

THEORY OF LR PARSING

The canonical collection of LR(0) items, discussed in detail in SLR(1) Parser, can also be represented by a directed labeled graph. Let the canonical collection of LR(0) items be represented as sets, $C = \{I_0, I_1, \dots, I_n\}$.

- A node of the graph, labeled I_i , is constructed for each member of C .
- For every nonempty collection given by $\text{goto}(I_i, X) = I_k$, for a grammar symbol X , a directed edge (I_i, I_k) is added to the graph labeled with X .
- The graph is a deterministic finite automaton with the node labeled I_0 as the start state and all other nodes treated as final states.
- The finite automaton associated with the collection of items for a sample declaration grammar is constructed brick by brick and shown in the following figures for illustration.

Construction of SLR(1) Automaton for Expression Grammar

Expression Grammar : 1) $E' \rightarrow E$ 2,3) $E \rightarrow E + T \mid T$ 4, 5) $T \rightarrow T * F \mid F$
6, 7) $F \rightarrow (E) \mid \text{id}$

1. E' is the new start symbol in the augmented grammar.
2. We begin with the first collection of LR(0) items, named as I_0 . The kernel item of I_0 is always, $E' \rightarrow \bullet E$ which relates the new start symbol with the old start symbol. To complete the collection of items in I_0 , find the $\text{closure}(E' \rightarrow \bullet E)$.
3. $\text{Closure}(E' \rightarrow \bullet E)$: since $\bullet E$ appears in the rhs of the LR(0) item, it results in including all rules of E with a \bullet at the first position in their rhs, namely, $E \rightarrow \bullet E + T$ and $E \rightarrow \bullet T$
4. The first LR(0) item is already closed and does not add a new item. However $\text{closure}(E \rightarrow \bullet T)$ due to the 2nd item, results in the addition of 2 more items, $T \rightarrow \bullet T * F \mid \bullet F$
5. Continuing further, $\text{closure}(T \rightarrow \bullet F)$ adds two more items, $F \rightarrow \bullet (E) \mid \bullet \text{id}$ The last 2 items do not have \bullet followed by a nonterminal and hence do not need to be closed.
6. The calculations performed above are summarized as follows :
 $I_0 = \text{closure}(E' \rightarrow \bullet E) = \{ E' \rightarrow \bullet E; E \rightarrow \bullet E + T \mid \bullet T; T \rightarrow \bullet T * F \mid \bullet F; F \rightarrow \bullet (E) \mid \bullet \text{id} \}$
 $\text{closure}(\bullet F) = \{ F \rightarrow \bullet (E) \mid \bullet \text{id} \};$ $\text{closure}(\bullet T) = \{ T \rightarrow \bullet T * F \mid \bullet F; F \rightarrow \bullet (E) \mid \bullet \text{id} \};$ and
 $\text{closure}(\bullet E) = \{ E \rightarrow \bullet E + T \mid \bullet T; T \rightarrow \bullet T * F \mid \bullet F; F \rightarrow \bullet (E) \mid \bullet \text{id} \}$
7. The last three closures were obtained as a byproduct. Since these closures are independent of the context, they can readily be reused in the calculations that follow. This collection is represented as a node, where the kernel item is separated from the items added by closure, by drawing a **red line** separating them.

Once a set of items is constructed, we need to determine other collections that are generated by it. For each item in the collection, in which \bullet is not at the right end, a new collection of set of items are constructed by moving the \bullet past the grammar symbol that follows it. Items in which \bullet is at the right end do not permit the \bullet to move any further.

For I_0 , we observe the following pairs of \bullet followed by a grammar symbol.

1. Two items with $\bullet E$, as in $\{E' \rightarrow \bullet E; E \rightarrow \bullet E + T\}$. The new collection when \bullet goes past E is created by placing the items $\{E' \rightarrow E\bullet; E \rightarrow E\bullet + T\}$ in the kernel of the new collection, named say as I_1 . This collection is also denoted by the notation, $\text{goto}(I_0, E)$. The collection of all items in I_1 is constructed by taking the closure of the kernel items of I_1 .
2. Two items with $\bullet T$, as in $\{E \rightarrow \bullet T; T \rightarrow \bullet T * F\}$. The new collection, when \bullet goes past T , is created by placing the items $\{E \rightarrow T\bullet; T \rightarrow T\bullet * F\}$ in the kernel of the new collection, named say as I_2 . This collection is denoted by the notation, $\text{goto}(I_0, T)$. The collection of all items in I_2 is constructed by taking the closure of the kernel items of I_2 .
3. One item with $\bullet F$, as in $\{T \rightarrow \bullet F\}$. The new collection, when \bullet goes past F , is created by placing the items $\{T \rightarrow F\bullet\}$ in the kernel of the new collection, named say as I_3 . This collection is denoted by the notation, $\text{goto}(I_0, F)$. The collection of all items in I_3 is constructed by taking the closure of the kernel items of I_3 ; however since the \bullet is at the right end in I_3 , closure is not applicable.
4. One item with $\bullet \text{id}$, as in $\{F \rightarrow \bullet \text{id}\}$. The new collection, when \bullet goes past id , is created by placing the items $\{F \rightarrow \text{id}\bullet\}$ in the kernel of the new collection, named say as I_4 . This collection is denoted by the notation, $\text{goto}(I_0, \text{id})$. The collection of all items in I_4 is constructed by taking the closure of the kernel items of I_4 ; however since the \bullet is at the right end in I_4 , closure is not applicable.
5. One item with $\bullet ($, as in $\{F \rightarrow \bullet (E)\}$. The new collection, when \bullet goes past $($, is created by placing the items $\{F \rightarrow (\bullet E)\}$ in the kernel of the new collection, named say as I_5 . This collection is denoted by the notation, $\text{goto}(I_0, ($). The collection of all items in I_5 is constructed by taking the closure of the kernel items of I_5 ; however since the \bullet is at the right end in I_5 , closure is not applicable.
6. All the 7 LR(0) items in I_0 have now been processed for movement of \bullet past the grammar symbols and the kernels of the 5 new collections constructed. At this stage of the construction, the collection I_0 and its $\text{goto}(I_0, X)$ collections are complete and a summary is captured in the following figure.

<table><tr><td>$E' \rightarrow \bullet E$</td></tr><tr><td>$E \rightarrow \bullet E + T \mid \bullet T$</td></tr><tr><td>$T \rightarrow \bullet T * F \mid \bullet F$</td></tr><tr><td>$F \rightarrow \bullet (E) \mid \bullet \text{id}$</td></tr></table> <p>Collection I_0</p>	$E' \rightarrow \bullet E$	$E \rightarrow \bullet E + T \mid \bullet T$	$T \rightarrow \bullet T * F \mid \bullet F$	$F \rightarrow \bullet (E) \mid \bullet \text{id}$	<p>Kernel of $I_1 = \text{Kernel}(\text{goto}(I_0, E)) = \{ E' \rightarrow E\bullet; E \rightarrow E\bullet + T \}$</p> <p>Kernel of $I_2 = \text{Kernel}(\text{goto}(I_0, T)) = \{ E \rightarrow T\bullet; T \rightarrow T\bullet * F \}$</p> <p>Kernel of $I_3 = \text{Kernel}(\text{goto}(I_0, F)) = \{ T \rightarrow F\bullet \}$</p> <p>Kernel of $I_4 = \text{Kernel}(\text{goto}(I_0, \text{id})) = \{ F \rightarrow \text{id}\bullet \}$</p> <p>Kernel of $I_5 = \text{Kernel}(\text{goto}(I_0, ()) = \{ F \rightarrow (\bullet E) \}$</p>
$E' \rightarrow \bullet E$					
$E \rightarrow \bullet E + T \mid \bullet T$					
$T \rightarrow \bullet T * F \mid \bullet F$					
$F \rightarrow \bullet (E) \mid \bullet \text{id}$					

The collection of LR(0) items, I_1 through I_5 are constructed and displayed in the following table. It can be seen that in I_1 , I_2 , I_3 and I_4 do not get any items added by closure as \bullet is not followed by a nonterminal. Collection I_5 has 6 more LR(0) items added because of $\text{closure}(E)$, which has already been precomputed.

$I_1 = \text{goto}(I_0, E)$	$I_2 = \text{goto}(I_0, T)$	$I_3 = \text{goto}(I_0, F)$	$I_4 = \text{goto}(I_0, \text{id})$	$I_5 = \text{goto}(I_0, ()$
<div> $E' \rightarrow E\bullet$ $E \rightarrow E\bullet + T$ <hr/> Φ </div>	<div> $E \rightarrow T\bullet$ $T \rightarrow T\bullet * F$ <hr/> Φ </div>	<div> $T \rightarrow F\bullet$ <hr/> Φ </div>	<div> $F \rightarrow \text{id}\bullet$ <hr/> Φ </div>	<div> $F \rightarrow (\bullet E)$ <hr/> $E \rightarrow \bullet E + T \mid \bullet T$ $T \rightarrow \bullet T * F \mid \bullet F$ $F \rightarrow \bullet (E) \mid \bullet \text{id}$ </div>

The SLR(1) automaton at this stage of construction is drawn below. The LR(0) items contained in the nodes are omitted for brevity. The SLR(1) parsing table is shown in the other part of this table. The entries are self-explanatory, with sk, denoting the parser action **shift to state k**. There is no reduce action in state 0, since no handles are seen in this state (no item with \bullet at the right end).

