LALR Parser

Space efficient CLR (with slightly reduced recognition power)

LALR

- Lookahead LR
- Often used in practice.
- CLR has an increased number of states
  - C language will have a few thousands of states in LR(1) automaton.
- SLR is good but fails to parse often.
  - C language will have a few hundreds of states in LR(0) automaton.
- LALR has reduced number of states (similar in size to that of SLR)
  - C language will have a few hundreds of states in LALR automaton.

Example 4.54: Consider the following augmented grammar.

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & C\ C \\ C & \rightarrow & c\ C \mid d \end{array} \tag{4.55}$$

- The grammar generates the regular language c*dc*d
- LR(1) automaton
- CLR parse table

STATE	ACTION			GOTO	
	c	d	$\$$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

CLR(1) parsing table

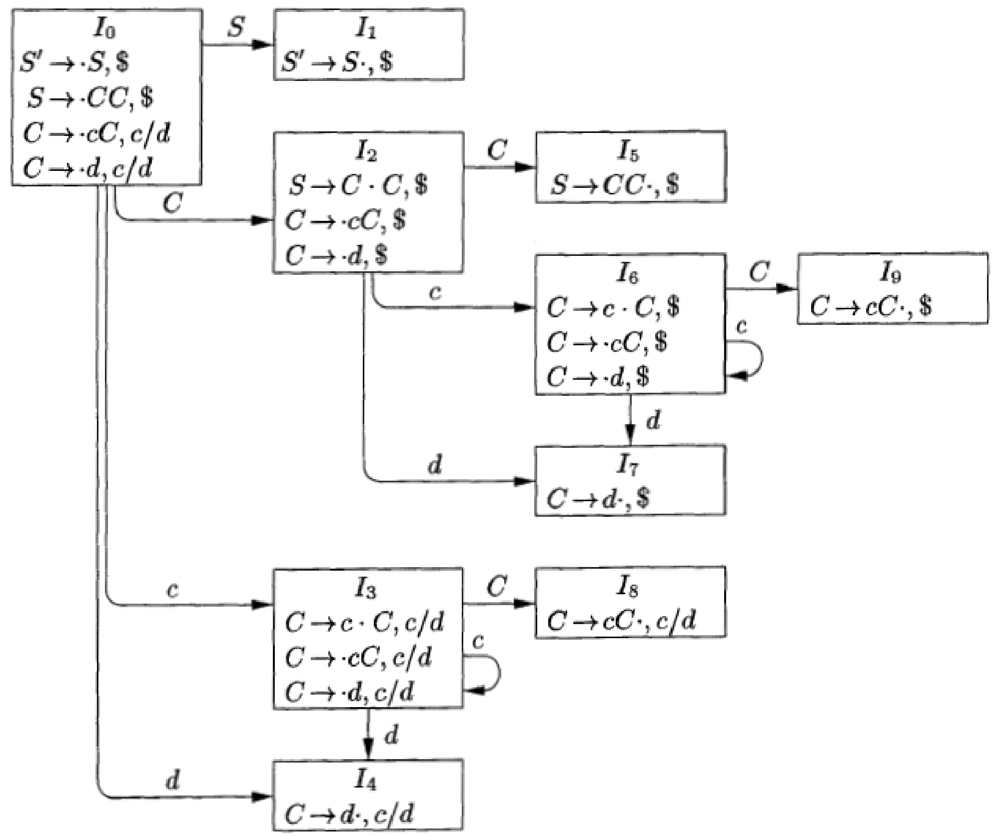


Figure 4.41: The GOTO graph for grammar (4.55)

When reading an input $cc \cdots cdcc \cdots cd$, the parser shifts the first group of c's and their following d onto the stack, entering state 4 after reading the d.

The parser then calls for a reduction by $C \to d$, provided the next input symbol is c or d.

The requirement that c or d follow makes sense, since these are the symbols that could begin strings in $\mathbf{c^*d}$.

