LR Parsing

$LR(k)$ Grammar

An $LR(k)$ grammar is a context-free grammar where the handle in a right sentential form can be identified with a lookahead of at most k input. We shall only consider $k = 0, 1$.

$LR(0)$ Parsing

An $LR(0)$ parser can take shift-reduce decisions entirely on the basis of the states of $LR(0)$ automaton[a] of the grammar. Consider the following grammar with the augmented start symbol and the production rule.

[a]The parsing table can be filled from the automaton.

Example

The production rules are,

$$S \rightarrow aSa \mid bSb \mid c$$

The production rules of the augmented grammar are,

$$S' \rightarrow S\$$$

$$S \rightarrow aSa \mid bSb \mid c$$

The states of the $LR(0)$ automaton are the following:

States

q_0 :	$S' \rightarrow \bullet S\$$ $\quad S \rightarrow \bullet aSa$ $\quad S \rightarrow \bullet bSb$ $S \rightarrow \bullet c$
q_1 :	$S' \rightarrow S \bullet \$$
q_2 :	$S \rightarrow a \bullet Sa$ $\quad S \rightarrow \bullet aSa$ $\quad S \rightarrow \bullet bSb$ $S \rightarrow \bullet c$
q_3 :	$S \rightarrow b \bullet Sb$ $\quad S \rightarrow \bullet aSa$ $\quad S \rightarrow \bullet bSb$ $S \rightarrow \bullet c$
q_4 :	$S \rightarrow c \bullet$

States

q_5 :	$S \rightarrow aS \bullet a$
q_6 :	$S \rightarrow bS \bullet b$
q_7 :	$S \rightarrow aSa\bullet$
q_8 :	$S \rightarrow bSb\bullet$

Complete and Incomplete Items

An $LR(0)$ item is called complete if the '$\bullet$' is at the right end of the production, $A \rightarrow \alpha\bullet$. This indicates that the DFA has already 'seen' a handle and it is on the top of the stack.

$LR(0)$ Grammar

A grammar G is of type $LR(0)$ if the DFA of its viable prefixes has the following properties:

- no state has both complete and incomplete items,
- no state has two complete items.

Note

A state with a unique complete item $A \to \alpha\bullet$, indicates a reduction of the handle α by the rule $A \to \alpha$.
A state with incomplete items indicates shift actions. The parsing table for the given grammar is as follows.

Parsing table

State	Action				Goto
	a	b	c	$\$$	S
0	s_2	s_3	s_4		1
1				accept	
2	s_2	s_3	s_4		5
3	s_2	s_3	s_4		6
4	r_3	r_3	r_3	r_3	
5	s_7				
6		s_8			
7	r_1	r_1	r_1	r_1	
8	r_2	r_2	r_2	r_2	

Note

The parser does not look-ahead for any shift operation. It gets the current state from the top-of-stack and the token from the scanner. Using the parsing table it gets the next state and pushes it in the stack[a]. The token is consumed.

[a]It may push the token and its attributes in the value stack for semantic action.

Note

In case of $LR(0)$ parser it does not look-ahead even for any reduce operation[a]. It gets the current state from the top-of-stack and the production rule number from the parsing table (for all correct input they are same), and reduces the right sentential form by the rule[b].

[a] It may read the input to detect error. Note the column corresponding to 'c' for the states 4, 7, 8 with unique complete items.

[b] The Goto portion of the table is used to push a new state on the stack after a reduction.

Parsing Example

Stack	Input	Handle	Action
$\$q_0$	`aabcbaa$`	nil	s_2
$\$q_0q_2$	`abcbaa$`	nil	s_2
$\$q_0q_2q_2$	`bcbaa$`	nil	s_3
$\$q_0q_2q_2q_3$	`cbaa$`	nil	s_4
$\$q_0q_2q_2q_3q_4$	`baa$`	$S \to c$	r_3

[a]

[a] The length of $|c| = 1$, so q_4 is popped out and $\text{Goto}(q_3, S) = q_6$ is pushed in the stack.

Parsing Example

Stack	Input	Handle	Action
$\$q_0q_2q_2q_3q_4$	baa$	$S \rightarrow c$	r_3
$\$q_0q_2q_2q_3q_6$	baa$	nil	s_8
$\$q_0q_2q_2q_3q_6q_8$	aa$	$S \rightarrow bSb$	r_2
$\$q_0q_2q_2q_5$	aa$	nil	s_7
$\$q_0q_2q_2q_5q_7$	a$	$S \rightarrow aSa$	r_1

[a]

[a] The length of $|bSb| = 3$, so $q_3q_6q_8$ are popped out and Goto$(q_2, S) = q_5$ is pushed in the stack.

Parsing Example

Stack	Input	Handle	Action
$\$q_0q_2q_2q_5q_7$	a$	$S \rightarrow aSa$	r_1
$\$q_0q_2q_5$	a$	nil	s_7
$\$q_0q_2q_5q_7$	$	$S \rightarrow aSa$	r_1
$\$q_0q_1$	$	$S' \rightarrow S$	accept

[a]

[a] The length of $|aSa| = 3$, so $q_2q_5q_7$ are popped out and $\text{Goto}(q_2, S) = q_5$ is pushed in the stack. Similarly, $\text{Goto}(q_0, S) = q_1$ is pushed in the stack.

