CS31003: Compilers

Syntax Analysis or Parsing

Lecture: 09-14

LR Parsers

Shift-Reduce Parser: Example: Grammar

Sample grammar G_1:

$$
\begin{array}{llll}
1: & E & \rightarrow & E + T \\
2: & E & \rightarrow & T \\
3: & T & \rightarrow & T * F \\
4: & T & \rightarrow & F \\
5: & F & \rightarrow & (E) \\
6: & F & \rightarrow & \textbf{id}
\end{array}
$$

Shift-Reduce Parser: Example: Parse Table

State	Action						GO TO		
	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			

Shift-Reduce Parser: Example: Parsing **id** * **id** + **id**

Step	Stack	Symbols	Input	Action
(1)	0		**id** * **id** + **id** $	shift
(2)	0 5	**id**	* **id** + **id** $	reduce by $F \to$ **id**
(3)	0 3	F	* **id** + **id** $	reduce by $T \to F$
(4)	0 2	T	* **id** + **id** $	shift
(5)	0 2 7	T *	**id** + **id** $	shift
(6)	0 2 7 5	T * **id**	+ **id** $	reduce by $F \to$ **id**
(7)	0 2 7 10	$T * F$	+ **id** $	reduce by $T \to T * F$
(8)	0 2	T	+ **id** $	reduce by $E \to T$
(9)	0 1	E	+ **id** $	shift
(10)	0 1 6	E +	**id** $	shift
(11)	0 1 6 5	E + **id**	$	reduce by $F \to$ **id**
(12)	0 1 6 3	$E + F$	$	reduce by $T \to F$
(13)	0 1 6 9	$E + T$	$	reduce by $E \to E + T$
(14)	0 1	E	$	accept

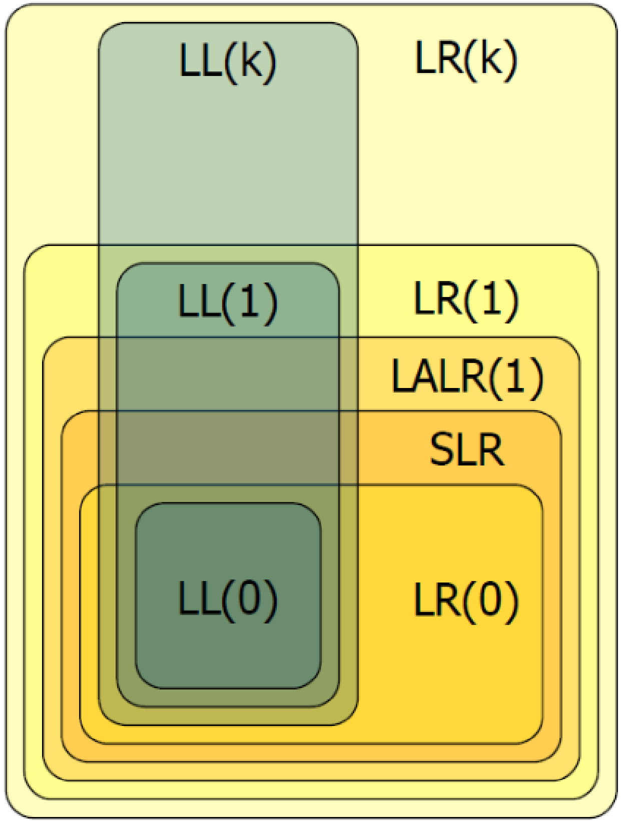
$$
\begin{aligned}
E\ \$ &\Rightarrow \underline{E + T}\ \$ \\
&\Rightarrow E + \underline{F}\ \$ \\
&\Rightarrow E + \underline{\mathbf{id}}\ \$ \\
&\Rightarrow \underline{T} + \mathbf{id}\ \$ \\
&\Rightarrow \underline{T * F} + \mathbf{id}\ \$ \\
&\Rightarrow T * \underline{\mathbf{id}} + \mathbf{id}\ \$ \\
&\Rightarrow \underline{F} * \mathbf{id} + \mathbf{id}\ \$ \\
&\Rightarrow \underline{\mathbf{id}} * \mathbf{id} + \mathbf{id}\ \$
\end{aligned}
$$

State	Action						GO TO		
	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			

LR Parsers

LR Fundamentals

LR Parsing: CFG Classes



- **LL(k), Top-Down, Predictive**: LL parser (Left-to-right, Leftmost derivation) with k look-ahead
- **LR(k), Bottom-Up, Shift-Reduce**: LR parser (Left-to-right, Rightmost derivation) with k look-ahead

LR Parsers

- LR parser (Left-to-right, Rightmost derivation in reverse)
- Reads input text from left to right without backing up
- Produces a rightmost derivation in reverse
- Performs bottom-up parse
- To avoid backtracking or guessing, an LR(k) parser peeks ahead at k look-ahead symbols before deciding how to parse earlier symbols. Typically k is 1.
- LR parsers are deterministic – produces a single correct parse without guesswork or backtracking
- Works in linear time
- Variants of LR parsers and generators:
  - LP(0) Parsers
  - SLR Parsers
  - LALR Parsers – Generator: Yacc (AT & T), Byacc (Berkeley Yacc)
  - Canonical LR(1) Parsers – Generator: Bison (GNU)
  - Minimal LR(1) Parsers – Generator: Hyacc (Hawaii Yacc)
  - GLR Parsers – Generator: Bison (GNU) with `%glr-parser` declaration
- Minimal LR and GLR parsers have better memory performance CLR Parsers and address reduce/reduce conflicts more effectively

LR Parsers

- An LR parser is a DPDA having:
  - An Input Buffer
  - A Stack of Symbols – terminals as well as non-terminals
  - A DFA that starts on the first state and has four types of actions:
    - **Shift** – Target state on input symbol
    - **Reduce** – Production rule and Target state on non-terminal on reduction (GOTO actions)
    - **Accept** – Successful termination of parsing
    - **Reject** – Failure termination of parsing
- Designing an LR Parser is all about designing its DFA and actions