If \$ follows the first d, we have an input like ccd, which is not in the language, and state 4 correctly declares an error if \$ is the next input.

The parser enters state 7 after reading the second d.

Then, the parser must see \$ on the input, or it started with a string not of the form $\mathbf{c^*dc^*d}$.

It thus makes sense that state 7 should reduce by $C \rightarrow d$ on input \$ and declare error on inputs c or d.

I_4 and I_7 are similar looking states.

Can we merge them in to a single state?

Let us call the merged state I_{47}, consisting of the set of three items represented by $[C \to d\cdot,\ c/d/\$]$.

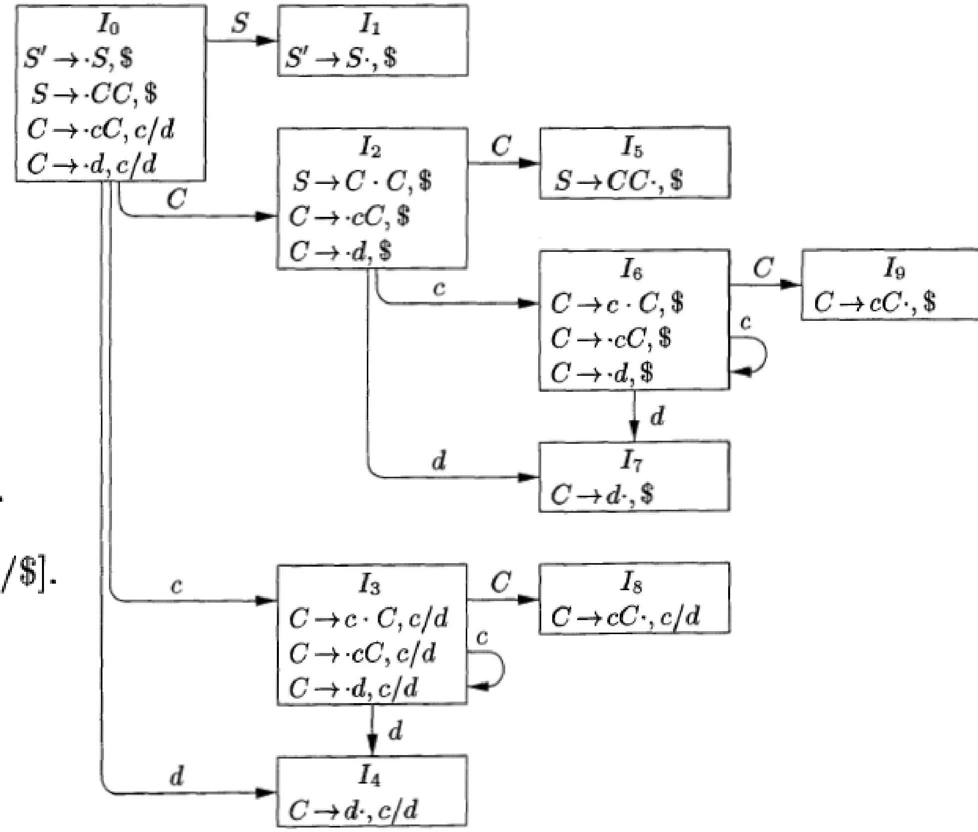


Figure 4.41: The GOTO graph for grammar (4.55)

The error declaration was previously done immediately after *ccd*
But, now *d* is reduced to *C*, and we will goto state 8. In state 8 with \$ as input, it is an error. Error is caught but at a later stage.

Similarly, for *cdcdc* the error is caught.

Core?

- LR(1) item has two components. First one is a production with a dot on r.h.s.; second is the look-ahead.
- In LR(1) item set, leaving the second, the remaining is called **core**.