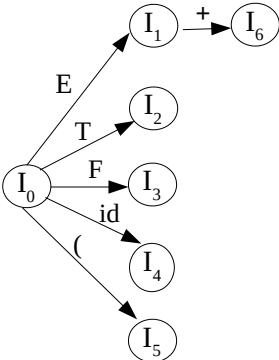
SLR(1) Automaton	SLR (1) Parsing Table										
	State	id	+	*	()	\$		E	T	F
	0	s4			s5				1	2	3

7. Let us explore the collection denoted by I_1 . The kernel items are $\{ E' \rightarrow E\bullet; E \rightarrow E\bullet + T \}$. The first item is a complete item and indicates a reduce by the associated rule 1, we usually denote this move as, “r1”, with the intended meaning as reduce by rule 1. The other item, $E \rightarrow E\bullet + T$, results in \bullet moving past E and forms the kernel item of the collection $\text{goto}(I_1, +)$, which is named as I_6 . The collection of items in I_6 , after its closure is $\{ E \rightarrow E + \bullet T; T \rightarrow \bullet T * F \mid \bullet F; F \rightarrow \bullet (E) \mid \bullet \text{id} \}$ and shown below.

$E' \rightarrow E\bullet$ $E \rightarrow E\bullet + T$ <hr/> Φ
I_1

$E \rightarrow E + \bullet T$ <hr/> $T \rightarrow \bullet T * F \mid \bullet F$ $F \rightarrow \bullet (E) \mid \bullet \text{id}$
I_6

We now have all the information to populate the row for state 1. Reduction by the rule, $E' \rightarrow E\bullet$, indicates that the entire input has been reduced to E' which is the unique accept move of this parser. This move is placed in the action[1, FOLLOW(E')] table for all terminals in FOLLOW(E'), which happens to be $\{\$ \}$ in this case.

SLR(1) Automaton	SLR (1) Parsing Table									
	State	id	+	*	()	\$	E	T	F
	0	s4			s5			1	2	3
	1		s6				acc			

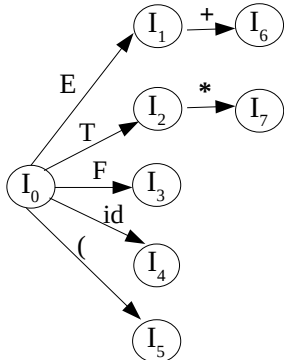
8. Let us construct the row for state 2 of the parsing table. This requires examining the LR(0) items in I_2 and also all the goto(I_2 , X) states. The items in I_2 are $\{E \rightarrow T\bullet; T \rightarrow T\bullet * F\}$. The first item calls for a reduce by rule 3 on all symbols in FOLLOW(E) = $\{+,), \$\}$. The 2nd item leads to a new state, goto(I_2 , $*$), named as I_7 . The LR(0) items in both the states are given below.

$E \rightarrow T\bullet$
$T \rightarrow T\bullet * F$
Φ

I_2

$T \rightarrow T\bullet * F$
$F \rightarrow \bullet(E) \mid \bullet id$

I_7

SLR(1) Automaton	SLR (1) Parsing Table									
	State	id	+	*	()	\$	E	T	F
	0	s4			s5			1	2	3
	1		s6				acc			
	2		r3	s7		r3	r3			

9. We now go to populate the table for the states 3, 4 and 5 together. The set of all LR(0) items in all the 3 states are reproduced below. The only item in I_3 , $T \rightarrow F \bullet$, calls for a reduce by rule 5 on all symbols in $\text{FOLLOW}(T) = \{*, +,), \$\}$. The only item in I_4 , $F \rightarrow \text{id} \bullet$, calls for a reduction by rule 7 on all symbols in $\text{FOLLOW}(F) = \text{FOLLOW}(T) = \{*, +,), \$\}$.

10. State I_5 has no complete items, no reduce action in state 5. Because of \bullet being followed by the 5 grammar symbols $\{E, T, F, (, \text{id}\}$, we construct the sets of LR(0) items, denoted by $\text{goto}(I_5, E)$, $\text{goto}(I_5, T)$, $\text{goto}(I_5, F)$, $\text{goto}(I_5, '(')$ and $\text{goto}(I_5, \text{id})$. The kernel of $\text{goto}(I_5, E)$ is $\{F \rightarrow (E \bullet), E \rightarrow E \bullet + T\}$ which is distinct from the kernels of all states constructed so far (though it has one item common with I_1), and hence we create a new state I_8 . It can be easily verified that kernel of $\text{goto}(I_5, T)$ is $\{E \rightarrow T \bullet\}$ which is same as $\text{kernel}(I_2)$, kernel of $\text{goto}(I_5, F)$ is $\{T \rightarrow F \bullet\}$ which is $\text{kernel}(I_3)$, kernel of $\text{goto}(I_5, '(')$ is $\{F \rightarrow (\bullet E)\}$ which is $\text{kernel}(I_5)$ and $\text{goto}(I_5, \text{id})$ is $\{F \rightarrow \text{id} \bullet\}$ which is $\text{kernel}(I_4)$.

$E \rightarrow T \bullet$	$T \rightarrow F \bullet$	$F \rightarrow \text{id} \bullet$	$F \rightarrow (\bullet E)$	$F \rightarrow (E \bullet)$
Φ	Φ	Φ	$E \rightarrow \bullet E + T \mid \bullet T$ $T \rightarrow \bullet T * F \mid \bullet F$ $F \rightarrow \bullet (E) \mid \bullet \text{id}$	$E \rightarrow E \bullet + T$
$I_2 = \text{goto}(I_0, T)$ $= \text{goto}(I_5, T)$	$I_3 = \text{goto}(I_0, F)$ $= \text{goto}(I_5, F)$	$I_4 = \text{goto}(I_0, \text{id})$ $= \text{goto}(I_5, \text{id})$	$I_5 = \text{goto}(I_0, '(')$ $= \text{goto}(I_5, '(')$	$I_8 = \text{goto}(I_5, E)$

SLR(1) Automaton	SLR (1) Parsing Table									
	State	id	+	*	()	\$	E	T	F
	0	s4			s5			1	2	3
	1		s6				acc			
	2		r3	s7		r3	r3			
	3		r5	r5		r5	r5			
	4		r7	r7		r7	r7			
	5	s4			s5			8	2	3

11. For I_6 , there are transitions on the grammar symbols $\{T, F, \text{id}, '('\}$. The kernel of $\text{goto}(I_6, T)$ has the items $\{E \rightarrow E + T \bullet; T \rightarrow T * F \bullet\}$ which is distinct from all the existing ones and is named I_9 , the kernels of the other $\text{goto}()$ are existing states, such as $\text{goto}(I_6, F)$ is I_3 , $\text{goto}(I_6, '(')$ is I_5 and $\text{goto}(I_6, \text{id})$ is I_4 .

12. For I_7 , there are transitions on 3 symbols $\{F, \text{id}, '('\}$. The kernel of $\text{goto}(I_7, F)$ is $\{T \rightarrow T * F \bullet\}$ which is new state and numbered as 10. The kernels of $\text{goto}(I_7, \text{id})$ and $\text{goto}(I_7, '(')$ are existing states 4 and 5 respectively.

13. For I_8 , there are transitions on the symbol $\{'\}$. The kernel of $\text{goto}(I_8, ')')$ is $\{F \rightarrow (E) \bullet\}$ which is new state and numbered as 11.

$E \rightarrow E + \bullet T$
$T \rightarrow \bullet T * F \mid \bullet F$
$F \rightarrow \bullet (E) \mid \bullet id$
I_6

$T \rightarrow T * \bullet F$
$F \rightarrow \bullet (E) \mid \bullet id$
I_7

$F \rightarrow (E \bullet)$
$E \rightarrow E + \bullet T$
Φ
I_8

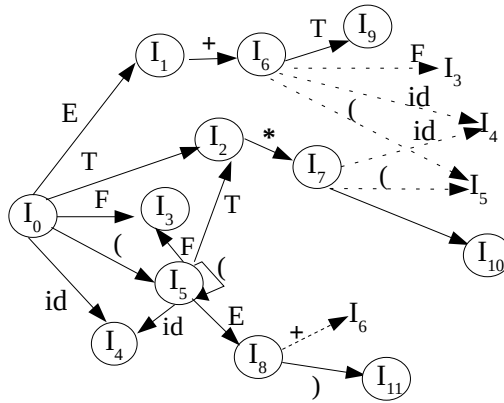
$E \rightarrow E + T \bullet$
$T \rightarrow T * \bullet F$
Φ
I_9

$T \rightarrow T * F \bullet$
Φ
I_{10}

$F \rightarrow (E) \bullet$
Φ
I_{11}

These computations are then inserted into the table filling up all entries of the states from 0 to 8.