FIRST and FOLLOW

- $FIRST(\alpha)$, where α is any string of grammar symbols, is defined to be the set of terminals that begin strings derived from α. If $\alpha \Rightarrow^* \epsilon$, then ϵ is also in $FIRST(\alpha)$. *Examples*:
  - Given $S \to 0|A$, $A \to AB|1$, $B \to 2$;
    $FIRST(B) = \{2\}$, $FIRST(A) = \{1\}$, $FIRST(S) = \{0, 1\}$
  - Given $E \to E + E|E * E|(E)|\mathbf{id}$;
    $FIRST(E) = \{\mathbf{id}, (\}$
  - Given $B \to A, A \to Ac|Aad|bd|\epsilon$;
    $FIRST(B) = FIRST(A) = \{\epsilon, a, b, c\}$
- $FOLLOW(A)$, for non-terminal A, is defined to be the set of terminals a that can appear immediately to the right of A in some sentential form; that is, the set of terminals a such that there exists a derivation of the form $S \Rightarrow^* \alpha A a \beta$, for some α and β. \$ can also be in the $FOLLOW(A)$. *Examples*:
  - Given $E \to E + E|E * E|(E)|\mathbf{id}$;
    $FOLLOW(E) = \{+, *,), \$\}$
  - Given $B \to A, A \to Ac|Aad|bd|\epsilon$;
    $FOLLOW(B) = \{\$\}$, $FOLLOW(A) = \{a, c, \$\}$

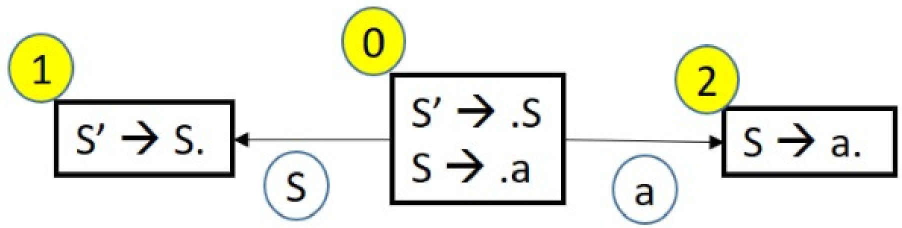
LR Parsers

LR(0) Parser

Intuitive LR Parser Construction

- $G_3 = \{S \to a\}$

LR(0) Parser:
S’ → S
S → a
L(G) = {a}



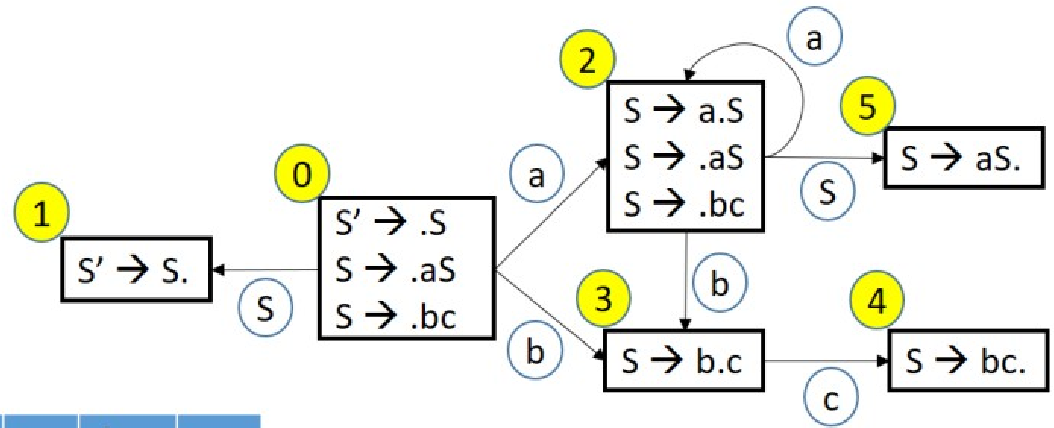
State	a	$	S
0	s2		1
1		Acc	
2	r1	r1	

S’ → S → a$

Intuitive LR Parser Construction

- $G_4 = \{S \rightarrow aS|bc\}$

LR(0) Parser:
S’ → S
S → aS | bc
L(G) = {a*bc}

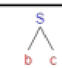
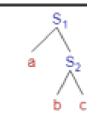
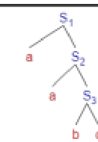
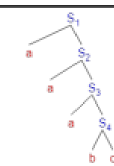
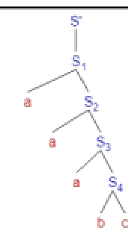


State	a	b	c	$	S
0	s2	s3			1
1				Acc	
2	s2	s3			5
3			s4		
4	r2	r2	r2	r2	
5	r1	r1	r1	r1	

S → bc$
S → aS$ → abc$
S → aS$ → aaS$ → aabc$

Intuitive LR Parser Construction

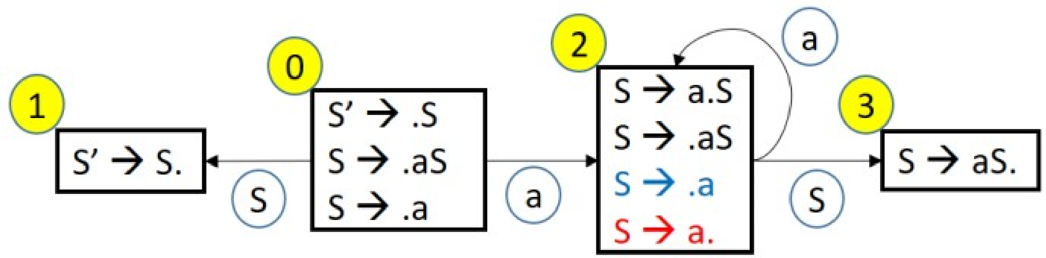
$G_4 = \{ S \rightarrow \mathbf{a}\ S \mid \mathbf{b}\ \mathbf{c} \}$. $S'\ \$ \Rightarrow S\ \$ \Rightarrow \mathbf{a}\ S\ \$ \Rightarrow \mathbf{a}\ \mathbf{a}\ S\ \$ \Rightarrow \mathbf{a}\ \mathbf{a}\ \mathbf{a}\ S\ \$ \Rightarrow \mathbf{a}\ \mathbf{a}\ \mathbf{a}\ \mathbf{b}\ \mathbf{c}\ \$$