For the following two LR(1) item sets	Core is same, and it is:
I_6: $C \to c \cdot C, \$$; $C \to \cdot cC, \$$; $C \to \cdot d, \$$ I_3: $C \to c \cdot C, c/d$; $C \to \cdot cC, c/d$; $C \to \cdot d, c/d$	$C \to c \cdot C$; $C \to \cdot cC$; $C \to \cdot d$

LR(1) items having the same core can be merged.

For example, in Fig. 4.41, I_4 and I_7 form such a pair, with core $\{C \to d\cdot\}$. Similarly, I_3 and I_6 form another pair, with core $\{C \to c\cdot C, C \to \cdot cC, C \to \cdot d\}$. There is one more pair, I_8 and I_9, with common core $\{C \to cC\cdot\}$.

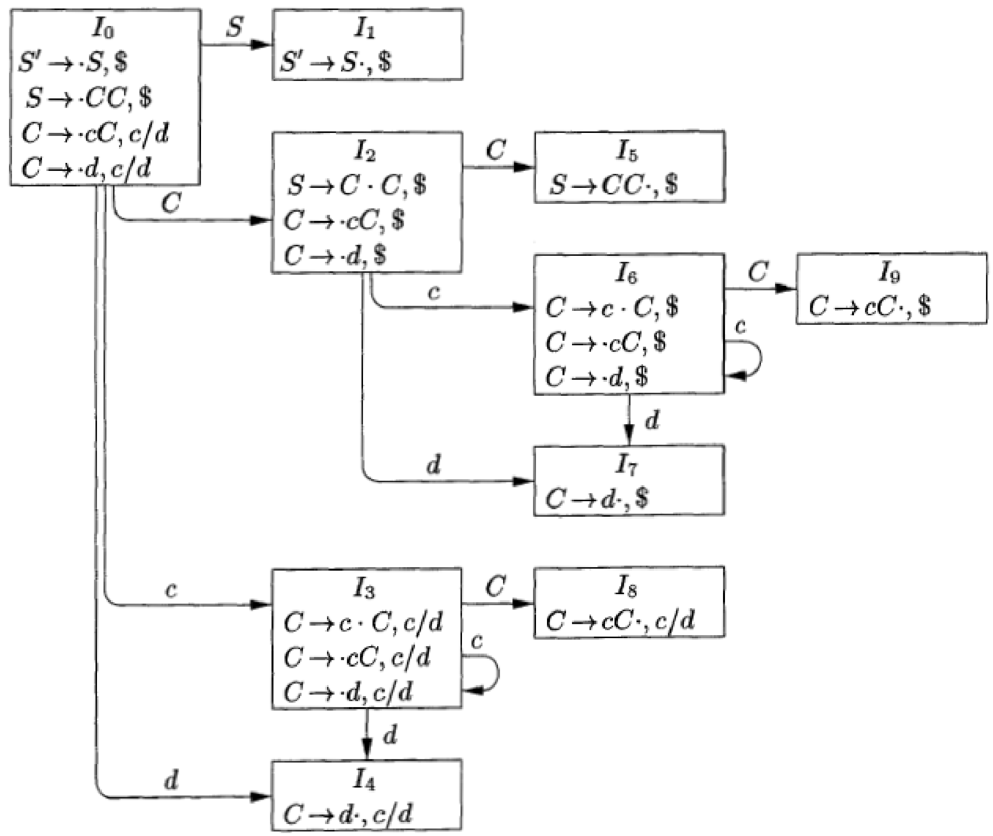


Figure 4.41: The GOTO graph for grammar (4.55)

STATE	ACTION			GOTO	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Canonical parsing table for grammar (4.55)

