified grammar is constructed as shown in **Table 1**,

Non-Terminal	(	)	a	b	,	$
S	$S \rightarrow (L)$		$S \rightarrow a$			
L				$L \rightarrow bL'$		
L'	$L' \rightarrow SL'$	$L' \rightarrow \epsilon$	$L' \rightarrow SL'$		$L' \rightarrow, SL'$	

Table 1: Predictive Parsing Table

2. We have the following CFG,

$$S \rightarrow Lp \mid qLr \mid sr \mid qsp$$
$$L \rightarrow s$$

We add the initializer rule $S' \rightarrow S$, where S' is the new start state, to the above grammar to get,

$$\begin{aligned}
&1.\ S' \to S\\
&2.\ S \to Lp\\
&3.\ S \to qLr\\
&4.\ S \to sr\\
&5.\ S \to qsp\\
&6.\ L \to s
\end{aligned}$$

To show that this grammar is not SLR(1), we will construct the SLR parsing table and look for multiple entries in a cell of the table. First, we enumerate the FIRST and FOLLOW sets for the non-terminals S', S, L.

$$\begin{aligned}
FIRST(S') &= \{qs\}\\
FIRST(S) &= \{qs\}\\
FIRST(L) &= \{s\}\\
FOLLOW(S') &= \{\$\}\\
FOLLOW(S) &= \{\$\}\\
FOLLOW(L) &= \{pr\}
\end{aligned}$$

Now we enumerate the canonical sets of LR(0) items and create the LR(0) automaton using the canonical sets as states and the GOTO functions as transitions. The final LR(0) automaton obtained is as shown in **Figure 1**,

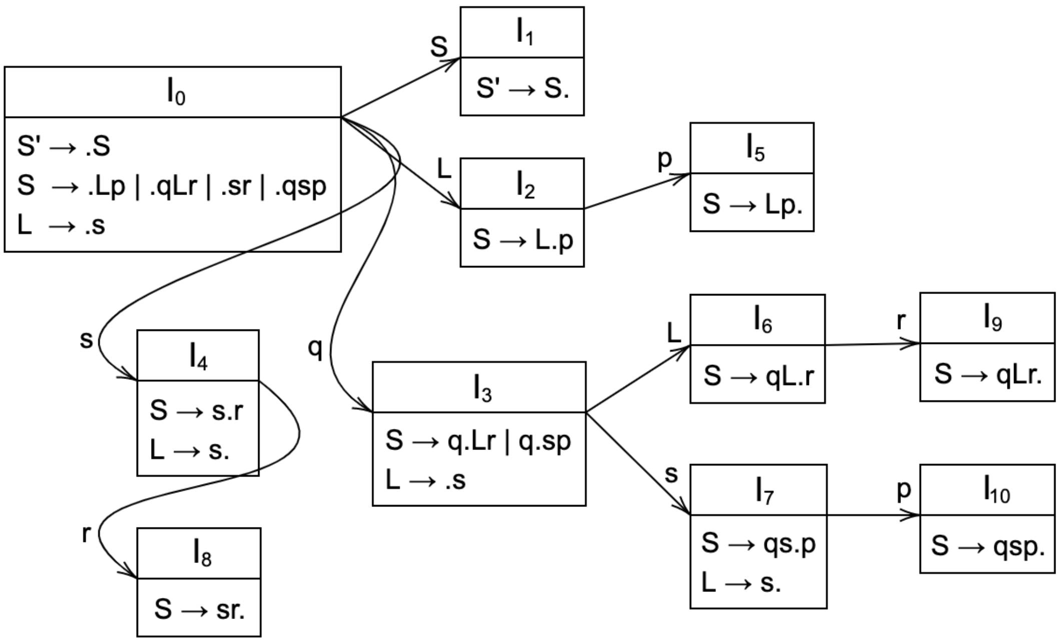

Figure 1: LR(0) Automaton

From the automaton, we now construct the SLR parsing table using the algorithm discussed in class. The SLR parsing table is shown in **Table 2**,