Step	Stack	Symbols	Input	Action	Parse Tree
(1)	0		**a a a b c $**	shift	
(2)	0 2	**a**	**a a b c $**	shift	
(3)	0 2 2	**a a**	**a b c $**	shift	
(4)	0 2 2 2	**a a a** S	**b c $**	shift	
(5)	0 2 2 2 3	**a a a b**	**c $**	shift	
(6)	0 2 2 2 3 4	**a a a** $\underline{\mathbf{b}\ \mathbf{c}}$	**$**	reduce by $S \rightarrow \mathbf{b}\ \mathbf{c}$	
(7)	0 2 2 2 5	**a a** $\underline{\mathbf{a}\ S}$	**$**	reduce by $S \rightarrow \mathbf{a}\ S$	
(8)	0 2 2 5	**a** $\underline{\mathbf{a}\ S}$	**$**	reduce by $S \rightarrow \mathbf{a}\ S$	
(9)	0 2 5	$\underline{\mathbf{a}\ S}$	**$**	reduce by $S \rightarrow \mathbf{a}\ S$	
(10)	1	$\underline{S}$	**$**	accept	

Intuitive LR Parser Construction

- $G_5 = \{S \rightarrow aS|a\}$

SLR(1) Parser:
S' → S
S → aS | a
L(G) = {a+}



State	a	$	S
0	s2		1
1		Acc	
2	s2/r2	r2	3
3	r1	r1	

S → a$
S → aS$ → aa$
S → aS$ → aaS$ → aaa$

LR(0) Parser Construction

Sample Grammar, G_6

1:	S	$\rightarrow$	x
2:	S	$\rightarrow$	(L)
3:	L	$\rightarrow$	S
4:	L	$\rightarrow$	L, S

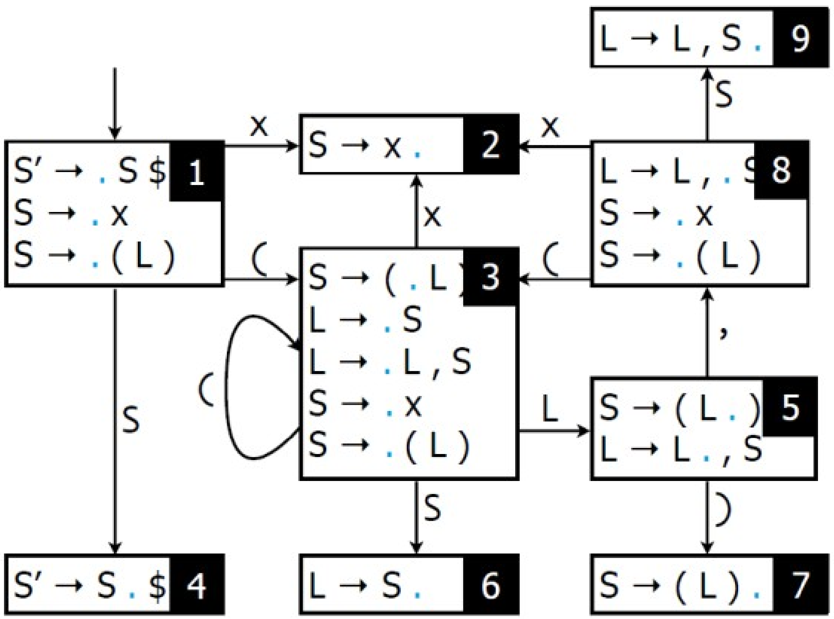
Augmented Grammar, G_6

0:	S'	$\rightarrow$	S
1:	S	$\rightarrow$	x
2:	S	$\rightarrow$	(L)
3:	L	$\rightarrow$	S
4:	L	$\rightarrow$	L, S

- **LR(0) Item**: An LR (0) item is a production in G with dot at some position on the right side of the production. *Examples*: $S \rightarrow .(L)$, $S \rightarrow (.L)$, $S \rightarrow (L.)$, $S \rightarrow (L).$
- **Closure**: Add all items arising from the productions from the non-terminal after the period in an item. Closure is computed transitively. *Examples*:
  - Closure($S \rightarrow .(L)$) = $\{S \rightarrow .(L)\}$
  - Closure($S \rightarrow (.L)$) = $\{S \rightarrow (.L), L \rightarrow .S, L \rightarrow .L, S, S \rightarrow .x, S \rightarrow .(L)\}$
- **State**: Collection of LR(0) items and their closures. *Examples*:
  - $\{S' \rightarrow .S, S \rightarrow .x, S \rightarrow .(L)\}$
  - $\{S \rightarrow (.L), L \rightarrow .S, L \rightarrow .L, S, S \rightarrow .x, S \rightarrow .(L)\}$
- **Actions**: Shift (s#), Reduce (r#), Accept (acc), Reject (' '), GOTO (#):
  - Shift on input symbol to state# (dot precedes the terminal to shift)
  - Reduction on all input symbols by production# (dot at the end of a production)
  - Accept on reduction by the augmented production $S' \rightarrow S$
  - Reject for blank entries – cannot be reached for a valid string
  - GOTO on transition of non-terminal after reduction (dot precedes the non-terminal to reduce to)

LR(0) Parser Example

$$
\begin{array}{llll}
0: & S' & \rightarrow & S \\
1: & S & \rightarrow & x \\
2: & S & \rightarrow & (L) \\
3: & L & \rightarrow & S \\
4: & L & \rightarrow & L, S
\end{array}
$$

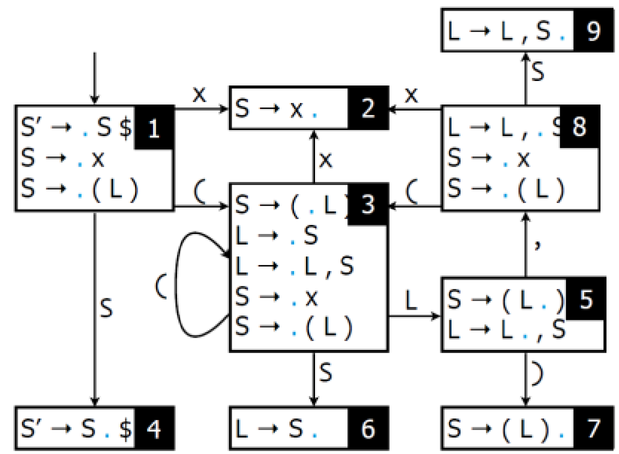


	(	)	x	,	$	S	L
1	s 3		s 2			g 4	
2	r 1	r 1	r 1	r 1	r 1		
3	s 3		s 2			g 6	g 5
4					a		
5		s 7		s 8			
6	r 3	r 3	r 3	r 3	r 3		
7	r 2	r 2	r 2	r 2	r 2		
8	s 3		s 2			g 9	
9	r 4	r 4	r 4	r 4	r 4		