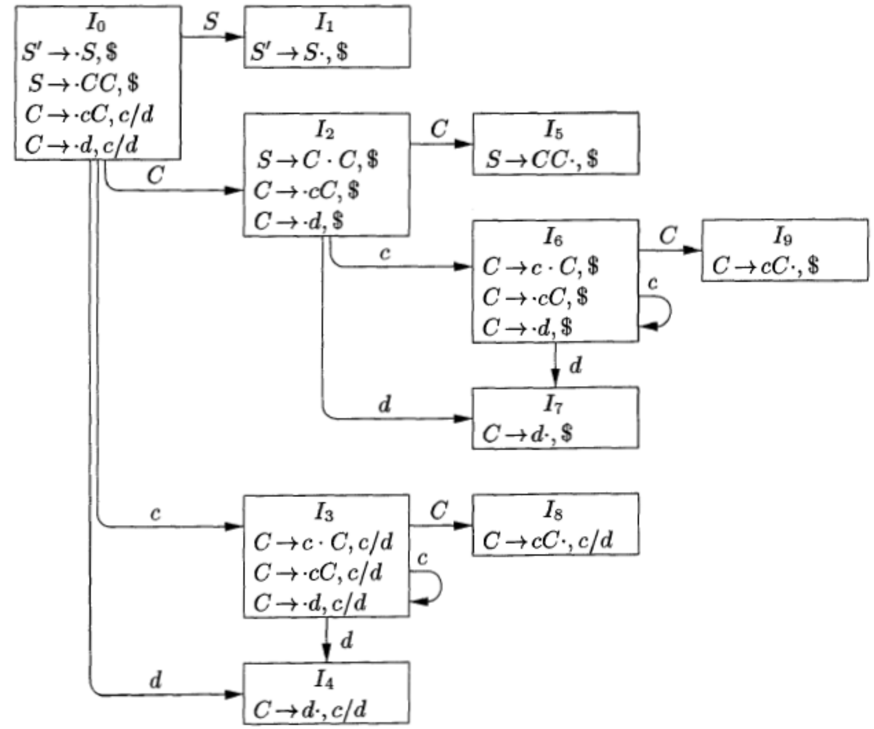
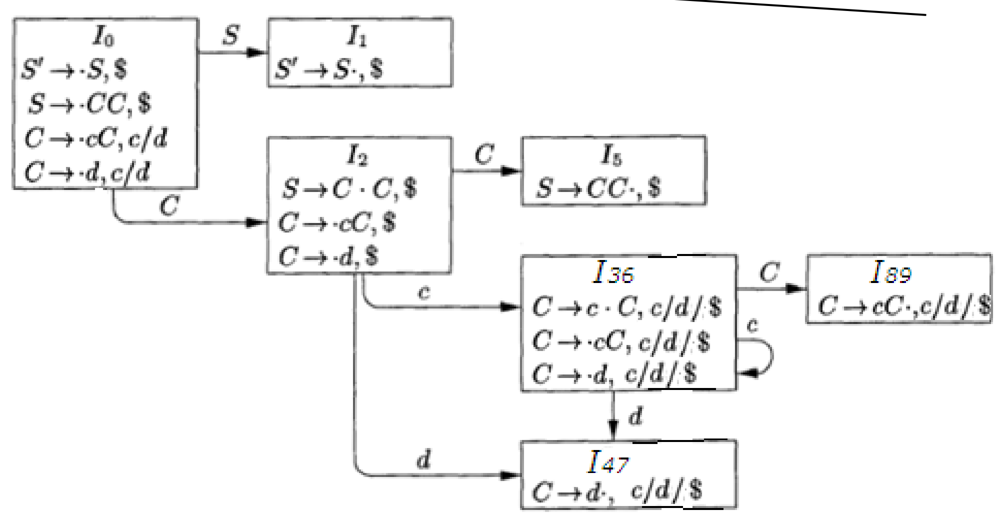


Figure 4.41: The GOTO graph for grammar (4.55)

STATE	ACTION			GOTO	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Figure 4.43: LALR parsing table for the grammar



Merging? No problems!

- If no problems, then merge !!
  - Because of a merge, you will never introduce a shift-reduce conflict. {We can prove this}
  - But, it may possible that you may introduce a reduce-reduce conflict.
    - In this case don’t merge
    - WE SAY THE GRAMMAR IS NOT LALR.

