Bottom-up parsing

Bottom-up parsing – an overview

- The most general bottom-up parsing algorithm that we will be considering in this lecture, is the *LR*(1) parsing algorithm.
  - The *L* indicates that the input is processed from the *left* to the right.
  - The *R* indicates that a *rightmost derivation* is obtained.
  - The 1 indicates that a single token is used for lookahead.
- *LR*(0) parsers examine the "lookahead" token only after it appears on the parsing stack.
- *SLR*(1) (simple *LR*(1)) parsers improve on *LR*(0) parsers.
- A more powerful method, but not as general as *LR*(1) parsing, is *LALR*(1) (lookahead *LR*(1)) parsing.
- Bottom-up parsers are generally more powerful than their top-down counterparts – for example left recursion can be handled.
- Bottom-up parsers are unsuitable for hand coding, so parser generators such as *bison* are used.

Bottom-up parsing – overview

- The parsing stack contains tokens and nonterminals *PLUS* state information.
- The parsing stack starts empty and ends with the *start symbol* alone on the stack.
- Actions: *shift*, *reduce* and *accept*.
- A *shift* merely moves a token from the input to the top of the stack.
- A *reduce* replaces the string α on top of the stack with a nonterminal A, given we have the rule $A \to \alpha$.
- If the grammar does not possess a unique start symbol that only appears once in the grammar, then the grammar is augmented to contain such a start symbol.

Bottom-up parse for ()

- Consider the grammar $S \rightarrow (\; S \;) \; S \mid \epsilon$.
- Augment it by adding: $S' \rightarrow S$.
- A bottom-up parse for () follows:

	Parsing stack	*Input*	*Action*
1	$	()$	*shift*
2	$ (	)$	*reduce* $S \rightarrow \epsilon$
3	$ (S	)$	*shift*
4	$ (S)	$	*reduce* $S \rightarrow \epsilon$
5	$ (S) S	$	*reduce* $S \rightarrow (\; S \;) \; S$
6	$ S	$	*reduce* $S' \rightarrow S$
7	$ S'	$	*accept*

- The corresponding derivation is: $S' \Rightarrow S \Rightarrow (S)S \Rightarrow (S) \Rightarrow ()$

A bottom-up parse for $n + n$

- Consider the grammar $E \to E + n \mid n$.
- Augment it by adding: $E' \to E$.
- A bottom-up parse for $n + n$:

	Parsing stack	Input	Action
1	$\$$	$n + n\$$	shift
2	$\$\, n$	$+ n\$$	reduce $E \to n$
3	$\$\, E$	$+ n\$$	shift
4	$\$\, E +$	$n\$$	shift
5	$\$\, E + n$	$\$$	reduce $E \to E + n$
6	$\$\, E$	$\$$	reduce $E' \to E$
7	$\$\, E'$	$\$$	accept

- The corresponding derivation is: $E' \Rightarrow E \Rightarrow E + n \Rightarrow n + n$

Bottom-up parse – overview

	Parsing stack	Input	Action
1	\$	$n + n\$$	shift
2	$\$\ n$	$+ n\$$	reduce $E \rightarrow n$
3	$\$\ E$	$+ n\$$	shift
4	$\$\ E\ +$	$n\$$	shift
5	$\$\ E + n$	\$	reduce $E \rightarrow E + n$
6	$\$\ E'$	\$	reduce $E' \rightarrow E$
7	$\$\ E$	\$	accept

- In the derivation: $E' \Rightarrow E \Rightarrow E + n \Rightarrow n + n$, each of the intermediate strings is called a *sentential form*, and the sentential form is split between the parse stack and the input.
- $E + n$ occurs in step 3 of the parse as $E \,|\, + n$, and as $E + | n$ in step 4, and finally as $E + n|$.
- The string of symbols on top of the stack is called a *viable prefix* of a sentential form. E, $E+$ and $E + n$ are all viable prefixes of $E + n$ in step 5.
- The viable prefixes of $n + n$ are ϵ and n, but $n+$ and $n + n$ are not.