Source: https://www.slideshare.net/eelcovisser/lr-parsing-71059803?from_action=save

LR(0) Parser Example: Parsing (x , x) $

	(	)	x	,	$	S	L
1	s 3		s 2			g 4	
2	r 1	r 1	r 1	r 1	r 1		
3	s 3		s 2			g 6	g 5
4					a		
5		s 7		s 8			
6	r 3	r 3	r 3	r 3	r 3		
7	r 2	r 2	r 2	r 2	r 2		
8	s 3		s 2			g 9	
9	r 4	r 4	r 4	r 4	r 4		



Step	Stack	Symbols	Input	Action
(1)	1		(x , x) $	shift
(2)	1 3	(	x , x) $	shift
(3)	1 3 2	(x	, x) $	reduce by $S \rightarrow$ **x**
(4)	1 3 6	(S	, x) $	reduce by $L \rightarrow S$
(5)	1 3 5	(L	, x) $	shift
(6)	1 3 5 8	(L ,	x) $	shift
(7)	1 3 5 8 2	(L , x	) $	reduce by $S \rightarrow$ **x**
(8)	1 3 5 8 9	(L , S	) $	reduce by $L \rightarrow L$ **,** S
(9)	1 3 5	(L	) $	shift
(10)	1 3 5 7	(L)	$	reduce by $S \rightarrow$ **(** L **)**
(11)	1 4	S	$	accept

Source: https://www.slideshare.net/eelcovisser/lr-parsing-71059803?from_action=save

LR(0) Parser: Practice Example

Construct an LR(0) parser for G_7:

$$
\begin{array}{llll}
1: & S & \rightarrow & A\ A \\
2: & A & \rightarrow & a\ A \\
3: & A & \rightarrow & b
\end{array}
$$

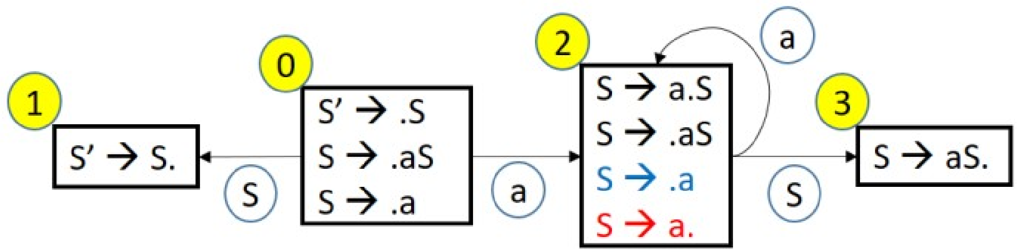
LR Parsers

SLR(1) Parser

LR(0) Parser: Shift-Reduce Conflict

- $G_5 = \{S \rightarrow aS|a\}$

SLR(1) Parser:
S' → S
S → aS | a
L(G) = {a+}

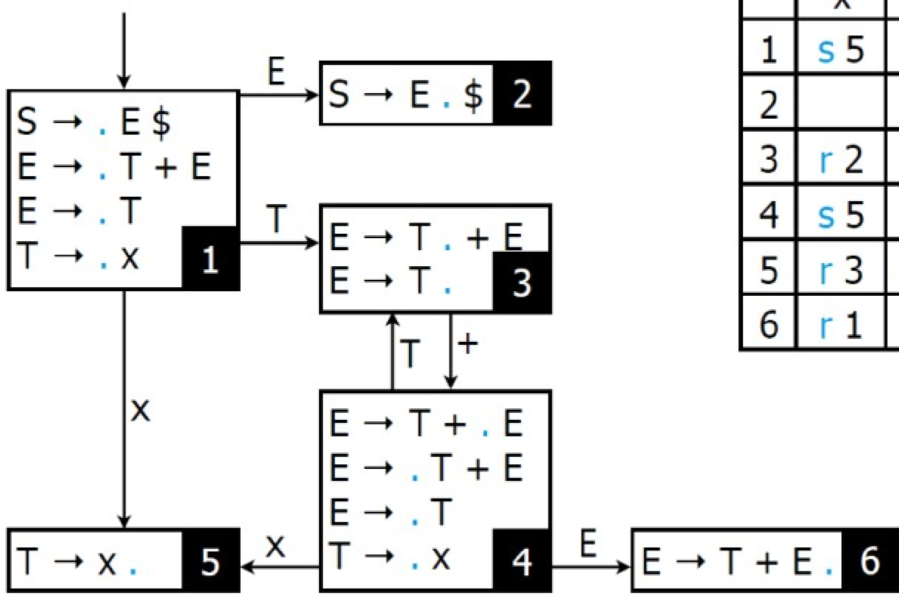


State	a	$	S
0	s2		1
1		Acc	
2	s2/r2	r2	3
3	r1	r1	

S → a$
S → aS$ → aa$
S → aS$ → aaS$ → aaa$

- Consider State 2.
  - By $S \rightarrow .a$, we should shift on a and remain in state 2
  - By $S \rightarrow a.$, we should reduce by production 2
- We have a Shift-Reduce Conflict
- As $FOLLOW(S) = \{\$\}$, we decide in favor of shift. Why?