Merging does not produce shift-reduce conflicts, if LR(1) was conflict free.

- After merging, let us say, we got $[A \rightarrow \alpha., a]$ and $[B \rightarrow \beta.a\gamma, b]$ in to a single state. Then for a lookahead a there is a shift-reduce conflict.
- This is not possible. Why?
- You merged two states having same core. Then the items in those two states should include something like $[A \rightarrow \alpha., a]$and $[B \rightarrow \beta.a\gamma, c]$ for some c.
  - There was a conflict in LR(1) items. [contradiction].

Based on same core we merge, no problems subsequently?

- Let us say, based on same core, we merged, and we have not landed in any problems.
- But, subsequently, don’t we get any problems?
- **No.**
- GOTO(I, X) depends on the core of I, the goto’s of merged sets can themselves be merged.

EXAMPLE: REDUCE-REDUCE CONFLICT

Example 4.58 : Consider the grammar

$$
\begin{array}{lcl}
S' & \to & S \\
S & \to & a\ A\ d \mid b\ B\ d \mid a\ B\ e \mid b\ A\ e \\
A & \to & c \\
B & \to & c
\end{array}
$$

which generates the four strings acd, ace, bcd, and bce. The reader can check that the grammar is LR(1) by constructing the sets of items. Upon doing so,

we find the set of items $\{[A \to c\cdot,\ d],\ [B \to c\cdot,\ e]\}$ valid for viable prefix ac and $\{[A \to c\cdot, e],\ [B \to c\cdot,\ d]\}$ valid for bc. Neither of these sets has a conflict, and their cores are the same. However, their union, which is

$$
\begin{array}{l}
A \to c\cdot,\ d/e \\
B \to c\cdot,\ d/e
\end{array}
$$

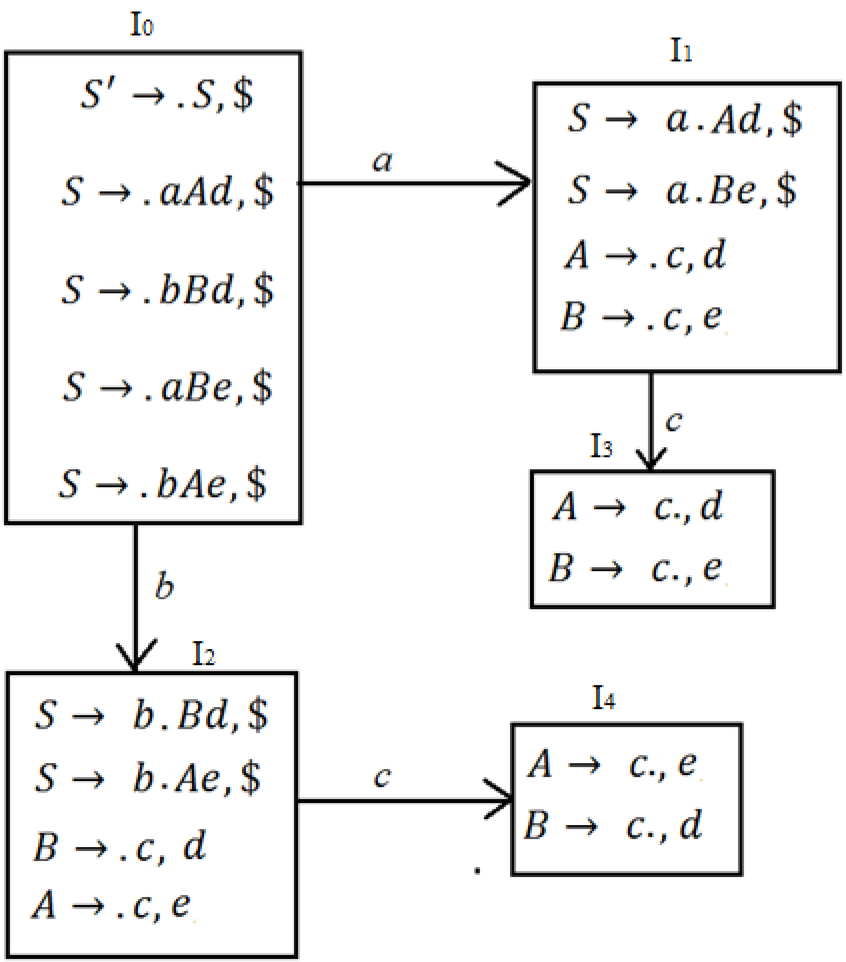
generates a reduce/reduce conflict, since reductions by both $A \to c$ and $B \to c$ are called for on inputs d and e. □