SLR(1) Automaton	SLR (1) Parsing Table									
	State	id	+	*	()	\$	E	T	F
The automaton is drawn separately. Dotted arrows with labels are used to denote a transition to an existing state in order to avoid clutter in the graph. Solid edges with labels are used to denote a transition to a new state. The states are numbered in the order of creation during the construction process.	0	s4			s5			1	2	3
	1		s6				acc			
	2		r3	s7		r3	r3			
	3		r5	r5		r5	r5			
	4		r7	r7		r7	r7			
	5	s4			s5			8	2	3
	6	s4			s5				9	3
	7	s4			s5					10
	8		s6		s11					



14. Three more states, 9, 10 and 11 have been created so far, whose entries in the table have not been entered. The states 10 and 11, have a complete item each and hence have no transitions out of these nodes. State 10 calls for reduce by rule 4 for all symbols in $FOLLOW(T) = \{+, *,), \$\}$. Similarly state 11 calls for reduce by rule 6 for all symbols in $FOLLOW(F) = \{*, +,), \$\}$.

$E \rightarrow E + T \bullet$
$T \rightarrow T * \bullet F$
Φ
I_9

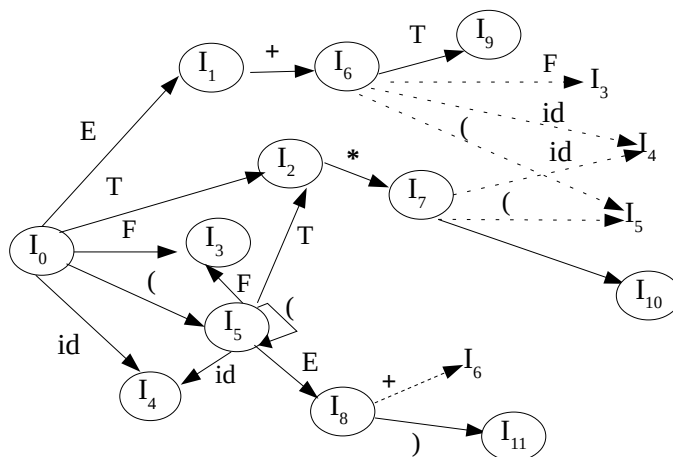
$T \rightarrow T * F \bullet$
Φ
I_{10}

$F \rightarrow (E) \bullet$
Φ
I_{11}

$T \rightarrow T * \bullet F$
$F \rightarrow \bullet (E) \mid \bullet id$
$goto(I_9, *) = I_7$

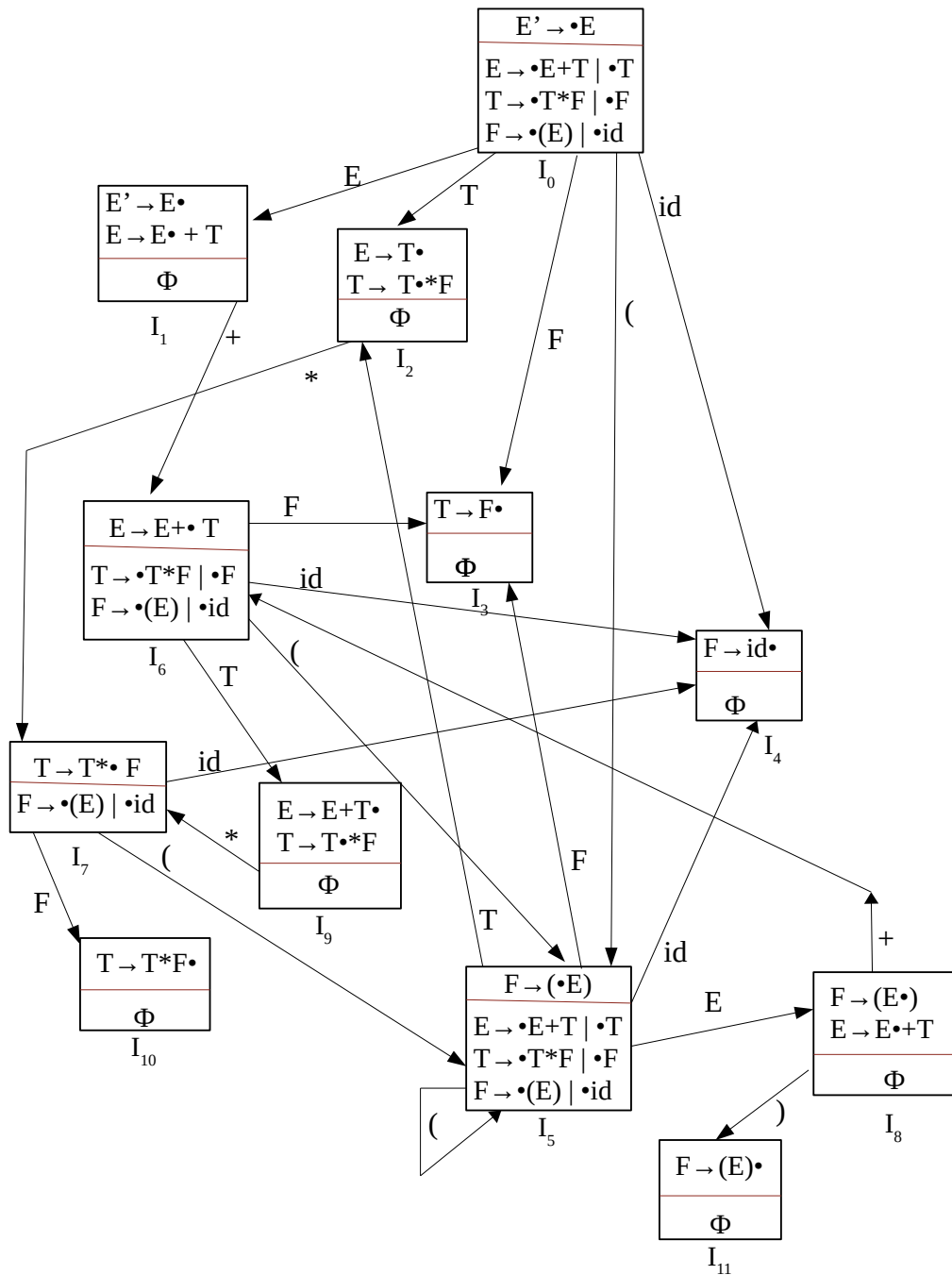
15. State 9 has a complete kernel item, $E \rightarrow E + T \cdot$, which calls for reduce by rule 2 for all symbols in $\text{FOLLOW}(E) = \{+,), \$\}$. The other kernel item is, $T \rightarrow T \cdot * F$, which results in the item, $T \rightarrow T \cdot * F$, in the kernel of $\text{goto}(I_9, *)$. However this state has already been constructed and is I_7 . Since no new state has been constructed and all the states have been processed for the LR(0) items contained therein, the construction of the SLR(1) automaton and its parsing table is complete. The final automaton and the parsing table is given below.
16. In the Action[] table, all blank entries denote a syntax error. In the Goto[] table, a blank entry is Not Applicable, as the parser can never reach such configuration. Why is the previous statement true?

SLR(1) Automaton	SLR (1) Parsing Table									
	State	id	+	*	()	\$	E	T	F
The automaton is drawn separately. Dotted arrows with labels are used to denote a transition to an existing state in order to avoid clutter in the graph. Solid edges with labels are used to denote a transition to a new state. The states are numbered in the order of creation during the construction process.	0	s4			s5			1	2	3
	1		s6				acc			
	2		r3	s7		r3	r3			
	3		r5	r5		r5	r5			
	4		r7	r7		r7	r7			
	5	s4			s5			8	2	3
	6	s4			s5				9	3
	7	s4			s5					10
	8		s6			s11				
	9		r2	s7		r2	r2			
	10		r4	r4		r4	r4			
	11		r6	r6		r6	r6			



A few important observations follow.

- The closure($\bullet A$), for every nonterminal A can be computed once and used repeatedly.
- The kernel items of a state s, together with the closure($\bullet A$) sets are sufficient to generate the kernel items of the goto(s, X).
- The SLR(1) automaton and the parsing table can be constructed efficiently by the two steps mentioned above.



Viable Prefix and Valid LR(0) Items

A viable prefix of a grammar is defined to be the prefixes of right sentential forms that do not contain any symbols to the right of a handle.

Only the kernel items of each state is shown in the above.