Bottom-up parse – overview

- A shift-reduce parser will shift terminals to the stack until it can perform a reduction to obtain the next sentential form.
- This occurs when the top of the stack matches the right-hand side of a production.
- This string, on the top of the stack, together with the position in the sentential form where it occurs, and the production used to reduce it, is known as a *handle* for the sentential form.
- In step 2 a handle of $n + n$ is thus the leftmost n together with the production $E \to n$. In step 5 a handle of $E + n$ is $E + n$ together with the production $E \to E + n$.
- The main task of a shift-reduce parser is to find the next handle.

Bottom-up parse – overview

	Parsing stack	Input	Action
1	\$	()\$	*shift*
2	\$ (	)\$	*reduce* $S \to \varepsilon$
3	\$ (S	)\$	*shift*
4	\$ (S)	\$	*reduce* $S \to \varepsilon$
5	\$ (S) S	\$	*reduce* $S \to (S) S$
6	\$ S	\$	*reduce* $S' \to S$
7	\$ S	\$	*accept*

- Reductions only occur if the reduced string is part of a sentential form.
- In step 3 above the reduction $S \to \epsilon$ cannot be performed, because the resulting string after the shift of) onto the stack would be $(S\ S)$, which is not a sentential form. Thus ϵ and the production $S \to \epsilon$ is not a handle at this position of the sentential form (S).
- In order to reduce with $S \to (S)S$, the parser has to know that $(S)S$ is on the top of the stack by using a DFA of "items".

$LR(0)$ items

- The grammar $S' \to S,\ \ S \to (S)S \mid \epsilon$ has three productions and eight $LR(0)$ items:

$$\begin{array}{lcl} S' & \to & .S \\ S' & \to & S. \\ S & \to & .(S)S \\ S & \to & (.S)S \\ S & \to & (S.)S \\ S & \to & (S).S \\ S & \to & (S)S. \\ S & \to & . \end{array}$$

- The grammar $E' \to E,\ E \to E + n \mid n$ has three productions and eight $LR(0)$ items:

$$\begin{array}{lcl} E' & \to & .E \\ E' & \to & E. \\ E & \to & .E + n \\ E & \to & E\,.+n \\ E & \to & E + .n \\ E & \to & E + n. \\ E & \to & .n \\ E & \to & n. \end{array}$$

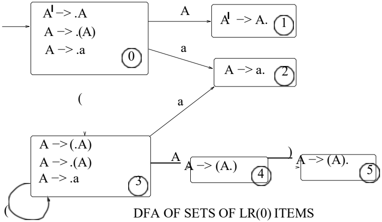
$LR(0)$ parsing – $LR(0)$ items

- An $LR(0)$ *item* of a CFG is a production with a distinguished position in its right-hand side.
- The distinguished position is usually denoted with the meta symbol ".".
- e.g. if $A \rightarrow \alpha$ and β and γ are any two strings such that $\alpha = \beta\gamma$, then $A \rightarrow .\beta\gamma$, $A \rightarrow \beta.\gamma$ and $A \rightarrow \beta\gamma.$ are all $LR(0)$ items.
- They are called $LR(0)$ items because they contain no explicit reference to lookahead.
- The item "records" the recognition of the right-hand side of a particular production.
- $A \rightarrow \beta.\gamma$ denotes that the β part is on top of the parsing stack.

$LR(0)$ parsing – $LR(0)$ items

- The item $A \rightarrow .\alpha$ (called an *initial item*) indicates that α could potentially be reduced to A if we can get α on the top of the stack.
- The item $A \rightarrow \alpha.$ (called a *complete item*) indicates that α is on the top of the stack and α is a handle if $A \rightarrow \alpha$ is used to reduce α to A.
- The $LR(0)$ items are used as states of a finite automaton that maintains information about the parse stack and the progress of a shift-reduce parse.