State	Action					Goto	
	p	q	r	s	$	S	L
0		s3		s4		1	2
1					acc		
2	s5						
3				s7			6
4	r6		s8, r6				
5					r2		
6			s9				
7	s10, r6		r6				
8					r4		
9					r3		
10					r5		

Table 2: SLR(1) Parsing Table

From the above table, we find that there occurs **shift-reduce conflicts** in the states I_4 and I_7 with the next tokens being r and p respectively. Hence the grammar is not SLR(1). To find whether it is an LALR(1) grammar we will now create the canonical sets for LR(1) items and using it we will create the LR(1) automaton. The LR(1) automaton for the same grammar is shown in **Figure 2**,

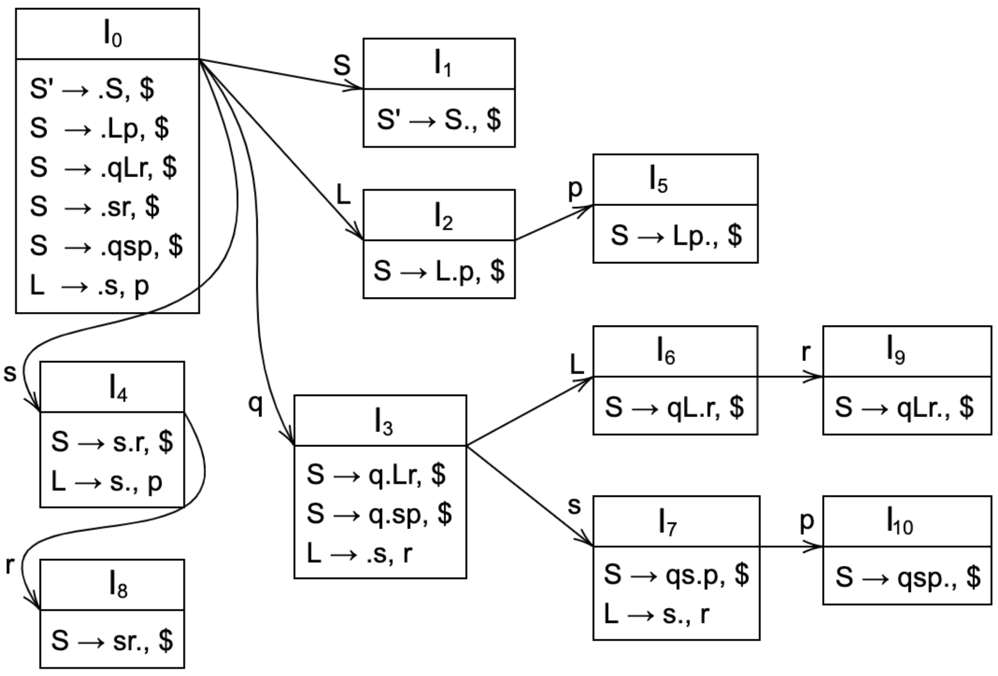

Figure 2: LR(1) Automaton

We see that the LR(1) automaton also has the same number of states as the LR(0) automaton and so there are no states with the same set of core LR(0) items that we could merge. Using the automaton, we construct the LALR parsing table shown in **Table 3**,

State	Action					Goto	
	p	q	r	s	$	S	L
0		s3		s4		1	2
1					acc		
2	s5						
3				s7			6
4	r6		s8				
5					r2		
6			s9				
7	s10		r6				
8					r4		
9					r3		
10					r5		

Table 3: LALR(1) Parsing Table

There are **no shift-reduce or reduce-reduce conflicts**, hence we conclude the given grammar is LALR(1).