1. By adding terminal symbols to viable prefixes, rightmost sentential forms can be constructed.
2. Viable prefixes are precisely the set of symbols that can ever appear on the stack of a shift reduce parser.
3. A viable prefix either contains a handle or contains parts of one or more handles.
4. Given a viable prefix, if one could identify the set of potential handles associated with it then a recognizer for viable prefixes would also recognize handles.

The significance of LR(0) item can now be understood. A complete item corresponds to a handle while an incomplete item indicates the part of a handle seen so far, the symbols to the left of dot (\bullet), while the symbols to the right of dot are yet to be seen in the input processed thus far.

A LR(0) item $A \rightarrow \beta_1 \bullet \beta_2$ is defined to be valid for a viable prefix, $\alpha\beta_1$, provided

$$S \Rightarrow_{rm}^* \alpha A w \Rightarrow_{rm} \alpha\beta_1\beta_2w \quad \dots (1)$$

That is from the start symbol S , we are able to reach a right sentential form, $\alpha\beta_1\beta_2w$, using the LR(0) item, $A \rightarrow \beta_1\bullet\beta_2$. The proof of validity of this item for the prefix $\alpha\beta_1$ is provided by the rm -derivation given in (1), which gives evidence about the fact that $\alpha\beta_1$ is a prefix of the right sentential form $\alpha\beta_1\beta_2w$ and of the rule, $A \rightarrow \beta_1\beta_2$, the part β_1 (before \bullet) has been seen, while the part β_2 is yet to appear.

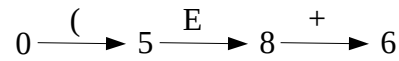
1. It is interesting to note that in above, if $\beta_2 = B\gamma$ where B is a nonterminal and $B \rightarrow \delta$ is a rule, then $B \rightarrow \bullet\delta$ is also a valid item for this viable prefix. This is because, when we continue with (1), we get $S \Rightarrow_{rm}^* \alpha A w \Rightarrow_{rm} \alpha\beta_1\beta_2w = \alpha\beta_1B\gamma w \Rightarrow_{rm} \alpha\beta_1\delta\gamma w \quad \dots (2)$

The last right sentential form in (2), $\alpha\beta_1\delta\gamma w$, shows that for the viable prefix $\alpha\beta_1$, since δ appears immediately to its right, the item $B \rightarrow \bullet\delta$ is a valid item. However $B \rightarrow \delta\bullet$ is not a valid item for the viable prefix $\alpha\beta_1$. It may however be a valid item for a different viable prefix.

2. There could be several distinct items which are valid for same viable prefix γ , as was observed above. Therefore a mapping from Sets-of-Items \rightarrow viable-prefix is many to one.
3. Conversely, given a particular LR(0) item, how many distinct viable prefixes exist for which this item is valid. It so turns that this mapping is also many to one.
4. The parser would halt when the viable prefix S is seen, which formally stated is a move from $\bullet S$ to $S\bullet$. This is essentially the reason for augmenting the input grammar.

Illustrative Example : For the SLR(1) automaton given in the Figure above, let us work out these concepts and their relationships in some detail.

- Consider the path from state 0 to state 6 shown below. The labels on the edges traversed is “(E+”.



The items that are placed in state 6 are $\{E \rightarrow E+\bullet T; T \rightarrow \bullet T * F; T \rightarrow \bullet F; F \rightarrow \bullet (E); F \rightarrow \bullet id\}$.

Q. Why exactly these 5 LR(0) items are placed in state 6 and no other LR(0) items?

The answer lies in the understanding the relationship between viable prefix and valid items.

1. For the given path, the viable prefix is “(E +”. We wish to examine the rightmost derivations that include this viable prefix.
2. $E' \Rightarrow_{rm} E \Rightarrow_{rm} T \Rightarrow_{rm} F \Rightarrow_{rm} (E) \Rightarrow_{rm} (E + T)$... (1)
- $E' \Rightarrow_{rm} E \Rightarrow_{rm} T \Rightarrow_{rm} F \Rightarrow_{rm} (E) \Rightarrow_{rm} (E + T) \Rightarrow_{rm} (E + T * F)$... (2)
- $E' \Rightarrow_{rm} E \Rightarrow_{rm} T \Rightarrow_{rm} F \Rightarrow_{rm} (E) \Rightarrow_{rm} (E + T) \Rightarrow_{rm} (E + F)$... (3)
- $E' \Rightarrow_{rm} E \Rightarrow_{rm} T \Rightarrow_{rm} F \Rightarrow_{rm} (E) \Rightarrow_{rm} (E + T) \Rightarrow_{rm} (E + F) \Rightarrow_{rm} (E + (E))$... (4)
- $E' \Rightarrow_{rm} E \Rightarrow_{rm} T \Rightarrow_{rm} F \Rightarrow_{rm} (E) \Rightarrow_{rm} (E + T) \Rightarrow_{rm} (E + F) \Rightarrow_{rm} (E + id)$... (5)

Can you write another rightmost sentential form that includes the string “(E+” but is not included above ?

3. Consider derivation (1) above. $E' \Rightarrow_{rm}^* (E + T)$ has the string “(E+” as a prefix. The remaining string is “T)”. The LR(0) item, $E \rightarrow E+T$, has the part “E+” of the prefix already seen, hence the item, $E \rightarrow E+\bullet T$, is a valid item for “(E+”.
4. Consider derivation (2) above. $E' \Rightarrow_{rm}^* (E + T * F)$ has the string “(E+” as a prefix. The remaining string is “T * F)”. The LR(0) item, $T \rightarrow T * F$, has the no part of “E+” of the prefix, hence the item, $T \rightarrow \bullet T * F$, is also a valid item for “(E+”.
5. As a self assessment exercise, find the valid items for derivations (3) to (5).

Q. Why a particular LR(0) item, say $T \rightarrow \bullet F$, appears in multiple states, which is 0, 5, and 6 in the given automaton ?

The viable prefix of state 0 is ϵ (the null string). The rightmost derivation, $E' \Rightarrow_{rm} E \Rightarrow_{rm} T \Rightarrow_{rm} F$ is interpreted as, $\epsilon \mid F$, where the viable prefix is to the left of \mid . Since F is yet to be seen, $T \rightarrow \bullet F$ is a valid item for this state.

The viable prefix of state 6 is “E+”. The rightmost derivation, $E' \Rightarrow_{rm} E + T \Rightarrow_{rm} E + F$, can be interpreted as, $E+ \mid F$, where the viable prefix “E+” has been seen and F is expected to appear, showing that $T \rightarrow \bullet F$ is a valid item for this state.

The viable prefix of state 5 is $\{ '(', '(', '(((, '((((, \dots \}$, that is a sequence of one or more left parenthesis. The rightmost derivation, $E' \Rightarrow_{rm}^* F \Rightarrow_{rm} (E) \Rightarrow_{rm}^* (T) \Rightarrow_{rm} (F)$, can be interpreted as, $(\mid F$, where the viable prefix $'('$ has been seen and F is expected to appear, again showing that $T \rightarrow \bullet F$ is a valid item for this state. On similar lines, the derivation, $E' \Rightarrow_{rm}^* (F) \Rightarrow_{rm} ((E)) \Rightarrow_{rm}^* ((T))$, shows that $T \rightarrow \bullet F$ is a valid item for the viable prefix, $((('$ and also for 3 or more consecutive left parantheses.

You should now be in a position to find answers to both the following problems.

- Given a LR(0) item, say $A \rightarrow \alpha 1 \bullet \alpha 2$ write a regular expression for all the viable prefixes for which this item is valid.
- Given a viable prefix, say β , write all the LR(0) items that are valid for β .

THEOREM : The set of all valid items for a viable prefix β is exactly the set of items reached from the start state along a path labeled β in the DFA that can be constructed from the canonical collection of sets of items, with the transitions given by goto.

- The theorem stated without proof above is a key result in LR Parsing. It provides the basis for the correctness of the construction process we have learnt so far.
- An LR parser does not scan the entire stack to determine when and which handle appears on top of stack (compare with shift reduce parser).
- The state symbol on top of stack provides all the information that is present in the stack. The parser moves from one state s_1 to another state s_2 by shifting a terminal t , because it knows that there is at least one handle which includes t in state s_2 , and the complete handle may get exposed subsequently.
- In a state which contains a complete item, that is a handle, a reduction is called for. However, the lookahead symbols for which the reduction should be applied is not obvious. SLR(1) parser uses the FOLLOW information to guide reductions.

Canonical LR(1) Parser

Canonical LR(1) parser is the next parser that we study. In order to motivate the need for having another LR parser, the limitations of SLR(1) parser need to be explained. As we shall see in the following, the FOLLOW information is imprecise and is the reason for issues with SLR(1) parser.