An LR(0) parsing example

- Consider the grammar $A \rightarrow (\ A\ ) \mid a$.
  We augment this grammar with the rule $A' \rightarrow A$, where A' is the new start symbol.
- The procedure to construct the following DFA of LR(0) items will be explained later.
  At this stage we show how to use this DFA of LR(0) items in order to obtain a parsing table and we also describe the parsing actions for the string $((a))$.
- 

DFA OF SETS OF LR(0) ITEMS

LR(0) parsing example continue

PARSING ACTIONS

	Parsing stack	Input	Action
1	$ 0	((a)) $	shift
2	$ 0 (3	(a)) $	shift
3	$ 0 (3 (3	a)) $	shift
4	$ 0 (3 (3 a 2	)) $	reduce A –> a
5	$ 0 (3 (3 A 4	)) $	shift
6	$ 0 (3 (3 A 4) 5	) $	reduce A –> (A)
7	$ 0 (3 A 4	) $	shift
8	$ 0 (3 A 4) 5	$	reduce A –> (A)
9	$ 0 A 1	$	accept

PARSING TABLE

State	Action	Rule	Input			Goto
			(	a	)	A
0	shift		3	2		1
1	reduce	A' –> A				
2	reduce	A –> a				
3	shift		3	2		4
4	shift				5	
5	reduce	A –> (A)				

$LR(0)$ parsing – automata of items

- An automaton of $LR(0)$ items keeps track of the progress of a parse.
- One approach is to first construct a NFA of $LR(0)$ items and then derive a *DFA* from it.
  Another approach is to construct the *DFA* of sets of $LR(0)$ items directly.
- Which transitions are present in the *NFA* of $LR(0)$ items?
  Suppose that the symbol X is a terminal or nonterminal.
  Let $A \to \alpha.X\eta$ be an $LR(0)$ item in one of the states of the NFA of $LR(0)$ items, which indicates that α is at the top of the parsing stack.
  Then we have the following transition:

$$\boxed{A \to \alpha.X\eta} \xrightarrow{X} \boxed{A \to \alpha X.\eta}$$

LR(0) parsing – automata of items

- The transition

$$\boxed{A \to \alpha.X\eta} \xrightarrow{X} \boxed{A \to \alpha X.\eta}$$

  where X is a nonterminal, corresponds to pushing X onto the stack after *reducing* some β to X by applying the rule $X \to \beta$
- Before using such a reduction, β must be at the top of the parsing stack, i.e. we must be in a state containing the item $X \to .\beta$
- For each production $X \to \beta$, ϵ-transitions are constructed from states containing $A \to \alpha.X\eta$ to a state containing $X \to .\beta$

$$\boxed{A \to \alpha.X\eta} \xrightarrow{\epsilon} \boxed{X \to .\beta}$$

$LR(0)$ parsing –automata of items

- We have the following two types of transitions in the NFA of $LR(0)$ items:

  $$\boxed{A \to \alpha.X\eta} \xrightarrow{X} \boxed{A \to \alpha X.\eta}$$

  where X is a terminal or nonterminal and

  $$\boxed{A \to \alpha.X\eta} \xrightarrow{\epsilon} \boxed{X \to .\beta}$$

  if we have a production $X \to \beta$
- The start state is a state containing $S' \to .S$, where S' is a new start variable. (Recall that we augment the grammar with the rule $S' \to S$.)

$LR(0)$ parsing – automata of items

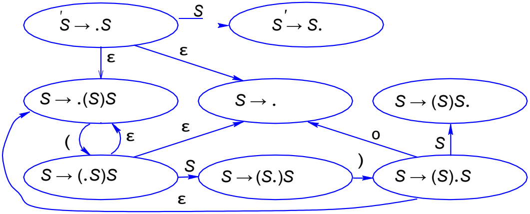
- Which states are accepting states in the *NFA*?
  The *NFA* does not need accepting states.
- The *NFA* is not being used to do the recognition of the language.
- The *NFA* is merely being applied to keep track of the state of the parse.
- The parser itself determines when it accepts an input stream by determining that the input stream is empty and the start symbol is on the top of the parse stack.