LR(0) Parser: Shift-Reduce Conflict



	x	+	$	E	T
1	s 5			g 2	g 3
2			a		
3	r 2	?	r 2		
4	s 5			g 6	g 3
5	r 3	r 3	r 3		
6	r 1	r 1	r 1		

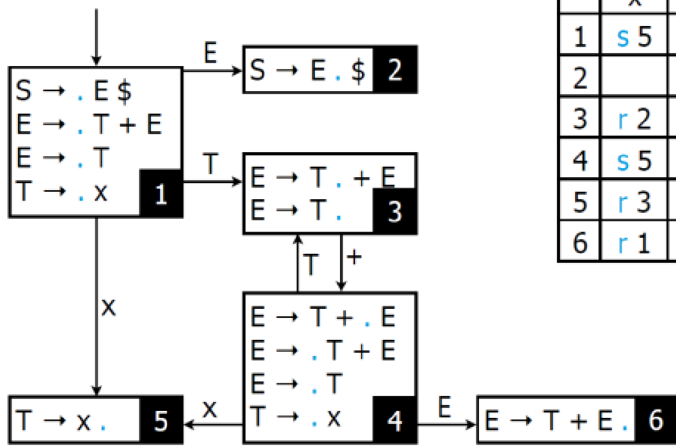
E → T + E
E → T
T → x

- Consider State 3.
  - By $E \to T. + E$, we should shift on $+$ and move to state 4
  - By $E \to T.$, we should reduce by production 2
- We have a Shift-Reduce Conflict
- To resolve, we build SLR(1) Parser

SLR(1) Parser Construction

- **LR(0) Item**: Canonical collection of LR(0) Items used in SLR(1) as well
- **Closure**: Same way as LR(0)
- **State**: Collection of LR(0) items and their closures.
- **Actions**: Shift (s#), Reduce (r#), Accept (acc), Reject (<space>), GOTO (#):
  - Shift on input symbol to state#
  - **Reduction by production# only on the input symbols that belong to the FOLLOW of the left-hand side**
  - Accept on reduction by the augmented production
  - GOTO on transition of non-terminal after reduction

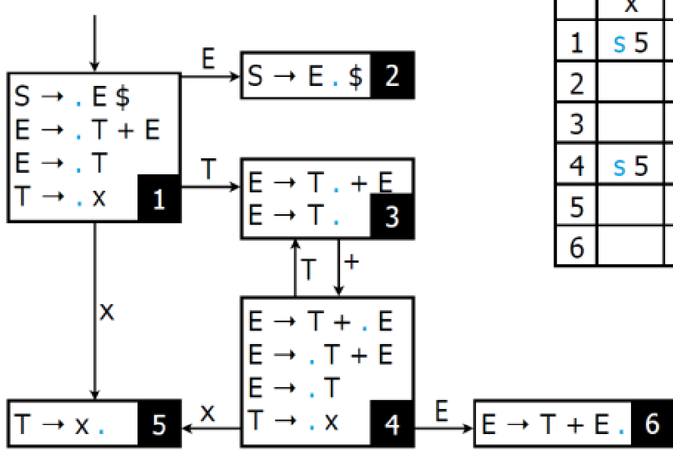
SLR Parse Table: Shift-Reduce Conflict on LR(0)



	x	+	$	E	T
1	s 5			g 2	g 3
2			a		
3	r 2	?	r 2		
4	s 5			g 6	g 3
5	r 3	r 3	r 3		
6	r 1	r 1	r 1		

E → T + E
E → T
T → x

Reduce a production $S \rightarrow \ldots$ on symbols $k \in T$ if $k \in \mathit{Follow}(S)$



	x	+	$	E	T
1	s 5			g 2	g 3
2			a		
3		s 4	r 2		
4	s 5			g 6	g 3
5		r 3	r 3		
6			r 1		

E → T + E
E → T
T → x

SLR(1) Parser: Practice Example

Construct an SLR(1) parser for G_8:

1:	S	$\rightarrow$	E
2:	E	$\rightarrow$	$E + T$
3:	E	$\rightarrow$	T
4:	T	$\rightarrow$	$T * F$
5:	T	$\rightarrow$	F
6:	F	$\rightarrow$	$\textbf{id}$

LR Parsers

LR(1) Parser

SLR(1) Parser: Shift-Reduce Conflict

Grammar G_9

1:	S	$\rightarrow$	$L = R$
2:	S	$\rightarrow$	R
3:	L	$\rightarrow$	$*R$
4:	L	$\rightarrow$	id
5:	R	$\rightarrow$	L

I_0: $S' \rightarrow \cdot S$
$S \rightarrow \cdot L = R$
$S \rightarrow \cdot R$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$
$R \rightarrow \cdot L$

I_1: $S' \rightarrow S \cdot$

I_2: $S \rightarrow L \cdot = R$
$R \rightarrow L \cdot$

I_3: $S \rightarrow R \cdot$

I_4: $L \rightarrow * \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_5: $L \rightarrow \mathbf{id} \cdot$

I_6: $S \rightarrow L = \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_7: $L \rightarrow * R \cdot$

I_8: $R \rightarrow L \cdot$

I_9: $S \rightarrow L = R \cdot$

- $= \in FOLLOW(R)$ as $S \Rightarrow L = R \Rightarrow *R = R$
- So in State#2 we have a shift/reduce Conflict on $=$
- The grammar is not ambiguous. Yet we have the shift/reduce conflict as SLR is not powerful enough to remember enough left context to decide what action the parser should take on input $=$, having seen a string reducible to L.
- To resolve, we build LR(1) Parser

LR(1) Parser Construction

Sample Grammar G_7				Augmented Grammar G_7			
				0:	S'	$\rightarrow$	S
1:	S	$\rightarrow$	CC	1:	S	$\rightarrow$	CC
2:	C	$\rightarrow$	cC	2:	C	$\rightarrow$	cC
3:	C	$\rightarrow$	d	3:	C	$\rightarrow$	d

- **LR(1) Item**: An LR(1) item has the form $[A \rightarrow \alpha.\beta, a]$ where $A \rightarrow \alpha\beta$ is a production and a is the look-ahead symbol which is a terminal or \$. As the dot moves through the right-hand side of the production, token a remains attached to it. LR(1) item $[A \rightarrow \alpha., a]$ calls for a reduce action when the look-ahead is a. *Examples*: $[S \rightarrow .CC, \$]$, $[S \rightarrow C.C, \$]$, $[S \rightarrow CC., \$]$
- **Closure(S)**:

  For each item $[A \rightarrow \alpha.B\beta, t] \in S$,
  For each production $B \rightarrow \gamma \in G$,
  For each token $b \in FIRST(\beta t)$,
  Add $[B \rightarrow .\gamma, b]$ *to* S