$SLR(1)$ Parsing

We consider our old grammar (augmented with S').

$$
\begin{aligned}
0: &\quad S' &\rightarrow &\quad P\$ \\
1: &\quad P &\rightarrow &\quad m\, L\, s\, e \\
2: &\quad L &\rightarrow &\quad D\, L \\
3: &\quad L &\rightarrow &\quad D \\
4: &\quad D &\rightarrow &\quad T\, V\, ; \\
5: &\quad V &\rightarrow &\quad d\, V \\
6: &\quad V &\rightarrow &\quad d \\
7: &\quad T &\rightarrow &\quad i \\
8: &\quad T &\rightarrow &\quad f
\end{aligned}
$$

States

q_0 :	$S' \rightarrow \bullet P$	$P \rightarrow \bullet m\ L\ s\ e$	
q_1 :	$S' \rightarrow P \bullet \$$		
q_2 :	$P \rightarrow m \bullet L\ s\ e$	$L \rightarrow \bullet D\ L$	$L \rightarrow \bullet D$
	$D \rightarrow \bullet T\ V\ ;$	$T \rightarrow \bullet i$	$T \rightarrow \bullet f$
q_3 :	$P \rightarrow m\ L \bullet s\ e$		
q_4 :	$L \rightarrow D \bullet L$	$L \rightarrow D \bullet$	$L \rightarrow \bullet D\ L$
	$L \rightarrow \bullet D$	$D \rightarrow \bullet T\ V\ ;$	$T \rightarrow \bullet i$
	$T \rightarrow \bullet f$		

States

q_5 :	$D \rightarrow T \bullet V\ ;$ $\quad V \rightarrow \bullet d\ V$ $\quad V \rightarrow \bullet d$
q_6 :	$T \rightarrow i \bullet$
q_7 :	$T \rightarrow f \bullet$
q_8 :	$P \rightarrow m\ L\ s \bullet e$
q_9 :	$L \rightarrow D\ L \bullet$
q_{10} :	$D \rightarrow T\ V \bullet ;$
q_{11} :	$V \rightarrow d \bullet V$ $\quad V \rightarrow d \bullet$ $\quad V \rightarrow \bullet d\ V$ $V \rightarrow \bullet d$

States

q_{12} :	$P \rightarrow m\ L\ s\ e\bullet$
q_{13} :	$D \rightarrow T\ V\ ;\bullet$
q_{14} :	$V \rightarrow d\ V\bullet$

Note

In the $LR(0)$ automaton of the grammar there are two states q_4 and q_{11} with both complete and incomplete items. So the grammar is not of type $LR(0)$.

Note

Consider the state q_4.
The complete item is $L \to D\bullet$ and the incomplete items are $T \to \bullet i$ and $T \to \bullet f$.
The Follow$(L) = \{s\}$ is different from i, f. So we can put Action$(4, i) = s_6$, Action$(4, f) = s_7$ and Action$(4, s) = r_3$ (reduce by the production rule number 3) in the parsing table.

SLR Parsing Table: Action

- If $A \rightarrow \alpha \bullet a\beta \in q_i$ $(a \in \Sigma)$ and $Goto(q_i, a) = q_j$, then Action$(i, a) = s_j$.
- If $A \rightarrow \alpha \bullet \in q_i$ $(A \neq S')$ and $b \in$ Follow(A), then Action$(i, b) = r_k$, where k is the rule number of $A \rightarrow \alpha$.
- If $S' \rightarrow S \bullet \$ \in q_i$, then Action$(i, \$) =$ accept.

Note

If this process does not lead to a table with multiple entries, then the grammar is of type SLR (simple LR).

SLR Parsing Table: Goto

If $A \rightarrow \alpha \bullet B\beta \in q_i$ $(B \in N)$ and Goto$(q_i, B) = q_j$, then in the table Goto$(i, B) = j$.
All other entries of the table are errors.

FOLLOW() Sets

Non-terminal	Follow
P	$\$$
L	s
D	i, f, s
T	d
V	$;$

SLR Parsing Table

S	*Action*								*Goto*				
	m	s	e	$;$	d	i	f	$\$$	P	L	D	V	T
0	s_2								1				
1								A					
2						s_6	s_7			3	4		5
3		s_8											
4		r_3				s_6	s_7			9	4		5
5					s_{11}							10	

Example

S	*Action*								*Goto*				
	m	s	e	$;$	d	i	f	$\$$	P	L	D	V	T
6					r_7								
7					r_8								
8				s_{12}									
9		r_2											
10				s_{13}									
11				r_6	s_{11}							14	

Example

S	$Action$								$Goto$				
	m	s	e	$;$	d	i	f	$\$$	P	L	D	V	T
12								r_1					
13		r_4				r_4	r_4						
14				r_5									

Non-SLR Grammar

Consider the following grammar G_{rr} (augmented by the S').

$$
\begin{array}{lcll}
0: & S' & \rightarrow & S\$ \\
1: & S & \rightarrow & E + T \\
2: & S & \rightarrow & T \\
3: & T & \rightarrow & i * E \\
4: & T & \rightarrow & i \\
5: & E & \rightarrow & T
\end{array}
$$

States

The states of the $LR(0)$ automaton are as follows:

q_0 :	$S' \rightarrow \bullet S$	$S \rightarrow \bullet E + T$	$S \rightarrow \bullet T$
	$E \rightarrow \bullet T$	$T \rightarrow \bullet i * E$	$T \rightarrow \bullet i$
q_1 :	$S' \rightarrow S \bullet \$$		
q_2 :	$S \rightarrow E \bullet + T$		
q_3 :	$S \rightarrow T \bullet$	$E \rightarrow T \bullet$	
q_4 :	$T \rightarrow i \bullet * E$	$T \rightarrow i \bullet$	
q_5 :	$S \rightarrow E + \bullet T$	$T \rightarrow \bullet i * E$	$T \rightarrow \bullet i$

States

q_6 :	$T \rightarrow i \; * \bullet E \quad E \rightarrow \bullet T \quad T \rightarrow \bullet i \; * \; E$ $T \rightarrow \bullet i$
q_7 :	$S \rightarrow E \; + \; T \bullet$
q_8 :	$T \rightarrow i \; * \; E \bullet$
q_9 :	$E \rightarrow T \bullet$

Note

- The state q_3 has two complete items $S \rightarrow T\bullet$ and $E \rightarrow T\bullet$.
- Also the $\text{Follow}(S) = \{\$\}$ and $\text{Follow}(E) = \{\$, +\}$ has a common element.
- So there are two conflicting reduce entries in the SLR table corresponding to the row-q_3 and the column-\$ - $\text{Action}[q_3][\$] = \{r_2, r_5\}$.