An example to illustrate this fact is the following grammar, which can be seen to be SLR(1) unambiguous. We shall refer to this grammar as a pointer grammar which generates strings such as :

id *id id = id *id = id id = *id *id = *id **id = *id

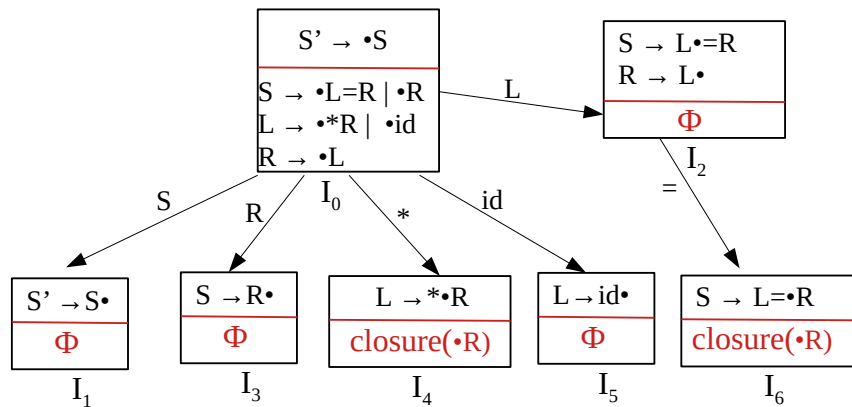
$G = (\{=, *, id\}, \{S, L, R\}, P, S)$ where P , the set of production rules, is given below.

1) $S' \rightarrow S$ (Augmented rule)

2,3) $S \rightarrow L = R \mid R$

4, 5) $L \rightarrow *R \mid id$

6) $R \rightarrow L$



- Consider the state with the viable prefix 'L', labeled as I_2 , as shown above with the LR(0) items present in it.
- The item, $R \rightarrow L \bullet$, in I_2 , calls for reduction by rule 6, that is $R \rightarrow L$. The reduction is called for all symbols in $\text{FOLLOW}(R)$ which is $\{= \$\}$
- The item, $S \rightarrow L \bullet = R$, in I_2 , results in a shifting '=' onto the stack and go to ($I_2, =$) which is numbered, say, as I_6 . The kernel of I_6 is the item $\{S \rightarrow L = \bullet R\}$; other items will get added to I_6 by closure $(\bullet R)$. The summary is the shift move to state 6 on terminal '='.
- The SLR(1) table at this stage of construction is shown below.

State	Action				Goto		
	id	*	=	\$	S	L	R
0	s5	s4	-	-	1	2	3
1	-	-	-	acc	-	-	-
2	-	-	s6 r6	r6	-	-	-

- Action[2, =] shows a shift-reduce conflict. As a consequence the given grammar cannot be directly used by a SLR(1) parser.

SLR(1) Parsing Table for Pointer Grammar

State	Action				Goto		
	id	*	=	\$	S	L	R
0	s5	s4	-	-	1	2	3
1	-	-	-	acc	-	-	-
2	-	-	s6 r6	r6	-	-	-
3	-	-	-	r3	-	-	-
4	s5	s4	-	-	-	8	7
5	-	-	r5	r5	-	-	-
6	s5	s4	-	-	-	8	9
7	-	-	r4	r4	-	-	-
8	-	-	r6	r6	-	-	-
9	-	-	-	r2	-	-	-

You may draw the complete SLR(1) automaton from the parsing table given above by using the same state numbers for the goto() states. As can be seen from the table, except for Action[2, =], there are no other conflicts in any other state.

- The shift move is justified in state 2, because in the new state 6, for the item $S \rightarrow L \bullet = R$, the bullet (\bullet) goes past “=” to the item $S \rightarrow L = \bullet R$ which a valid item for the viable prefix “L=”
- What about the reduction by $R \rightarrow L \bullet$ in state 2? Can the validity of this move this ascertained like the shift move above?
- The reduce move uses the symbols in FOLLOW(R) and since FOLLOW(R) has { $\$, =$ }, the reduce is also unavoidable.
- The only source of error can be in the use of FOLLOW(). Let the correctness of FOLLOW(R) be examined critically. Can “=” follow the nonterminal R (and NOT *R) in some rightmost derivation ?

$$S' \Rightarrow_{rm} R \Rightarrow_{rm} L \quad (6)$$

$$S' \Rightarrow_{rm} R \Rightarrow_{rm} L \Rightarrow_{rm} *R \Rightarrow_{rm} *L \quad (7)$$

$$S' \Rightarrow_{rm} L = R \Rightarrow_{rm} L = L \Rightarrow_{rm} L = *R \Rightarrow_{rm} L = *L \Rightarrow_{rm} L = *id \Rightarrow_{rm} *R = *id \Rightarrow_{rm} *L = *id \quad (8)$$

- You may examine more rightmost derivations starting from S' to consolidate the observation. In derivations such as (6) and (7), “=” does not follow R in any right sentential form, after L is reduced to R. The candidate L has been highlighted for ease of application.

- However in (8), the last sentential form has “=” following “L”. In this situation when L is reduced to R, we find that “=” follows “*R”. Note that the viable prefix for this state is “*L” which is represented by state 8 in the SLR(1) automaton and not “L” which corresponds to state 2.

Q. What is wrong with FOLLOW ?

The information that “=” belongs to FOLLOW(R) is not correct for state 2. The rightmost sentential forms that permit “=” to follow R are of the form “*L=...” and are taken care of in the state 8 of the parser (see complete SLR(1) table).

The context in state 2 is different (viable prefix is L), and the use of FOLLOW constrains the parser. In other words, FOLLOW() is a global information for the grammar and may not hold for every state of the parser.

Q. How to circumvent this problem ?

Given an item and a state the need is to identify the terminal symbols that can actually follow the lhs nonterminal associated with the item.

An item of the form, $A \rightarrow \alpha \cdot \beta, a$ where the first component is a LR(0) item and the second component is a set of terminal symbols is called a **LR(1) item**. We are using the comma (,) symbol to separate the two components. The second component is known as lookahead symbol. The value 1 indicates that the **length of lookahead is 1**.

- The lookahead symbol is used for a reduce operation only. For an item of the form, $A \rightarrow \alpha \cdot \beta, a$ if β is not ϵ , the lookahead has no effect. But for an item, $A \rightarrow \alpha \cdot, a$ the reduction is applied only if the next token is a.
- It can be shown the terminal set X associated in the 2nd component, $A \rightarrow \alpha \cdot, X$ always satisfies the relation $X \subseteq \text{FOLLOW}(A)$.
- A parser that constructs collection of sets of LR(1) items to build a automaton and a parsing table is called as a Canonical LR(1) parser, also referred to as CLR(1). The precision of the CLR(1) parser is better than SLR(1) parser when $X \subset \text{FOLLOW}(A)$.

VALID LR(1) ITEM

An LR(1) item $A \rightarrow \alpha \cdot \beta, a$ is valid for a viable prefix γ , if there is a derivation

$$S \Rightarrow_{rm}^* \delta A w \Rightarrow_{rm} \delta \alpha \beta w, \text{ where}$$

- $\gamma = \delta \alpha$, and
- either a is the first symbol of w, or w is ϵ and a is \$

LR(1) sets of items collection

The basic procedure is same as for LR(0) sets of items collection, the only difference is in specifying how to compute the lookahead for the new LR(1) item obtained by closure.

- If $A \rightarrow \alpha \cdot B\beta, a$ is a valid item for a viable prefix γ , then the item $B \rightarrow \cdot \eta, b$ is also valid for the same prefix γ . Here $B \rightarrow \eta$ is a rule and $b \in \text{FIRST}(\beta a)$.

CONSTRUCTING LR(1) SETS OF ITEMS

The algorithm is similar to the Algorithm for LR(0) sets of item construction and given in the following.

```
procedure items( $G'$ )  
begin  
     $C = \{\text{closure} (S' \rightarrow \cdot S, \$)\};$   
    repeat  
        for each set of items  $I$  in  $C$  and each grammar symbol  $X$ , such that  
             $\text{goto}(I, X)$  is not empty and is not already in  $C$ ,  
        do add  $\text{goto}(I, X)$  to  $C$   
    until no more sets of items can be added to  $C$   
end
```

```
function closure( $I$ )  
begin  
    repeat  
        for each item  $A \rightarrow \alpha \cdot B\beta, a$  and each rule  $B \rightarrow \gamma$  and each  $b \in \text{FIRST}(\beta a)$   
            such that  $B \rightarrow \cdot \gamma, b$  is not in  $I$   
        do add  $B \rightarrow \cdot \gamma, b$  to  $I$ ;  
    until no more items can be added to  $I$ ;  
    return  $I$ ;  
end
```

```
function goto( $I, X$ )  
begin  
     $\text{gotoitems} := \varnothing$  ;  
    for each item  $A \rightarrow \alpha \cdot X\beta, a$  in  $I$ , such that  $A \rightarrow \alpha X \cdot \beta, a$  is not in  $\text{gotoitems}$   
    do add  $A \rightarrow \alpha X \cdot \beta, a$  to  $\text{gotoitems}$   
     $\text{goto} := \text{closure}(\text{gotoitems})$ ;  
    return  $\text{goto}$ ;
```

end

The application of this algorithm to construct a canonical collection of sets of LR(1) items is illustrated in the following.