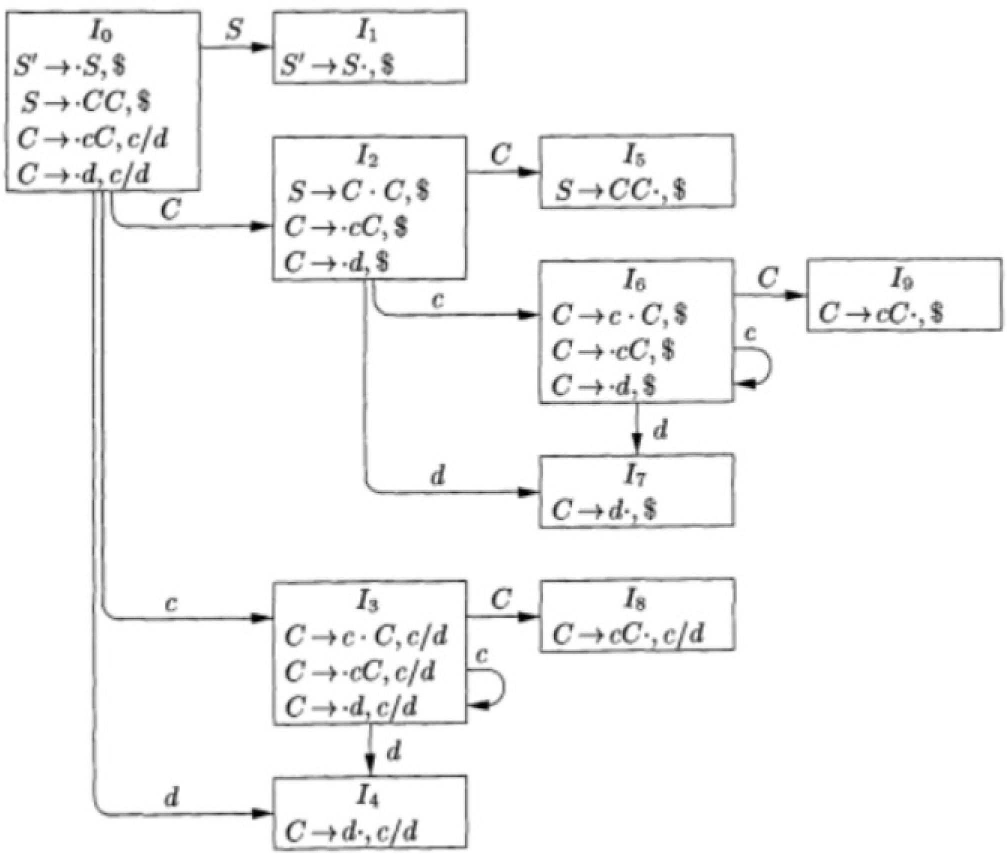
  Closure is computed transitively. *Examples*:

  - Closure$([S \rightarrow C.C, \$]) = \{[S \rightarrow C.C, \$], [C \rightarrow .cC, \$], [C \rightarrow .d, \$]\}$
  - Closure$([C \rightarrow c.C, c/d]) = \{[C \rightarrow c.C, c/d], [C \rightarrow .cC, c/d], [C \rightarrow .d, c/d]\}$
- **State**: Collection of LR(1) items and their closures. *Examples*:
  - $\{[S \rightarrow C.C, \$], [C \rightarrow .cC, \$], [C \rightarrow .d, \$]\}$
  - $\{[C \rightarrow c.C, c/d], [C \rightarrow .cC, c/d], [C \rightarrow .d, c/d]\}$

LR(1) Parser: Example

Construct an LR(1) parser for G_7:

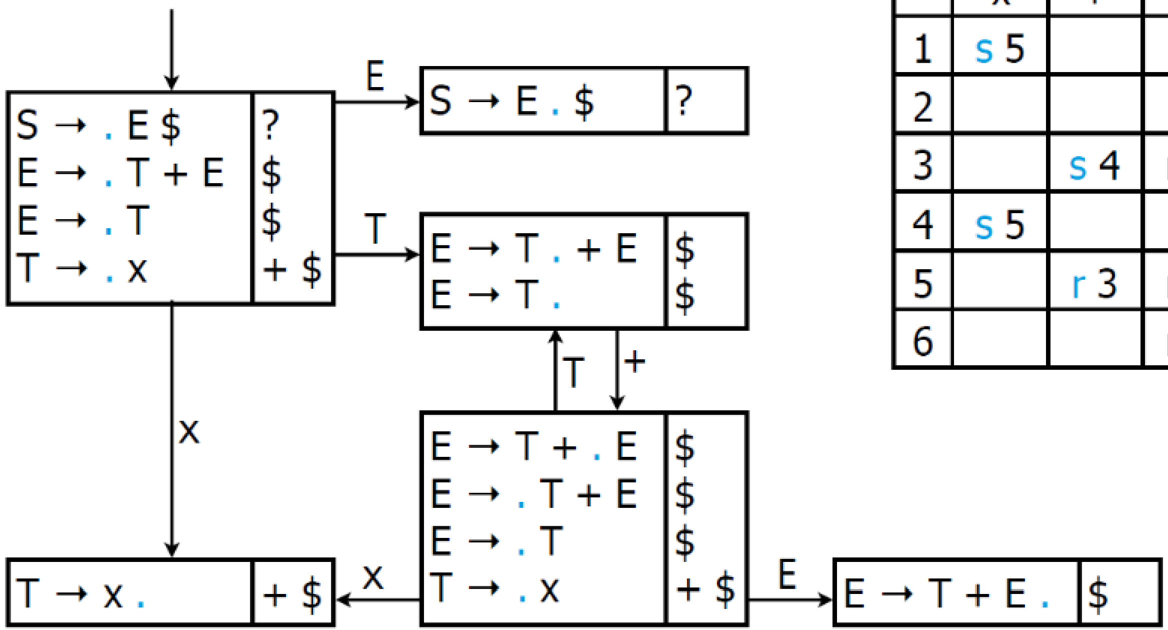
$$\begin{array}{llll} 1: & S & \rightarrow & CC \\ 2: & C & \rightarrow & cC \\ 3: & C & \rightarrow & d \end{array}$$



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		

Source: Dragon Book

LR(1) Parser: Example



	x	+	$	E	T
1	s 5			g 2	g 3
2			a		
3		s 4	r 2		
4	s 5			g 6	g 3
5		r 3	r 3		
6			r 1		

E → T + E
E → T
T → x

Source: https://www.slideshare.net/eelcovisser/lr-parsing-71059803?from_action=save