Consider the grammar G_{sr} (augmented by the S').

$$
\begin{aligned}
0: \quad S' &\rightarrow A\$ \\
1: \quad A &\rightarrow B\ a \\
2: \quad A &\rightarrow C\ b \\
3: \quad A &\rightarrow a\ C\ a \\
4: \quad C &\rightarrow B \\
5: \quad B &\rightarrow c\ A \\
6: \quad B &\rightarrow b
\end{aligned}
$$

States

Some of the states of the $LR(0)$ automaton are as follows:

q_0 :	$S' \rightarrow \bullet A$ $A \rightarrow \bullet B\ a$ $A \rightarrow \bullet C\ b$ $A \rightarrow \bullet a\ C\ a$ $B \rightarrow \bullet c\ A$ $B \rightarrow \bullet b$ $C \rightarrow \bullet B$
q_1 :	$S' \rightarrow A \bullet \$$
q_2 :	$A \rightarrow B \bullet a$ $C \rightarrow B \bullet$
q_3 :	$A \rightarrow C \bullet b$
q_4 :	$A \rightarrow a \bullet C\ a$ $C \rightarrow \bullet B$

States

q_5 :	$B \rightarrow c \bullet A$ $\quad A \rightarrow \bullet B\ a$ $\quad A \rightarrow \bullet C\ b$ $A \rightarrow \bullet a\ C\ a$ $\quad B \rightarrow \bullet c\ A$ $\quad B \rightarrow \bullet b$ $C \rightarrow \bullet B$
q_6 :	$B \rightarrow b\bullet$
q_7 :	$A \rightarrow B\ a\bullet$
q_8 :	
q_9 :	

Note

- The state q_2 has one complete item, $C \rightarrow B\bullet$ and one incomplete item, $A \rightarrow B \bullet a$.
- The SLR parsing table will have two entries for $\text{Action}[q_2][a] = \{s_7, r_4\}$, as $a \in \text{Follow}(C)$.

Note

- The grammar G_{rr} is not SLR due to the reduce/reduce conflict.
- The grammar G_{sr} is not SLR due to the shift/reduce conflict.

Note

- If the state of an $LR(0)$ automaton contains a complete item $A \to \alpha\bullet$ and the next input $a \in \text{FOLLOW}(A)$, the SLR action is reduction by the rule $A \to \alpha$.
- But in the same state if there is another complete item $B \to \beta\bullet$ with $a \in \text{Follow}(B)$, or a shift item $C \to \gamma \bullet a\mu$, there will be conflict in action.

Note

- The set FOLLOW(A) is the super set of what can follow a complete A-item at a particular state.
- In the grammar G_{rr}, in the state q_3, E cannot be followed by a $\$$. Similarly S cannot be followed by a $+$.
- Similarly in the grammar G_{sr}, in state q_2, a cannot follow the variable C.

Note

Both the reduce/reduce (G_{rr}) and shift/reduce (G_{sr}) conflicts may be resolved by explicitly carrying the look-ahead information.

Canonical $LR(1)$ Item

- An object of the form $A \rightarrow \alpha \bullet \beta, a$, where $A \rightarrow \alpha\beta$ is a production rule and $a \in \Sigma \cup \{\$\}$, is called an $LR(1)$ item.
- 'a' is called the look-ahead symbol that can follow A with this item.
- If there are more than one $LR(1)$ items with same $LR(0)$ core, we write them as $A \rightarrow \alpha \bullet \beta, a/b/\cdots$, a set.

Reduction

- The look-ahead symbols of an $LR(1)$ item $A \to \alpha \bullet \beta, L$ are important when the item is complete i.e. $\beta = \varepsilon$.
- The reduction by the rule $A \to \alpha$ can take place if the look-ahead symbol is in L of $A \to \alpha \bullet, L$.
- The look-ahead set L is a subset of FOLLOW(A), but we carry them explicitly to resolve more conflicts.

Valid Item

An $LR(1)$ item $A \rightarrow \alpha \bullet \beta, a$ is valid for a viable prefix '$u\alpha$', if there is a rightmost derivation: $S \rightarrow uAx \rightarrow u\alpha\beta x$, so that $a \in \text{FIRST}(x)$ or if $x = \varepsilon$, then $a = \$$.

Closure()

If i is an $LR(1)$ item, then Closure(i) is defined as follows:

- $i \in$ Closure(i) - basis,
- If $(A \rightarrow \alpha \bullet B\beta, a) \in$ Closure(i) and $B \rightarrow \gamma$ is a production rule, then $(B \rightarrow \bullet\gamma, b) \in$ Closure(i), where $b \in$ FIRST(βa).

Closure()

The closure of I, a set of $LR(1)$ items, is defined as $\text{Closure}(I) = \bigcup_{i \in I} \text{Closure}(i)$.

Goto(I, X)

Let I be a set of $LR(1)$ items and $X \in \Sigma \cup N$. The set of $LR(1)$ items Goto(I, X) is

$$\text{Closure}\left(\{(A \rightarrow \alpha\ X \bullet \beta, a) :\ (A \rightarrow \alpha \bullet X\ \beta, a) \in I\}\right).$$

$LR(1)$ Automaton

The start state of the $LR(1)$ automaton is Closure($S' \rightarrow \bullet S, \$$). Other reachable and final states can be constructed by computing GOTO() of already existing states. This is a fixed-point computation.