1. Consider the pointer grammar again. The first collection, i. e., state I_0 , is given by $\text{closure}(S' \rightarrow \bullet S, \$)$. This causes the LR(1) items $S \rightarrow \bullet R, \$$ and $S \rightarrow \bullet L = R, \$$ to be added to I_0 . $\text{Closure}(S \rightarrow \bullet R, \$)$ causes the addition of the item $R \rightarrow \bullet L, \$$ whose closure in turn adds the items $L \rightarrow \bullet * R, \$$ and the item $L \rightarrow \bullet \text{id}, \$$ to I_0 .
2. Closure of the item $S \rightarrow \bullet L = R, \$$ leads to the inclusion of items $L \rightarrow \bullet * R, =$ and $L \rightarrow \bullet \text{id}, =$. The lookaheads for these items are given by $\text{FIRST}(=R \$)$.
3. Since some LR(1) items added by steps 1 and 2 above have the same core, but different lookaheads, we combine the lookaheads to write them in a compact form, $L \rightarrow \bullet * R, \$ =$ and $L \rightarrow \bullet \text{id}, \$ =$.
4. Collection $I_0 = \{S' \rightarrow \bullet S, \$; S \rightarrow \bullet R, \$; S \rightarrow \bullet L = R, \$; L \rightarrow \bullet * R, \$ = ; L \rightarrow \bullet \text{id}, \$ = ; R \rightarrow \bullet L, \$\}$
5. The goto function on I_0 is defined for the symbols S, L, R, id and *. The resulting states are obtained by taking closures of the kernel items as given below.

$$I_1 = \text{goto}(I_0, S) = \text{closure}(S' \rightarrow S \bullet, \$)$$

$$I_2 = \text{goto}(I_0, L) = \text{closure}(S \rightarrow L \bullet = R, \$; R \rightarrow L \bullet, \$)$$

$$I_3 = \text{goto}(I_0, R) = \text{closure}(S \rightarrow R \bullet, \$)$$

$$I_4 = \text{goto}(I_0, *) = \text{closure}(L \rightarrow * \bullet R, \$ =)$$

$$I_5 = \text{goto}(I_0, \text{id}) = \text{closure}(L \rightarrow \text{id} \bullet, \$ =)$$

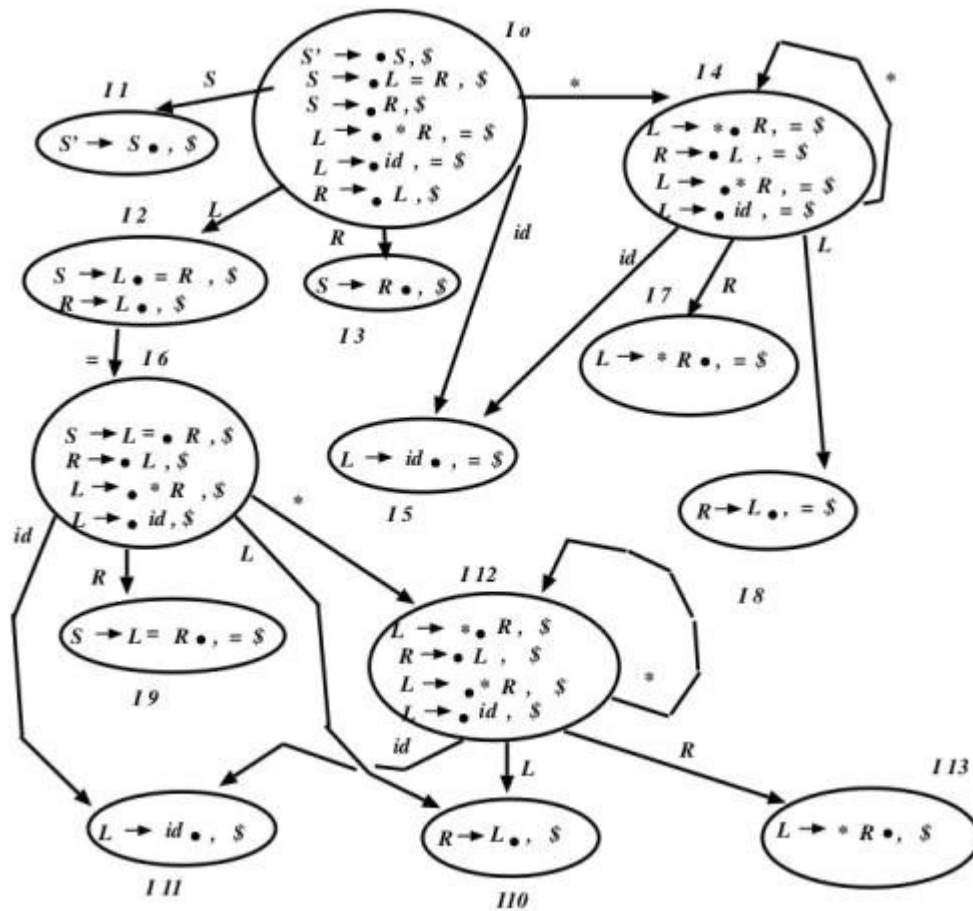
The rest of the collection can be constructed on the same lines. The canonical collection $C = \{I_0, I_1, \dots, I_n\}$ is used to construct the Action and Goto part of the table.

Canonical LR(1) Parsing Table for Pointer Grammar

State	Action				Goto		
	id	*	=	\$	S	L	R
0	s5	s4	-	-	1	2	3
1	-	-	-	acc	-	-	-
2	-	-	s6	r6	-	-	-
3	-	-	-	r3	-	-	-
4	s5	s4	-	-	-	8	7
5	-	-	r5	r5	-	-	-
6	s11	s12	-	-	-	10	9
7	-	-	r4	r4	-	-	-
8	-	-	r6	r6	-	-	-
9	-	-	-	r2	-	-	-
10	-	-	-	r6	-	-	-
11	-	-	-	r5	-	-	-
12	s11	s12	-	-	-	10	13
13	-	-	-	r4	-	-	-

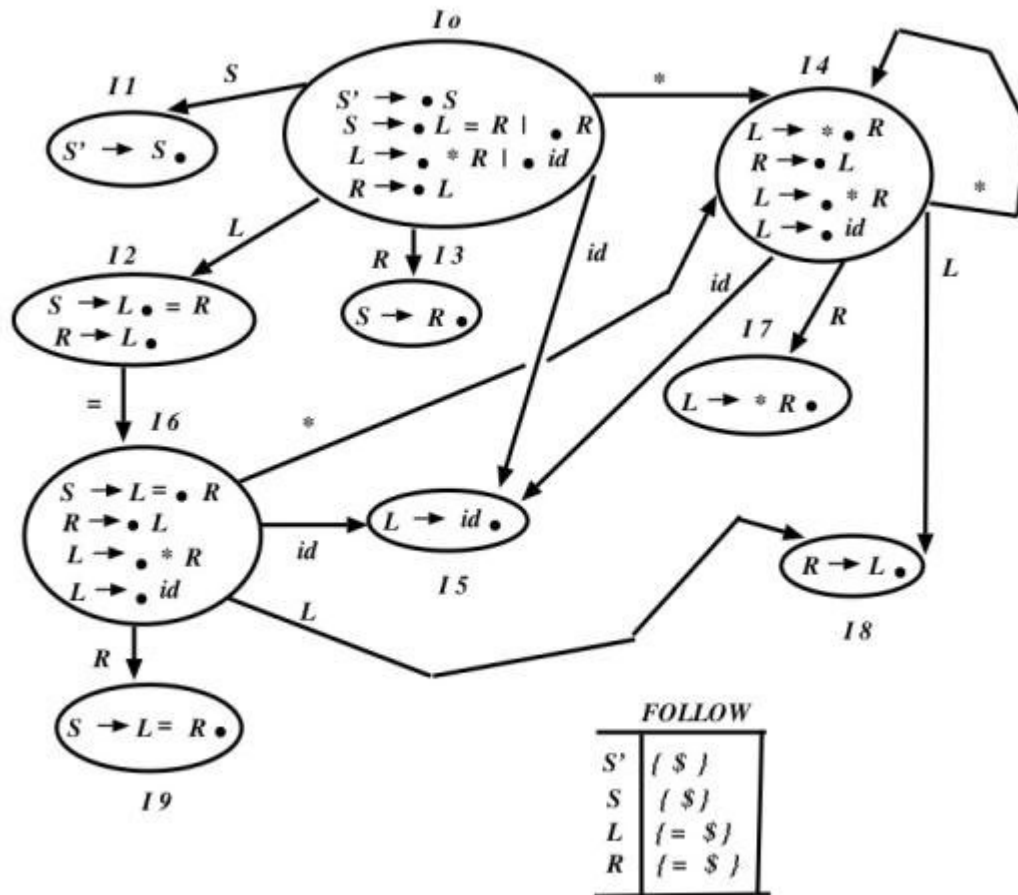
- It is interesting to observe that this automaton has 14 states as compared to SLR(1) automaton which has 10 states.
- The two automaton have identical contents from state 0 to state 5.
- There is reasonable commonality between the remaining states of the two parsers.
- In the parsing table of each parser there are many rows whose entries have high degree of commonality.