See next slide for a part of LR(1) automaton for these viable prefixes

(1) $S \rightarrow a\ A\ d$
(2) $S \rightarrow b\ B\ d$
(3) $S \rightarrow a\ B\ e$
(4) $S \rightarrow b\ A\ e$
(5) $A \rightarrow c$
(6) $B \rightarrow c$

State	..	d	e	..
1	..	..	..	..
2	..	..	..	..
3	..	r5	r6	..
4	..	r6	r5	..

Fragment of the parse table for LR(1)



State	..	d	e	..
1	..	..	..	..
2	..	..	..	..
34	..	r5/r6	r6/r5	..
..	..	...	..	..

Fragment of the parse table after merging.

Part of LR(1) automaton {to elucidate the example}

CLR Vs. LALR

- On correct inputs, CLR and LALR mimic each other.
  - State names may differ; but when one does shift, other also does shift; when one does reduce, other also reduces (on same production).
- With erroneous input, CLR finds error quickly and stops. But, LALR may proceed to do some more reductions (it will not do shifts!). And, eventually lands in error.

A method to construct LALR parser

- Build LR(1) automaton
- Find same cores and merge them
- Now create the parse table
- If no conflicts, then it is the LALR parse table.
- Else (you may be having reduce-reduce conflicts, because of merging), declare that the grammar is not LALR.

Previous method is space consuming

- There is a direct method that works with LR(0) automaton to create LALR parser.
- Refer Dragon Book.

Compaction of LALR parse table

- Normally used parsing method is LALR
- But, space is still a problem. {Eventhough, we reduced the parse table size than that of CLR}.

Space of the parse table is a problem.

- 50 to 100 terminals
- 100 productions
- LALR parse table can have several 100s of states.
- Action part of the parse table can easily have 20,000 entries
- So regular 2D array storage consumes lot of space.

STATE	ACTION			GOTO	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

- One can find same rows often.
- Use pointers to access same rows.
- Coding a row as a 1D array is good to reduce time.
- But represent a row as a list. (terminal, action) represents an element of this list.
  - This takes advantage of many blank entries.
  - Blank entries can be pushed towards the end. You can give a single entry for it.
  - Then name it as (any, error)

- Blank (error) entries can be safely replaced by some reduction entries (penalty is, error is encountered at a later stage).

List is better

- For each row, list is better.

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

actions for states 0, 4, 6, and 7 agree. We can represent them all by the list

SYMBOL	ACTION
id	s5
(	s4
any	error

In state 2, we can replace the error entries by r2, so reduction by production 2 will occur on any input but $*$. Thus the list for state 2 is

*	s7
any	r2

Ambiguous grammars

Precedence and associativity can be used to resolve shift/reduce conflicts in ambiguous grammars.

- lookahead with higher precedence $\Rightarrow$ *shift*
- same precedence, left associative $\Rightarrow$ *reduce*

Advantages:

- more concise, albeit ambiguous, grammars
- shallower parse trees $\Rightarrow$ fewer reductions

Classic application: expression grammars

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$