Consider the grammar G_{rr} (augmented by the S').

$$
\begin{aligned}
0: \quad S' &\rightarrow S\$ \\
1: \quad S &\rightarrow E + T \\
2: \quad S &\rightarrow T \\
3: \quad T &\rightarrow i * E \\
4: \quad T &\rightarrow i \\
5: \quad E &\rightarrow T
\end{aligned}
$$

States

The states of the $LR(1)$ automaton are as follows:

q_0 :	$S' \rightarrow \bullet S, \$$	$S \rightarrow \bullet E + T, \$$	$S \rightarrow \bullet T, \$$
	$E \rightarrow \bullet T, +$	$T \rightarrow \bullet i * E, +/\$$	$T \rightarrow \bullet i, +/\$$
q_1 :	$S' \rightarrow S \bullet, \$$		
q_2 :	$S \rightarrow E \bullet + T, \$$		
q_3 :	$S \rightarrow T \bullet, \$$	$E \rightarrow T \bullet, +$	
q_4 :	$T \rightarrow i \bullet * E, +/\$$	$T \rightarrow i \bullet, +/\$$	
q_5 :	$S \rightarrow E + \bullet T, \$$	$T \rightarrow \bullet i * E, \$$	$T \rightarrow \bullet i, \$$

States

q_6 :	$T \rightarrow i \ * \bullet E, +/\$$ $E \rightarrow \bullet T, +/\$$ $T \rightarrow \bullet i \ * \ E, +/\$$ $T \rightarrow \bullet i, +/\$$
q_7 :	$S \rightarrow E \ + \ T \bullet, \$$
q_8 :	$T \rightarrow i \bullet * \ E, \$$ $T \rightarrow i \bullet, \$$
q_9 :	$T \rightarrow i \ * \ E \bullet, +/\$$
q_{10} :	$E \rightarrow T \bullet, +/\$$
q_{11} :	$T \rightarrow i \ * \bullet E, \$$ $E \rightarrow \bullet T, \$$ $T \rightarrow \bullet i \ * \ E, \$$ $T \rightarrow \bullet i, \$$

States

$q_{12}:$	$T \rightarrow i\ *\ E\bullet, \$$
$q_{13}:$	$E \rightarrow T\bullet, \$$

Note

Number of states of the $LR(1)$ automaton are more than that of $LR(0)$ automaton.
Several states have the same core $LR(0)$ items, but different look-ahead symbols - (q_4, q_8), (q_6, q_{11}), (q_9, q_{12}).

$LR(1)$ Parsing Table: Action

- If $(A \rightarrow \alpha \bullet a\beta, b) \in q_i$ $(a \in \Sigma)$ and $Goto(q_i, a) = q_j$, then Action$(i, a) = s_j$.
- If $(A \rightarrow \alpha \bullet, a) \in q_i$ $(A \neq S')$, then Action$(i, a) = r_k$, where k is the rule number of $A \rightarrow \alpha$.
- If $(S' \rightarrow S \bullet, \$) \in q_i$, then Action$(i, \$) =$ accept.

$LR(1)$ Parsing Table: Goto

If $A \rightarrow \alpha \bullet B\beta \in q_i$ ($B \in N$) and Goto$(q_i, B) = q_j$, then in the table Goto$(i, B) = j$.
All other entries of the table are errors.

Note

If the process constructs a table **without multiple entries**, the grammar is $LR(1)$.

$LR(1)$ Parsing Table

S	*Action*				*Goto*		
	$+$	$*$	i	$\$$	S	E	T
0			s_4		1	2	3
1				A			
2	s_5						
3	r_5			r_2			
4	r_4	s_6		r_4			
5			s_8				7
6			s_4		9		10

$LR(1)$ Parsing Table

S	*Action*				*Goto*		
	$+$	$*$	i	$\$$	S	E	T
7				r_1			
8		s_{11}		r_4			
9	r_3			r_3			
10	r_5			r_5			
11			s_8			12	13
12				r_3			
13				r_5			

Non-$LR(1)$ Grammar

$$
\begin{aligned}
0: \quad S &\rightarrow A \\
1: \quad A &\rightarrow a\ A\ a \\
2: \quad A &\rightarrow a\ Aa\ a\ b \\
3: \quad A &\rightarrow a\ b
\end{aligned}
$$

States of $LR(1)$ Automaton

q_0 :	$S \rightarrow \bullet A, \$$ $A \rightarrow \bullet aAa, \$$ $A \rightarrow \bullet aAaab, \$$ $A \rightarrow \bullet ab, \$$
q_1 :	$S \rightarrow A \bullet, \$$
q_2 :	$A \rightarrow a \bullet Aa, \$$ $A \rightarrow a \bullet Aaab, \$$ $A \rightarrow a \bullet b, \$$ $A \rightarrow \bullet aAa, a$ $A \rightarrow \bullet aAaab, a$ $A \rightarrow \bullet ab, a$
q_3 :	$A \rightarrow aA \bullet a, \$$ $A \rightarrow aA \bullet aab, \$$
q_4 :	$A \rightarrow ab \bullet, \$$
q_5 :	$A \rightarrow a \bullet Aa, a$ $A \rightarrow a \bullet Aaab, a$ $A \rightarrow a \bullet b, a$ $A \rightarrow \bullet aAa, a$ $A \rightarrow \bullet aAaab, a$ $A \rightarrow \bullet ab, a$

States of $LR(1)$ Automaton

q_6 :	$A \to aAa\bullet, \$$	$A \to aAa \bullet ab, \$$
q_7 :	$A \to aA \bullet a, a$	$A \to aA \bullet aab, a$
q_8 :	$A \to ab\bullet, a$	
q_9	$A \to aAaa \bullet b, \$$	
q_{10} :	$A \to aAa\bullet, a$	$A \to aAa \bullet ab, a$
q_{11}	$A \to aAaab\bullet, \$$	