CLR(1) Automaton for Pointer Grammar



You may compare the CLR(1) and SLR(1) automaton by keeping them side by side and draw your own conclusions about similarity and differences.

SLR(1) Automaton for Pointer grammar



LALR(1) PARSER CONSTRUCTION

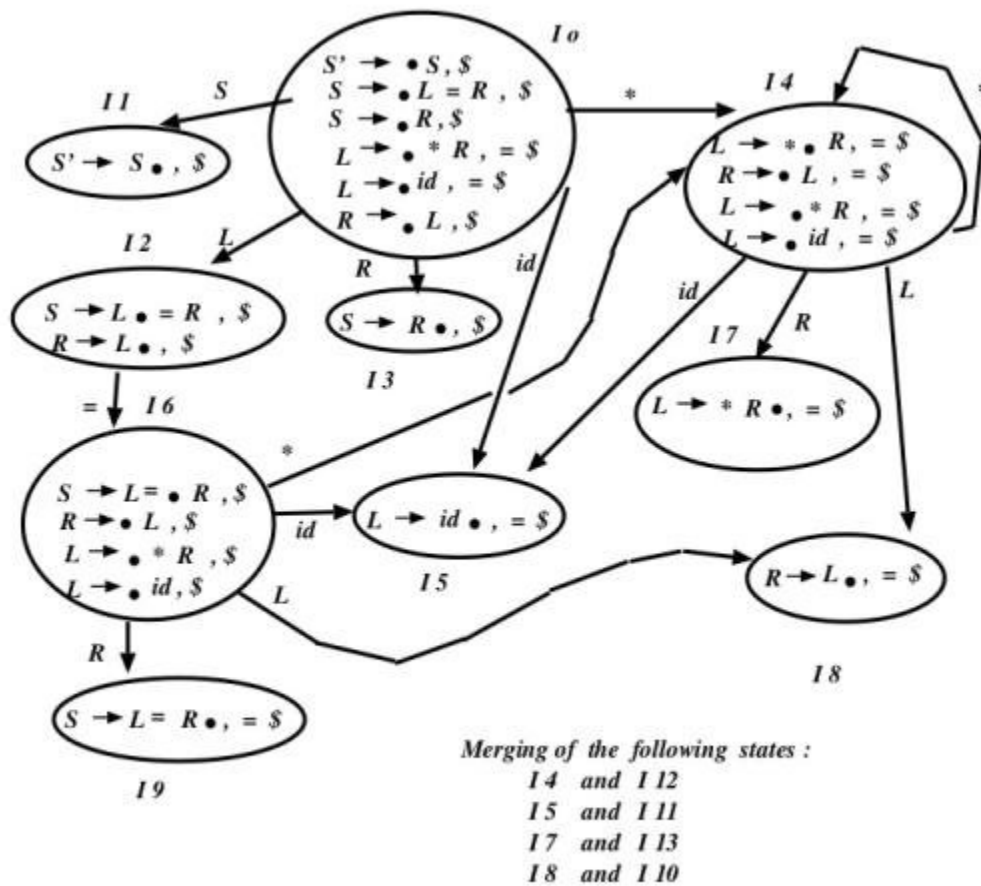
- This parser is of intermediate capability as compared to SLR(1) and CLR(1). The parser has the size advantage of SLR and lookahead capability like CLR.
- In a CLR parser, several states may have the same first components (LR(0) items part), but differ in the lookaheads associated and hence are represented as distinct states.
- In terms of the first component only, both SLR and LR define the same collection of LR(0) items.
- LALR automaton is created by merging states of LR automaton that have the same core (set of first components). The merge results in the union of the lookahead symbols of the respective items.
- This method of LALR(1) automaton construction enforces that CLR(1) automaton be constructed first. However as we show later, one construct LALR(1) automaton directly from the grammar instead of building it via CLR(1) automaton. The reason most literature on LR parsing uses this indirect method is that understanding the essential difference between LALR(1) and CLR(1) is better understood this way.

LALR(1) Parsing Table Construction

1. Construct the collection $C = \{I_0, I_1, \dots, I_n\}$ of CLR(1) items.
2. For each core (set of kernel items) among the set of CLR(1) items, find all sets having the same core and replace these sets by their union.
3. From the reduced collection, say, $C = \{J_0, J_1, \dots, J_m\}$, the Action table is constructed the same way as that for CLR(1).
4. To compute $\text{Goto}[J, X]$, let $J = I_{p1} \cup I_{p2} \cup \dots \cup I_{pr}$, that is collection J of LALR(1) items is the union of all the r collections CLR(1) items, $\{I_{p1}, I_{p2}, \dots, I_{pr}\}$, such that all these CLR(1) collections have the same core. Then construct $K = \{\text{Goto}(I_{p1}, X) \cup \text{Goto}(I_{p2}, X) \cup \dots \cup \text{Goto}(I_{pr}, X)\}$. Finally, we determine, $\text{Goto}[J, X] = K$.

To illustrate the construction of a LALR automaton, consider the LR(1) automaton for the pointer grammar given earlier.

- Examine this automaton to identify the states whose cores are identical. We find that pair of states, given by (I_4, I_{12}) , (I_5, I_{11}) , (I_7, I_{13}) , and (I_8, I_{10}) have identical cores.
- The merger of the states given above results in $I_4 = I_4 \cup I_{12}$, $I_{45} = I_5 \cup I_{11}$, $I_7 = I_7 \cup I_{13}$, and $I_8 = I_8 \cup I_{10}$. This happens since in each case the lookahead sets are proper subsets.
- The LALR automaton and the parsing table for the grammar are given in the following.



You may observe that in I_2 , the shift reduce conflict of SLR(1) has vanished, exactly the same way it was absent in the corresponding state of CLR(1).

LALR(1) Parsing Table for pointer grammar follows from the LALR(1) automaton constructed above.

LALR(1) Parsing Table for Pointer Grammar

State	Action				Goto		
	id	*	=	\$	S	L	R
0	s5	s4	-	-	1	2	3
1	-	-	-	acc	-	-	-
2	-	-	s6	r6	-	-	-
3	-	-	-	r3	-	-	-
4	s5	s4	-	-	-	8	7
5	-	-	r5	r5	-	-	-
6	s5	s4	-	-	-	8	9
7	-	-	r4	r4	-	-	-
8	-	-	r6	r6	-	-	-
9	-	-	-	r2	-	-	-

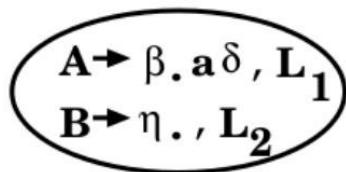
LALR(1) GRAMMAR AND PARSER

A grammar for which there is a conflict free LALR(1) parsing table is called a LALR(1) grammar and a parser which uses such a table is known as LALR(1) parser.

RESULTS IN LALR(1) PARSING

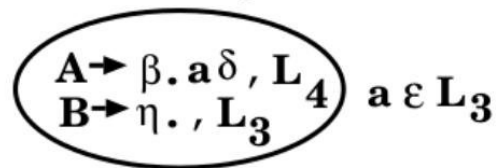
1. A theorem states that a LALR(1) parsing table is always free of shift reduce conflicts, provided the corresponding LR(1) table is so.
2. The key observation towards proving this fact is given in the following figure.