3. We have the CFG,

$$R \to R`|'R \mid RR \mid R* \mid (R) \mid a \mid b$$

After adding the initializer rule $S \to R$ with S as the new start state we get the rules,

1. $S \to R$
2. $R \to R`|'R$
3. $R \to RR$
4. $R \to R*$
5. $R \to (R)$
6. $R \to a$
7. $R \to b$

Note that the '|' in the 2nd rule is the actual "or" symbol and not a separator for two production rules. From now onwards, we will use it without quotes and without surrounding spaces to indicate the literal '|'. The FIRST and FOLLOW sets for S and R are,

$$\begin{aligned} FIRST(S) &= \{ab(\} \\ FIRST(R) &= \{ab(\} \\ FOLLOW(S) &= \{\$\} \\ FOLLOW(R) &= \{| * (ab)\$\} \end{aligned}$$

Constructing the LR(0) automaton the same way we did before we get (Figure 3 & 4),

I_0

S → .R
R → .R|R | .RR | .R*
R → .(R) | .a | .b

I_1

S → R.
R → R.|R | R.R | R.*
R → .R|R | .RR | .R*
R → .(R) | .a | .b

I_2

R → (.R)
R → .R|R | .RR | .R*
R → .(R) | .a | .b

I_3

R → a.

I_4

R → b.

I_5

R → RR.
R → R.|R | R.R | R.*
R → .R|R | .RR | .R*
R → .(R) | .a | .b

I_6

R → R|.R
R → .R|R | .RR | .R*
R → .(R) | .a | .b

I_7

R → R*.

I_8

R → (R.)
R → R.|R | R.R | R.*
R → .R|R | .RR | .R*
R → .(R) | .a | .b

I_9

R → R|R.
R → R.|R | R.R | R.*
R → .R|R | .RR | .R*
R → .(R) | .a | .b

I_{10}

R → (R).

Figure 3: Canonical Sets of LR(0) Items

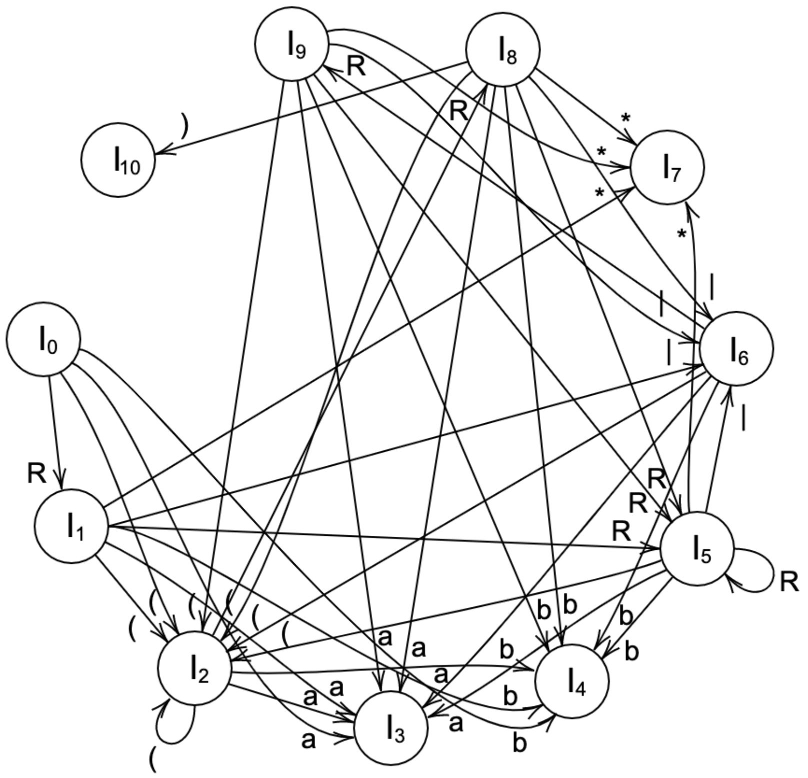

Figure 4: LR(0) Automaton

Using this, we once again construct the SLR parsing table (Table 4),

State	Action							Goto
	|	$*$	(	a	b	)	\$	R
0			s2	s3	s4			1
1	s6	s7	s2	s3	s4		acc	5
2			s2	s3	s4			8
3	r6	r6	r6	r6	r6	r6	r6	
4	r7	r7	r7	r7	r7	r7	r7	
5	s6, r3	s7, r3	s2, r3	s3, r3	s4, r3	r3	r3	5
6			s2	s3	s4			9
7	r4	r4	r4	r4	r4	r4	r4	
8	s6	s7	s2	s3	s4	s10		5
9	s6, r2	s7, r2	s2, r2	s3, r2	s4, r2	r2	r2	5
10	r5	r5	r5	r5	r5	r5	r5	

Table 4: SLR(1) Parsing Table

In the above table, we have **10 shift-reduce conflicts**. To resolve them, we use the following disambiguating rules which closely follows from the natural precedence rules for star, or, parentheses and concatenation operators used in Regex.