Note

In state q_{10}, the shift/reduce conflict cannot be resolved and there will be multiple entries in $\text{Action}(10, a) = \{s_i, r_1\}$, where $\text{Goto}(q_{10}, a) = q_i$. This can be resolved with 2-look-ahead

States of $LR(2)$ Automaton

q_0 :	$S \rightarrow \bullet A, \$$	$A \rightarrow \bullet aAa, \$$	$A \rightarrow \bullet aAaab, \$$
	$A \rightarrow \bullet ab, \$$		
q_1 :	$S \rightarrow A\bullet, \$$		
q_2 :	$A \rightarrow a \bullet Aa, \$$	$A \rightarrow a \bullet Aaab, \$$	$A \rightarrow a \bullet b, \$$
	$A \rightarrow \bullet aAa, aa/a\$$	$A \rightarrow \bullet aAaab, aa/a\$$	$A \rightarrow \bullet ab, aa/a\$$
q_3 :	$A \rightarrow aA \bullet a, \$$	$A \rightarrow aA \bullet aab, \$$	
q_4 :	$A \rightarrow ab\bullet, \$$		
q_5 :	$A \rightarrow a \bullet Aa, aa/a\$$	$A \rightarrow a \bullet Aaab, aa/a\$$	$A \rightarrow a \bullet b, aa/a\$$
	$A \rightarrow \bullet aAa, aa$	$A \rightarrow \bullet aAaab, aa$	$A \rightarrow \bullet ab, aa$

States of $LR(2)$ Automaton

q_6 :	$A \to aAa\bullet, \$$	$A \to aAa \bullet ab, \$$
q_7 :	$A \to aA \bullet a, aa/a\$$	$A \to aA \bullet aab, aa/a\$$
q_8 :	$A \to ab\bullet, aa/a\$$	
q_9	$A \to aAaa \bullet b, \$$	
q_{10} :	$A \to aAa\bullet, aa/a\$$	$A \to aAa \bullet ab, aa/a\$$
q_{11}	$A \to aAaab\bullet, \$$	

Note

In state q_{10}, the action is r_1 if the next two symbols are either '`aa`' or '`a$`'. The action is shift if they are '`ab`'. But we shall not use $LR(2)$ parsing.

LALR Parser

- There are pairs of $LR(1)$ states for the grammar G_{rr} with the same $LR(0)$ items. These are (q_4, q_8), (q_6, q_{11}), (q_9, q_{12}).
- If we can merge states with the same $LR(0)$ items, the number of states of the automaton will be same as that of $LR(0)$ automaton.

LALR Parser

- For some $LR(1)$ grammar this merging will not lead to multiple entries in the parsing table.
- Such a grammar is known as LALR(1) (lookahead LR) grammar.

Note

- Merging of two $LR(1)$ states with the same $LR(0)$ item cannot give rise to a new shift/reduce conflict.
- If there is a pair of items of the form $\{A \rightarrow \alpha \bullet a\beta, \cdots, B \rightarrow \gamma\bullet, a\}$ in the merged state, it is already there in some $LR(1)$ state.
- So the grammar is not even $LR(1)$.

Note

- Two states of an LALR parser cannot have the same set of $LR(0)$ items.
- So the number of states of an $LR(0)$ and an LALR(1) automaton are same.
- An LALR parser uses a better heuristic, than the global FOLLOW() sets of non-terminals, about symbols that can follow an $LR(0)$ item at a state.

LALR States

The states of the $LR(1)$ automaton are as follows:

q_0 :	$S' \rightarrow \bullet S, \$$	$S \rightarrow \bullet E + T, \$$	$S \rightarrow \bullet T, \$$
	$E \rightarrow \bullet T, +$	$T \rightarrow \bullet i * E, +/\$$	$T \rightarrow \bullet i, +/\$$
q_1 :	$S' \rightarrow S \bullet, \$$		
q_2 :	$S \rightarrow E \bullet + T, \$$		
q_3 :	$S \rightarrow T \bullet, \$$	$E \rightarrow T \bullet, +$	
$q_{4 \cdot 8}$:	$T \rightarrow i \bullet * E, +/\$$	$T \rightarrow i \bullet, +/\$$	
q_5 :	$S \rightarrow E + \bullet T, \$$	$T \rightarrow \bullet i * E, \$$	$T \rightarrow \bullet i, \$$

States

$q_{6\cdot 11}$:	$T \rightarrow i\ * \bullet E, +/\$$ $E \rightarrow \bullet T, +/\$$ $T \rightarrow \bullet i\ *\ E, +/\$$ $T \rightarrow \bullet i, +/\$$
q_7 :	$S \rightarrow E\ +\ T\bullet, \$$
$q_{9\cdot 12}$:	$T \rightarrow i\ *\ E\bullet, +/\$$
q_{10} :	$E \rightarrow T\bullet, +/\$$
q_{13} :	$E \rightarrow T\bullet, \$$

LALR Parsing Table

S	*Action*				*Goto*		
	$+$	$*$	i	$\$$	S	E	T
0			$s_{4\cdot 8}$		1	2	3
1				A			
2	s_5						
3	r_5			r_2			
$4 \cdot 8$	r_4	$s_{6\cdot 11}$		r_4			
5			$s_{4\cdot 8}$				7
$6 \cdot 11$			$s_{4\cdot 8}$		$9 \cdot 12$		10

LALR Parsing Table

S	*Action*				*Goto*		
	$+$	$*$	i	$\$$	S	E	T
7				r_1			
$9 \cdot 12$	r_3			r_3			
10	r_5			r_5			
13				r_5			

$LR(1)$ but not LALR

Consider the grammar

$$
\begin{array}{llll}
0: & S & \rightarrow & A\$ \\
1: & A & \rightarrow & a\ B\ a \\
2: & A & \rightarrow & b\ B\ b \\
3: & A & \rightarrow & a\ D\ b \\
4: & A & \rightarrow & b\ D\ a \\
5: & B & \rightarrow & c \\
6: & D & \rightarrow & c
\end{array}
$$