$LR(0)$ parsing – automata of items

- The grammar $S' \rightarrow S,\ \ S \rightarrow (S)S \mid \epsilon$ has three productions and eight $LR(0)$ items:

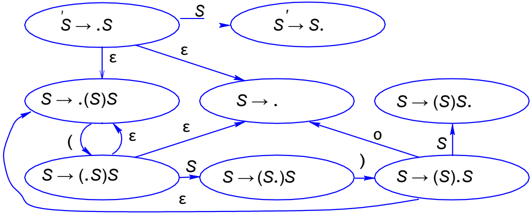
$$
\begin{array}{lcl}
S' & \rightarrow & .S \\
S' & \rightarrow & S. \\
S & \rightarrow & .(S)S \\
S & \rightarrow & (.S)S \\
S & \rightarrow & (S.)S \\
S & \rightarrow & (S).S \\
S & \rightarrow & (S)S. \\
S & \rightarrow & .
\end{array}
$$

- The *NFA* of $LR(0)$ items:

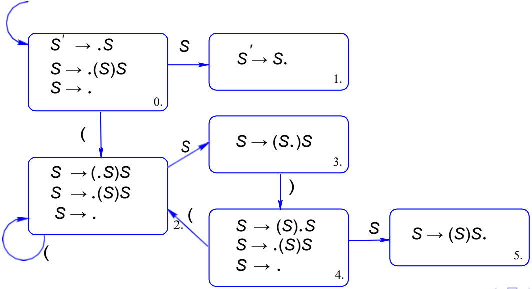


LR(0) parsing - *NFA* and corresponding *DFA* of *LR*(0) items

- The *NFA* of *LR*(0) items:



- The *DFA* of *LR*(0) items derived from the *NFA*:

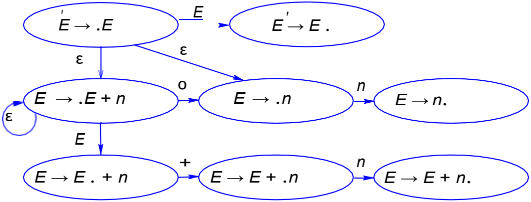


$LR(0)$ parsing – finite automata of items

- Consider the grammar $E' \to E$, $E \to E + n \mid n$ with three productions and eight $LR(0)$ items:

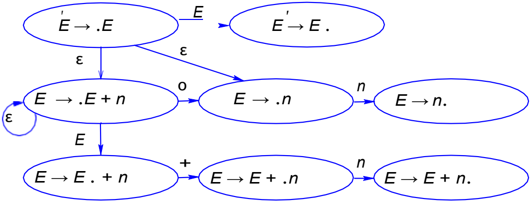
$$
\begin{array}{lcl}
E' & \to & .E \\
E' & \to & E. \\
E & \to & .E + n \\
E & \to & E. + n \\
E & \to & E + .n \\
E & \to & E + n. \\
E & \to & .n \\
E & \to & n.
\end{array}
$$

- The *NFA* of $LR(0)$ items:

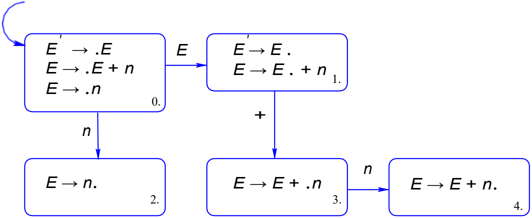


LR(0) parsing: *NFA* and equivalent *DFA*

- The *NFA* for the grammar:



- The *DFA* derived from the above *NFA*:



- The items that are added by ε-closure are known as *closure items* and those items that originate states are *kernel items*.