Disambiguation Rules:

(a) Since '|' gets the lowest precedence in regexes, we **always reduce** in case of a shift-reduce conflict with the lookahead token being '|'.

(b) '$*$' has the highest precedence after '()' in regexes, hence we **always shift** in case of a shift-reduce conflict with the lookahead token being '$*$'.

(c) In both the states I_5 and I_9, shifting of '(' faces conflict with reduction by concatenation and or operators respectively which are lower in precedence order. So we **shift** in both cases.

(d) Now consider the terminals 'a' and 'b' as the lookahead tokens. For state I_5 it effectively doesn't matter whether we shift or reduce, but for the state I_9 we have to **shift** because the canonical set I_9 contains reduction by the rule $R \rightarrow R|R$ which has the lowest precedence and cannot be applied before shifting 'a', 'b'. Hence we resolve all conflicts by deciding to **shift** 'a', 'b' in both the cases.

Using these disambiguation rules, we get the final SLR parsing table as shown in Table 5,

State	Action							Goto
	|	$*$	(	a	b	)	\$	R
0			s2	s3	s4			1
1	s6	s7	s2	s3	s4		acc	5
2			s2	s3	s4			8
3	r6	r6	r6	r6	r6	r6	r6	
4	r7	r7	r7	r7	r7	r7	r7	
5	r3	s7	s2	s3	s4	r3	r3	5
6			s2	s3	s4			9
7	r4	r4	r4	r4	r4	r4	r4	
8	s6	s7	s2	s3	s4	s10		5
9	r2	s7	s2	s3	s4	r2	r2	5
10	r5	r5	r5	r5	r5	r5	r5	

Table 5: SLR(1) Parsing Table with Disambiguation Rules

4. For this question, we built a Lexer and Parser using Lex and Yacc [1] for tokenizing and parsing the thesis. The lexer tokenizes the text into tokens $TITLE$, $CHAPTERNAME$, $SECTIONNAME$ and $PARAGRAPH$

and also keeps a count of the number of words and different types of sentences encountered. Using these tokens as terminals and s being the start state, the CFG for recognizing the thesis is given by,

$$\begin{aligned}
s &\rightarrow TITLE\ chapters \\
chapters &\rightarrow chapters\ chapter \mid chapter \\
chapter &\rightarrow CHAPTERNAME\ paragraphs\ sections \\
chapter &\rightarrow CHAPTERNAME\ paragraphs \\
chapter &\rightarrow CHAPTERNAME\ sections \\
sections &\rightarrow sections\ section \mid section \\
section &\rightarrow SECTIONNAME\ paragraphs \\
paragraphs &\rightarrow paragraphs\ paragraph \mid paragraph \\
paragraph &\rightarrow PARAGRAPH
\end{aligned}$$

This is implemented in the *common.h*, *thesislexer.l* and *thesisparser.y* files. To run the parser on a sample thesis, run the command

```
$ sh script.sh samplethesis.txt
```

This sends the output to the Standard Output (stdout). To redirect the output of the parser into another file, use the command

```
$ sh script.sh samplethesis.txt > parsedfile.txt
```

References

[1] Lex & Yacc http://dinosaur.compilertools.net/

[2] Interfacing Lex & YACC https://www.tldp.org/HOWTO/Lex-YACC-HOWTO-6.html

[3] Stackoverflow https://stackoverflow.com/questions/56105741/segmentation-fault-when-returnin-pointer-in-yacc