LR Parsers

LALR(1) Parser

LALR(1) Parser Construction

Sample Grammar G_7				**Augmented Grammar** G_7			
				0:	S'	$\rightarrow$	S
1:	S	$\rightarrow$	CC	1:	S	$\rightarrow$	CC
2:	C	$\rightarrow$	cC	2:	C	$\rightarrow$	cC
3:	C	$\rightarrow$	d	3:	C	$\rightarrow$	d

- **LR(1) States**: Construct the Canonical LR(1) parse table.
- **LALR(1) States**: Two or more LR(1) states having the same set of core LR(0) items may be merged into one by combining the look-ahead symbols for every item. Transitions to and from these merged states may also be merged accordingly. All other states and transitions are retained. *Examples*:
  - Merge
    State#3 $= \{[C \rightarrow c.C, c/d], [C \rightarrow .cC, c/d], [C \rightarrow .d, c/d]\}$ with
    State#6 $= \{[C \rightarrow c.C, \$], [C \rightarrow .cC, \$], [C \rightarrow .d, \$]\}$ to get
    State#36 $= \{[C \rightarrow c.C, c/d/\$], [C \rightarrow .cC, c/d/\$], [C \rightarrow .d, c/d/\$]\}$
  - Merge

    State#4 $= \{[C \rightarrow d., c/d]\}$ with

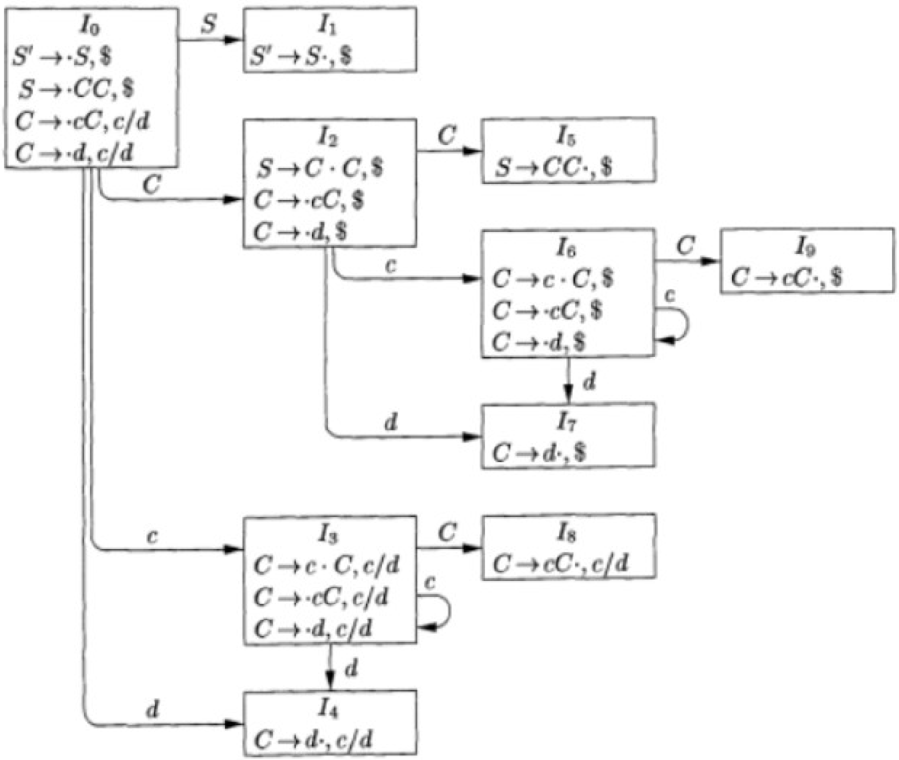
    State#7 $= \{[C \rightarrow d., \$]\}$ to get

    State#47 $= \{[C \rightarrow d., c/d/\$]\}$
- **Reduce/Reduce Conflict**: LR(1) to LALR(1) transformation cannot introduce any new shift/reduce conflict. But it may introduce reduce/reduce conflict.

LALR(1) Parser: Example

Construct an LALR(1) parser for G_7:

$$\begin{array}{llll} 1: & S & \rightarrow & CC \\ 2: & C & \rightarrow & cC \\ 3: & C & \rightarrow & d \end{array}$$



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		

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		

Source: Dragon Book

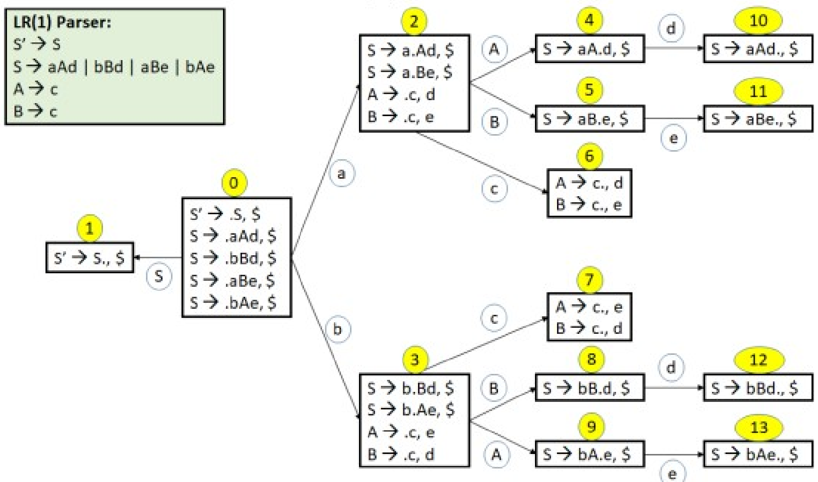
LALR(1) Parser: Reduce-Reduce Conflict

Consider $G_{10} = < \{a, b, c, d, e\}, \{S, A, B\}, S, P >$ where $P =$

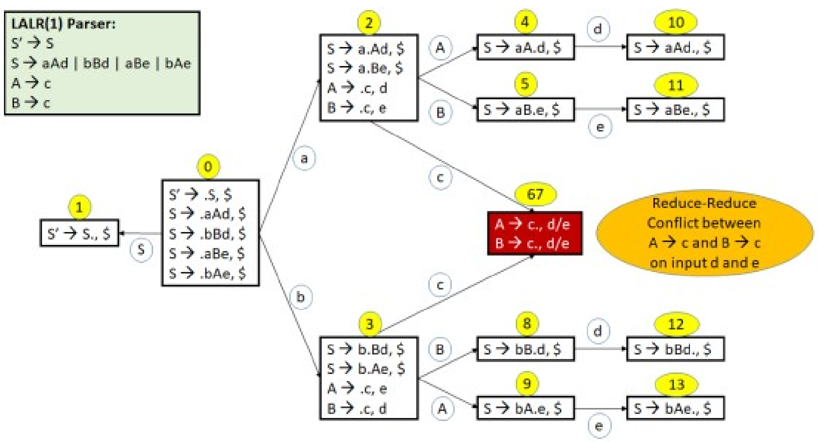
$$
\begin{array}{llll}
0: & S' & \rightarrow & S \\
1: & S & \rightarrow & aAd \\
2: & S & \rightarrow & bBd \\
3: & S & \rightarrow & aBe \\
4: & S & \rightarrow & bAe \\
5: & A & \rightarrow & c \\
6: & B & \rightarrow & c
\end{array}
$$