States of $LR(1)$ Automaton

q_0 :	$S \to \bullet A, \$$ $A \to \bullet aBa, \$$ $A \to \bullet bBb, \$$ $A \to \bullet aDb, \$$ $A \to \bullet bDa, \$$
q_1 :	$S \to A\bullet, \$$
q_2 :	$A \to a \bullet Ba, \$$ $A \to a \bullet Db, \$$ $B \to \bullet c, a$ $D \to \bullet c, b$
q_3 :	$A \to b \bullet Bb, \$$ $A \to b \bullet Da, \$$ $B \to \bullet c, b$ $D \to \bullet c, a$
q_4 :	$A \to aB \bullet a, \$$
q_5 :	$A \to aD \bullet b, \$$

States of $LR(1)$ Automaton

q_6 :	$B \rightarrow c\bullet, a$	$D \rightarrow c\bullet, b$
q_7 :	$A \rightarrow bB \bullet b, \$$	
q_8 :	$A \rightarrow bD \bullet a, \$$	
q_9 :	$B \rightarrow c\bullet, b$	$D \rightarrow c\bullet, a$

The states q_6 and q_9 have the same $LR(0)$ core, but they cannot be merged to form a LALR state as that will lead to reduce/reduce conflicts. So the grammar is $LR(1)$ but not LALR.

Resolving Shift-Reduce Conflicts

- Take longest sequence of handle for reduction i.e. shift when there is a shift/reduce conflict e.g. associate the `else` to the nearest `if`.
- In an operator grammar use the associativity and precedence of operators. As an example $A \rightarrow \alpha \bullet \otimes \beta$, $B \rightarrow \gamma \oplus \mu \bullet$. 'shift' if $\otimes$ is of higher precedence, reduce is $\oplus$ is of higher precedence etc.

Resolving Reduce-Reduce Conflicts

- There are two or more complete items in a state.
- It is often resolved using the first grammar rule of complete items.
- But it may not give a satisfactory result. Consider the following grammar. The terminals are {`i`, `f`, `id`}. The start symbol is D.

Resolving Reduce-Reduce Conflicts

1,2	D	$\rightarrow$	ID | FD
3	ID	$\rightarrow$	IS i
4	FD	$\rightarrow$	FS f
5,6	IS	$\rightarrow$	IS IV | IV
7,8	FS	$\rightarrow$	FS FV | FV
9	IV	$\rightarrow$	id
10	FV	$\rightarrow$	id

Resolving Reduce-Reduce Conflicts

- The state $IV \rightarrow id\bullet, FV \rightarrow id\bullet$ has a reduce-reduce conflict.
- But resolving it to reduct by rule 9 is unacceptable.

Ambiguous Grammar & LR Parsing

An ambiguous grammar cannot be LR. But for some ambiguous grammars[a] it is possible to use LR parsing techniques efficiently with the help of some extra grammatical information such as associativity and precedence of operators.

[a]As an example for operator-precedence grammars: CFG with no ε production and no production rule with two non-terminals coming side by side.

Example

Consider the expression grammar G_a

$$
\begin{array}{rccl}
0: & S & \rightarrow & E\$ \\
1: & E & \rightarrow & E - E \\
2: & E & \rightarrow & E * E \\
3: & E & \rightarrow & (E) \\
4: & E & \rightarrow & -E \\
5: & E & \rightarrow & i
\end{array}
$$

Note that the terminal '-' is used both as binary as well as unary operator.

States of $LR(0)$ Automaton

q_0 :	$S \rightarrow \bullet E$	$E \rightarrow \bullet E - E$	$E \rightarrow \bullet E * E$
	$E \rightarrow \bullet(E)$	$E \rightarrow \bullet - E$	$E \rightarrow \bullet i$
q_1 :	$S \rightarrow E \bullet \$$	$E \rightarrow E \bullet -E$	$E \rightarrow E \bullet *E$
q_2 :	$E \rightarrow (\bullet E)$	$E \rightarrow \bullet E - E$	$E \rightarrow \bullet E * E$
	$E \rightarrow \bullet(E)$	$E \rightarrow \bullet - E$	$E \rightarrow \bullet i$
q_3 :	$E \rightarrow - \bullet E$	$E \rightarrow \bullet E - E$	$E \rightarrow \bullet E * E$
	$E \rightarrow \bullet(E)$	$E \rightarrow \bullet - E$	$E \rightarrow \bullet i$
q_4 :	$E \rightarrow i \bullet$		

States of $LR(0)$ Automaton

q_5 :	$E \to E - \bullet E$	$E \to \bullet E - E$	$E \to \bullet E * E$
	$E \to \bullet(E)$	$E \to \bullet - E$	$E \to \bullet i$
q_6 :	$E \to E * \bullet E$	$E \to \bullet E - E$	$E \to \bullet E * E$
	$E \to \bullet(E)$	$E \to \bullet - E$	$E \to \bullet i$
q_7 :	$E \to (E\bullet)$	$E \to E \bullet -E$	$E \to E \bullet *E$
q_8 :	$E \to -E\bullet$	$E \to E \bullet -E$	$E \to E \bullet *E$
q_9 :	$E \to E - E\bullet$	$E \to E \bullet -E$	$E \to E \bullet *E$
q_{10} :	$E \to E * E\bullet$	$E \to E \bullet -E$	$E \to E \bullet *E$

There are a few more states.