Merged State I_{ij}



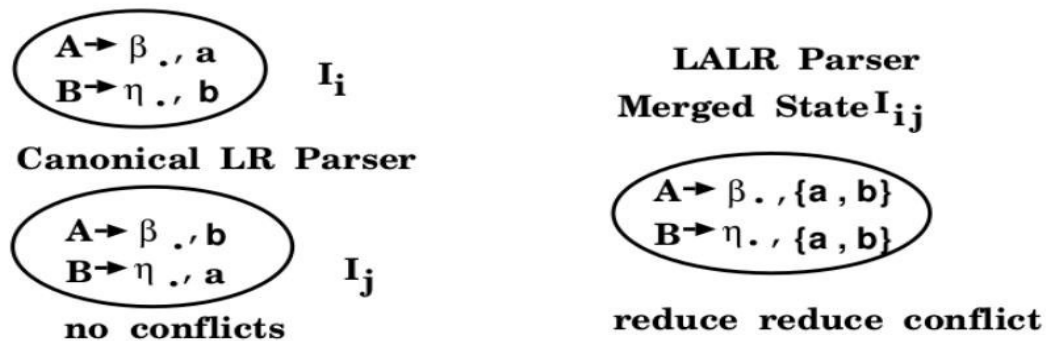
LALR Parser
shift reduce conflict

One of the original states I_i or I_j



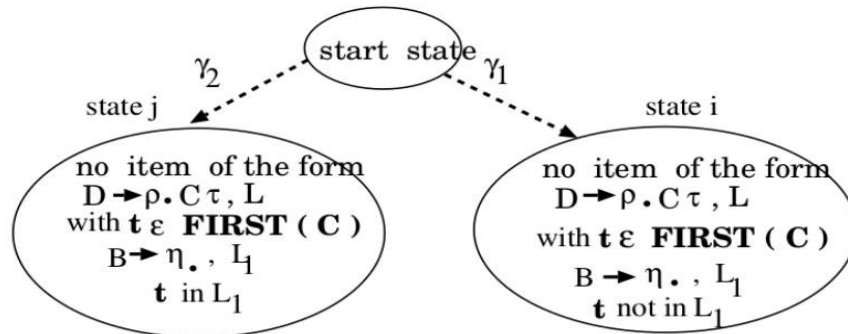
Canonical LR Parser
shift reduce conflict

3. However the theorem does not exclude the possibility of a LALR(1) parsing table having reduce reduce conflicts while the corresponding LR(1) table is conflict free as shown in the situation below.

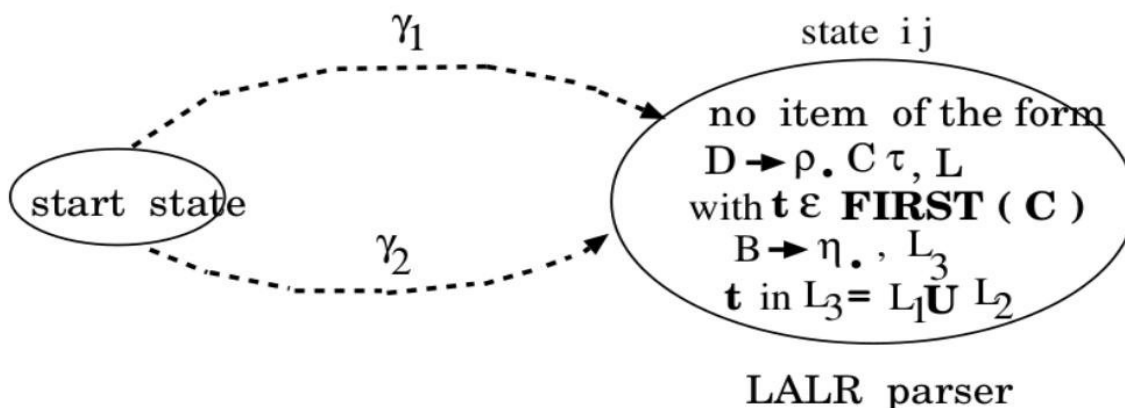


4. Another important result relates the behaviour of these two parsers. It can be formally established that the behaviour of LALR and LR parsers for the same grammar G are identical, that is the same sequence of shifts and reduces, for **a sentence** of the language, $L(G)$.
5. However, for an erroneous input, a LALR parser may continue to perform reductions after the LR parser would have caught and reported the error. An interesting fact is that a LALR parser **does not shift any symbol beyond the point** that the LR parser declared an error.
6. Use the parsers of LALR(1) and LR(1) to parse the input, $id = *id = \$$, and find the difference in behaviour. The reasons are analyzed in the following. The discussion can be converted into a proof for this result if LALR has only one merged state.

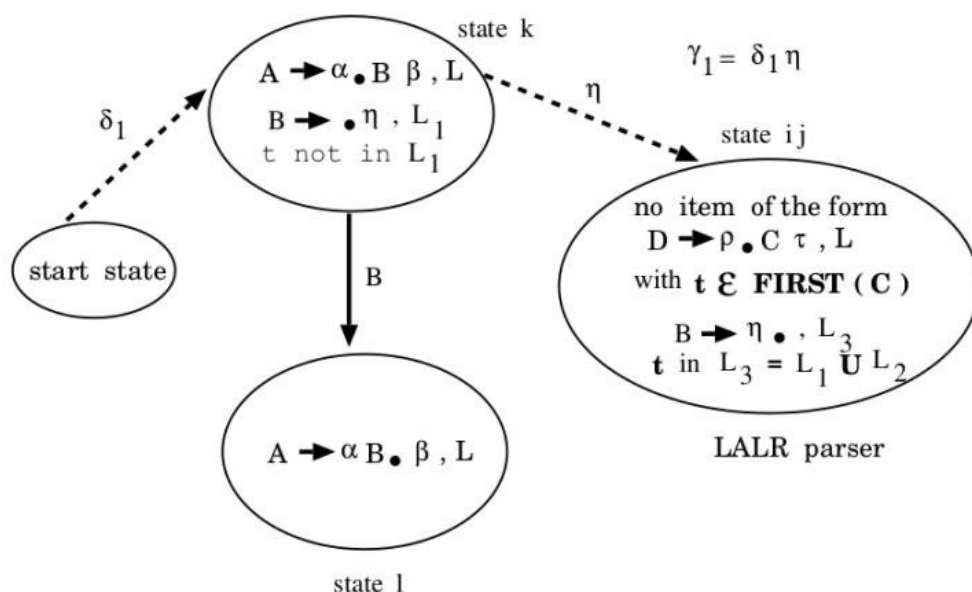
7. Let a CLR(1) parser be in state i with t as the next symbol in the input. This state i recognizes a viable prefix γ_1 . In the figure given below, state i indicates error on t since the first form of items, $D \rightarrow \rho \cdot C \tau, L$ where $t \notin \text{FIRST}(C)$, do not permit t to be shifted. The other LR(1) items of the form, $B \rightarrow \eta \cdot, L_1$ with $t \notin L_1$ prevent any reduce on t . CLR(1) parser will report an error in state i with t as the input symbol. However in the state j of the CLR(1) automaton, which has the same core as that of state i , while a shift on t is not permitted, a reduce action is possible on t .



8. State ij of the LALR(1) automaton, which is the result of merger of states i and j of the corresponding CLR(1) parser, is shown below. This state after recognizing γ_1 permits a reduce on t because of the state j of the LR parser. State ij permits a reduce action on t , inherited from state j of CLR(1).



9. On input t , state ij of LALR(1) will execute reduce by $B \rightarrow \eta \cdot$, and the LALR parser would reach state l whose relevant contents are shown below. It is assumed that states k and l of both the parsers are identical.



Presence of the item $A \rightarrow \alpha \cdot B \beta, L$ in state k is justified since $B \rightarrow \cdot \eta, L_1$ is an item in state k , and can only be added through closure. Terminal t must not be in $\text{FIRST}(\beta)$, since otherwise $t \in L_1$ in state i of the CLR(1) automaton, which is given to be false. Therefore, since $t \notin \text{FIRST}(\beta)$ in state k , this fact is also true for state l whereby state l does not permit a shift on t and consequently detects and reports the error in state l of the LALR(1) parser. This is different from the behaviour of CLR(1) parser which had reported the error in state i . The corresponding LALR(1) parser performs an extra reduce and reported the error in a different state l , without consuming the terminal t .

COMPARISON OF THE THREE LR PARSERS

The observations on SLR(1), CLR(1) and LALR(1) parsers are summarized below.

1. SLR parser is the one with the smallest size and easiest to construct. It uses FOLLOW information to guide reductions.
2. Canonical LR parser has the largest size among the other parsers of this family. The lookaheads are associated with the items and they make use of the left context available to the parser.
3. CLR grammar is a larger subclass of context free grammars as compared to that of SLR and LALR grammars.
4. LALR parser has the same size as that of a SLR parser but is applicable to a wider class of grammars than SLR grammar.

5. LALR is less powerful as compared to CLR, however most of the syntactic features of programming languages can be expressed in this grammar.
6. LALR parser for an erroneous input behaves differently than a CLR parser. Error detection capability is immediate in CLR but not so in LALR.
7. Both CLR and LALR parsers are expensive to construct in terms of time and space. However, efficient methods exist for constructing LALR parsers directly.

AUTOMATIC GENERATION OF THE LR PARSERS

All the parsers of this family can be generated automatically given a context free grammar as input. The parser generation tool YACC available in UNIX uses such a strategy for generating a LALR parser.

The basic steps to automate the process for generating a SLR parser from a grammar G are given below.

1. Augment the given grammar G .
2. Construct FOLLOW information for all the nonterminals of G .
3. Use the algorithms for Closure , goto and sets-of-items construction to construct the canonical collection of LR(0) sets of items.
4. Use the procedure for constructing a parsing table from the canonical collection to create the table. If the table contains conflicts, report them.
5. Supply a driver routine which uses a stack, an input buffer and the parsing table for actually parsing an input.

The other parsers can also be generated along similar lines.