Clearly, $L(G) = \{acd, bcd, ace, bce\}$

LR(1) Parser



LALR(1) Parser



LR Parsers: Practice Examples

Determine the LR Class (LR(0), SLR(1), LR(1) or LALR(1)) for the following grammars:

- G: $S \to aSb \mid b$
- G: $S \to Sa \mid b$
- G: $S \to (S) \mid SS \mid \epsilon$
- G: $S \to (S) \mid SS \mid ()$
- G: $S \to ddX \mid aX \mid \epsilon$
- G: $S \to E$; $E \to T + E \mid T$; $T \to int * T \mid int \mid (E)$
- G: $S \to V = E \mid E$; $E \to V$; $V \to x \mid *E$
- G: $S \to AB$; $A \to aAb \mid a$; $B \to d$

LR Parsers

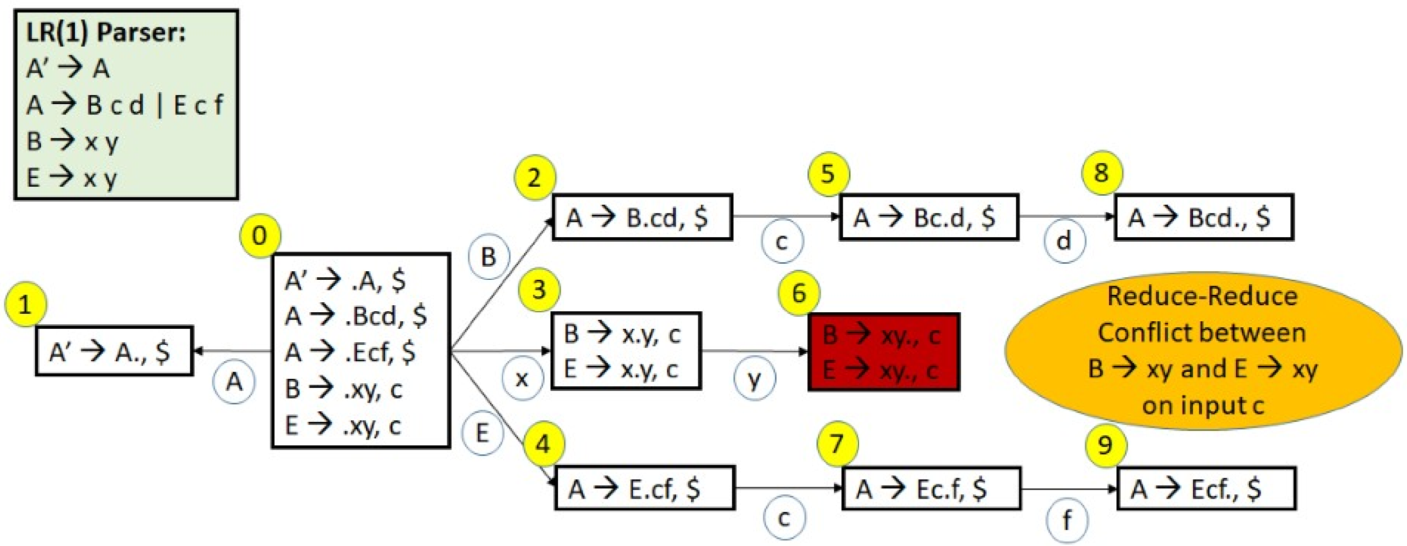
LR(k) Parser

LR(1) Parser: Shift-Reduce Conflict

Grammar G_{11}

1:	A	$\rightarrow$	$B\ c\ d$
2:	A	$\rightarrow$	$E\ c\ f$
3:	B	$\rightarrow$	$x\ y$
4:	E	$\rightarrow$	$x\ y$

- For this grammar, an example input that starts with *xyc* is enough to confuse an LR(1) parser, as it has to decide whether *xy* matches B or E after only seeing 1 symbol further (i.e. c).



- An LL(1) parser would also be confused, but at the x - should it expand A to $B\ c\ d$ or to $E\ c\ f$, as both can start with x. An LL(2) or LL(3) parser would have similar problems at the y or c respectively.
- An LR(2) parser would be able to also see the d or f that followed the c and so make the correct choice between B and E.
- An LL(4) parser would also be able to look far enough ahead to see the d or f that followed the c and so make the correct choice between expanding A to $B\ c\ d$ or to $E\ c\ f$.

LR(k) Parser: Shift-Reduce Conflict

Grammar G_{12}

1:	A	$\rightarrow$	$B\ C\ d$
2:	A	$\rightarrow$	$E\ C\ f$
3:	B	$\rightarrow$	$x\ y$
4:	E	$\rightarrow$	$x\ y$
5:	C	$\rightarrow$	$C\ c$
6:	C	$\rightarrow$	c

- The grammar would confuse any LR(k) or LL(k) parser with a fixed amount of look-ahead
- To workaround, rewrite

1:	A	$\rightarrow$	$B\ C\ d$
2:	A	$\rightarrow$	$E\ C\ f$
3:	B	$\rightarrow$	$x\ y$
4:	E	$\rightarrow$	$x\ y$

as

1:	A	$\rightarrow$	$BorE\ c\ d$
2:	A	$\rightarrow$	$BorE\ c\ f$
3:	$BorE$	$\rightarrow$	$x\ y$

LR(1) Parser:
A' → A
A → BorE c d | BorE c f
BorE → x y