Note

The states q_8, q_9 and q_{10} have complete and incomplete items. FOLLOW$(E) = \{\$, -, *,)\}$ cannot resolve the conflict. In fact no amount of look-ahead can help - the $LR(1)$ initial state is

q_0 :	$S \rightarrow \bullet E, \$$	$E \rightarrow \bullet E - E, \$/-/*$	$E \rightarrow \bullet E * E, \$/-/*$
	$E \rightarrow \bullet (E), \$/-/*$	$E \rightarrow \bullet - E, \$/-/*$	$E \rightarrow \bullet i, \$/-/*$

$q_8 : E \rightarrow -E\bullet,\ E \rightarrow E \bullet -E,\ E \rightarrow E \bullet *E$

The higher precedence of unary '$-$' over the binary '$-$' and binary '$*$' will help to resolve the conflict. The parser reduces the handle i.e. $\text{Action}(8, -) = \text{Action}(8, *) = \text{Action}(8,)) = \text{Action}(8, \$) = r_4$.

$q_9 : E \to E - E\bullet,\ E \to E \bullet -E,\ E \to E \bullet *E$

In this case if the look-ahead symbol is a '$-$' (it must be binary), the parser reduces due to the left associativity of binary '$-$'. But if the look-ahead symbol is a '$*$', the parser shifts i.e. $\text{Action}(9, -) = \text{Action}(9,)) = \text{Action}(9, \$) = r_4$ but $\text{Action}(9, *) = s_6$.

$$q_{10} : E \rightarrow E * E\bullet,\ E \rightarrow E \bullet -E,\ E \rightarrow E \bullet *E$$

Actions are always reduce.

Error Handling

- What happens when an (LA)LR(1)-parser is in state q, the input token is a, and the parsing table entry $\text{Action}(q, a)$ is empty i.e. no-shift, no-reduce, no-accept. This is an error condition.
- The token a is not a valid continuation of the input the parses has seen so far.

Error Handling

- The question is what action should the parser take.
- The simplest solution is highlight the position of error, and terminate parsing.
- But the error may be due to a missing semicolon (';') or a parenthesis ('(') only. After fixing it the compilation is to be restarted from the beginning.

Error Handling

- Another alternative may be to continue with the error. But then the parser will start generating dozens of spurious errors due to the single error.
- So the parser needs to recover from the current error and try to detect as many errors as possible in a pass.

Error Handling

- An error recovery strategy may try to modify either the stack or the input stream or both.
- Modification of the stack amounts to modification of a portion of the parse tree that has already been constructed and found to be correct.

Panic-Mode Error Recovery

- The parsing stack is scanned so that a state q with a Goto on a non-terminal A is found.
- A few input tokens are discarded until a token $b \in \mathsf{Follow}(A)$ is found in the input stream.
- The state $\mathrm{Goto}(A, b)$ is pushed in the stack and the normal parsing is resumed.

Panic-Mode Error Recovery

- The recovery works under the assumption that the error is within the string generated by A (within the phrase of A).
- The non-terminal A may represent an expression, where an operator or an operand is missing; or a statement, where a semicolon or an `end` is missing.

Embedding Error Actions in Parsing table

- Phrase-level recovery routines can be embedded in the (LA)LR(1) parsing table.
- Each error entry may be a pointer to the corresponding error-handling routine.
- The error-handling routine should not drive the parses in an infinite loop.

We consider our old grammar.

$$
\begin{aligned}
0: &\quad S' &\rightarrow&\quad P\$ \\
1: &\quad P &\rightarrow&\quad m\,L\,s\,e \\
2: &\quad L &\rightarrow&\quad D\,L \\
3: &\quad L &\rightarrow&\quad D \\
4: &\quad D &\rightarrow&\quad T\,V\,; \\
5: &\quad V &\rightarrow&\quad d\,V \\
6: &\quad V &\rightarrow&\quad d \\
7: &\quad T &\rightarrow&\quad i \\
8: &\quad T &\rightarrow&\quad f
\end{aligned}
$$

States

q_0 :	$S' \rightarrow \bullet P$	$P \rightarrow \bullet m\ L\ s\ e$	
q_1 :	$S' \rightarrow P \bullet \$$		
q_2 :	$P \rightarrow m \bullet L\ s\ e$	$L \rightarrow \bullet D\ L$	$L \rightarrow \bullet D$
	$D \rightarrow \bullet T\ V\ ;$	$T \rightarrow \bullet i$	$T \rightarrow \bullet f$
q_3 :	$P \rightarrow m\ L \bullet s\ e$		
q_4 :	$L \rightarrow D \bullet L$	$L \rightarrow D \bullet$	$L \rightarrow \bullet D\ L$
	$L \rightarrow \bullet D$	$D \rightarrow \bullet T\ V\ ;$	$T \rightarrow \bullet i$
	$T \rightarrow \bullet f$		

States

$q_5:$	$D \rightarrow T \bullet V\ ;$ $\quad V \rightarrow \bullet d\ V$ $\quad V \rightarrow \bullet d$
$q_6:$	$T \rightarrow i \bullet$
$q_7:$	$T \rightarrow f \bullet$
$q_8:$	$P \rightarrow m\ L\ s \bullet e$
$q_9:$	$L \rightarrow D\ L \bullet$
$q_{10}:$	$D \rightarrow T\ V \bullet ;$
$q_{11}:$	$V \rightarrow d \bullet V$ $\quad V \rightarrow d \bullet$ $\quad V \rightarrow \bullet d\ V$ $V \rightarrow \bullet d$

States

q_{12} :	$P \rightarrow m\ L\ s\ e\bullet$
q_{13} :	$D \rightarrow T\ V\ ;\bullet$
q_{14} :	$V \rightarrow d\ V\bullet$

Modified SLR Table

- All error entries of a state with reduction action are replaced by the same reduction action.
- This delays the error detection.

Error Routine - 0 (e_0)

- The parser is in a state i ($i \neq 1$) and it encounters the eof ($).
- Terminate parsing with a message 'unexpected `<eof>`'

Error Routine - 1 (e_1)

- The parser expects m at state 0 and the $\text{Action}(0, m) = 2$.
- If it encounters any other symbol at state 0, it pushes state 2 in the stack and generates error message '`m missing`'.

Error Routine - 2 (e_2)

- At state 1 the parser has already seen a valid input.
- If it sees anything other than eof ($), it may accept the input and generate the error message 'extra character' at the end of input.

Error Routine - 3 (e_3)

- At state 2 if there is anything other than i, f or \$, the parser push either state 6 or state 7 in the stack (does not matter as it is an error condition).
- It prints 'missing i or f'.

Error Routine - 4 (e_4)

- At state 3 if there is anything other than s or \$, the parser pushes state 8 in the stack.
- It prints ‘missing s’.

State - 4

- Error entries of state 4 are filled with reduction by rule 3 (r_3).
- The reduction takes place and the error detection is deferred.

Note

- Similarly we fill other error entries.
- The question is, whether there is any possibility of infinite loop.

SLR Parsing Table

S	*Action*								*Goto*				
	m	s	e	$;$	d	i	f	$\$$	P	L	D	V	T
0	s_2	e_1	e_1	e_1	e_1	e_1	e_1	e_0	1				
1	e_2	e_2	e_2	e_2	e_2	e_2	e_2	A					
2	e_3	e_3	e_3	e_3	e_3	s_6	s_7	e_0		3	4		5
3	e_4	s_8	e_4	e_4	e_4	e_4	e_4	e_0					
4	r_3	r_3	r_3	r_3	r_3	s_6	s_7	r_3		9	4		5
5	e_5	e_5	e_5	e_5	s_{11}	e_5	e_5	e_0				10	

Example

S	*Action*								*Goto*				
	m	s	e	;	d	i	f	$	P	L	D	V	T
6	r_7	r_7	r_7	r_7	r_7	r_7	r_7	r_7					
7	r_8	r_8	r_8	r_8	r_8	r_8	r_8	r_8					
8	e_6	e_6	e_6	s_{12}	e_6	e_6	e_6	e_0					
9	r_2	r_2	r_2	r_2	r_2	r_2	r_2	r_2					
10	e_7	e_7	e_7	s_{13}	e_7	e_7	e_7	e_0					
11	r_6	r_6	r_6	r_6	s_{11}	r_6	r_6	r_6				14	

Example

S	*Action*								*Goto*				
	m	s	e	$;$	d	i	f	$\$$	P	L	D	V	T
12	r_1	r_1	r_1	r_1	r_1	r_1	r_1	r_1					
13	r_4	r_4	r_4	r_4	r_4	r_4	r_4	r_4					
14	r_5	r_5	r_5	r_5	r_5	r_5	r_5	r_5					

Error Recovery in Yacc

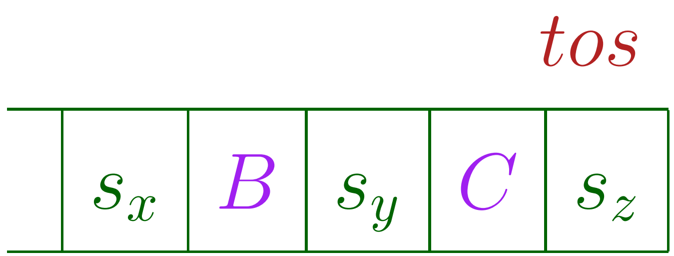
- Some non-terminals are chosen for error recovery.
- Such a non-terminal A has an added special production rule $A \rightarrow$ errRec to create a dummy node.
- If a syntax error is detected while constructing a subtree rooted at A, two actions are taken.

Error Recovery in Yacc

- Tokens from the input expected to reduce to A are discarded.
- A dummy node is created for A.
- Let the production rules of A be $A \rightarrow BCD \mid \text{errRec}$.

Error Recovery in Yacc

- The state at the top of stack is s_z and the current token is a. But Action(s_z, a) is empty, an error.
- Let the sequence of states and non-terminals at the top of stack are as follows.



Error Recovery in Yacc

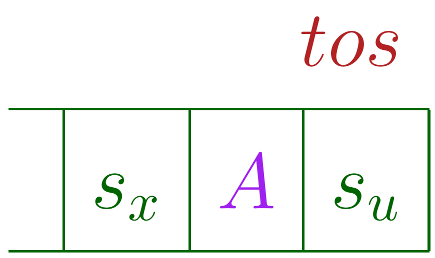
- The set of valid items for the state s_x are $\{X \to \alpha \bullet A\beta,\ A \to \bullet BCD,\ A \to \bullet\ \text{errRec},\ B \to \cdots\}$

  The sub-trees corresponding to B and C have already been constructed.

- Element from the top of the stack are removed to get the error recovery state (s_x), which has a Goto() on an error recovery non-terminal (A).

Error Recovery in Yacc

- A dummy node for A with errRec is created. Then A and Goto(s_x, A) are pushed in the stack.
- The top of stack looks like,



Error Recovery in Yacc

- The valid items of s_u are

$$\{X \rightarrow \alpha A \bullet \beta, \cdots\}$$

- Tokens from the input stream are discarded until there is a token b such that Action(s_u, b) is non-empty, not an error.
- This process cannot go to an infinite loop as there must be some Action() at the state s_u.

References

[ASRJ] Compilers Principles, Techniques, and Tools, by A. V. Aho, Monica S. Lam, R. Sethi, & J. D. Ullman, 2nd ed., ISBN 978-81317-2101-8, Pearson Ed., 2008.

[DKHJK] Modern Compiler Design, by Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J. H. Jacobs, Koen Langendoen, 2nd ed,, ISBN 978 1461 446989, Springer (2012).

[KL] Engineering a Compiler, by Keith D. Cooper & Linda Troczon, (2nd ed.), ISBN 978-93-80931-87-6, Morgan Kaufmann, Elsevier, 2012.