LR(0) parsing

	Parsing stack	Input	Action
1	\$ 0	n + n \$	shift
2	\$ 0 n 2	+ n \$	reduce E –> n
3	\$ 0 E 1	+ n \$	shift
4	\$ 0 E 1 + 3	n \$	shift
5	\$ 0 E 1 + 3 n 4	\$	reduce E –> E + n
6	\$ 0 E 1	\$	accept

Parsing actions for n+n

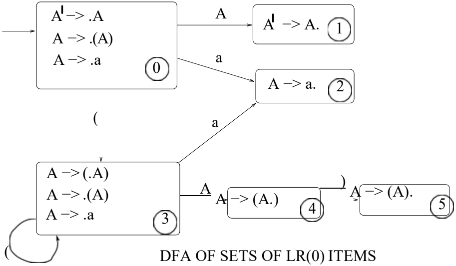
- The problem with parsing this grammar is that in both steps 2 and 6 we first have to look at the next input symbol (which is not allowed in LR(0) parsing), in order to decide if we should shift or reduce.
- We say that we have a shift-reduce conflict in state 1 of the DFA of sets of LR(0) items.

LR(0) parsing *shift-reduce* and *reduce-reduce* conflicts

- A grammar is said to be an *LR*(0) grammar if the parser rules are unambiguous.
- If a state contains the complete item $A \to \alpha.$, then it cannot contain other items, otherwise the grammar is not *LR*(0).
- If a state contains a complete item $A \to \alpha.$, and a *shift* item $A \to \alpha.X\beta$, where X is a terminal, then an ambiguity arises as to whether one should shift or reduce. This is called a *shift-reduce conflict*.
- If a state contains $A \to \alpha.$ and another complete item $B \to \beta.$, then an ambiguity arises as to which production ($A \to \alpha.$ or $B \to \beta.$) to apply during reduction. This is known as a *reduce-reduce* conflict.
- A grammar is therefore *LR*(0) if and only if each state is either a *shift* state or a *reduce* state containing a single complete item.

$LR(0)$ parsing

- Consider the grammar $A \rightarrow (A) \mid a$ with DFA of $LR(0)$ items given by:

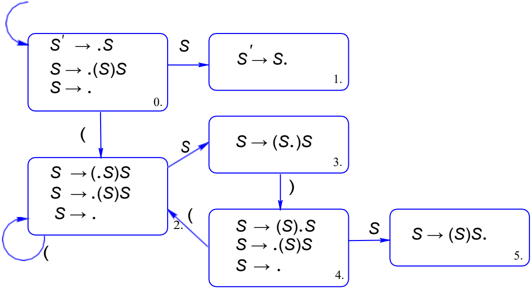


DFA OF SETS OF LR(0) ITEMS

- States 0,3,4 are shift states.
- States 1,2,5 are reduce states.
- This grammar is $LR(0)$.

$LR(0)$ parsing – automata of items

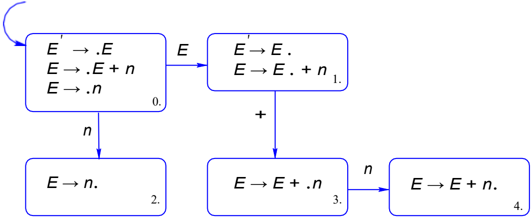
- Consider the grammar $S' \to S$, $S \to (S)S \mid \epsilon$ with *DFA* of $LR(0)$ items given by:



- This grammar is not LR(0) since states 0,2,4 have shift-reduce conflicts.

$LR(0)$ parsing – finite automata of items

- Consider the grammar $E' \to E$, $E \to E + n \mid n$ with *DFA* of $LR(0)$ items given by:



- This grammar is not LR(0), since state 1 has a shift-reduce conflict.

SLR(1) parsing

- Next we discuss *SLR*(1) parsing.

The *SLR*(1) parsing algorithm

- Simple *LR*(1), i.e. *SLR*(1) parsing, uses a *DFA* of sets of *LR*(0) items.
- The power of *LR*(0) is *significantly* increased by using the next token in the input stream to direct the actions of the parser in two ways:
  1. The input token is consulted *before* a shift is made, to ensure that an appropriate *DFA* transition exists.
  2. The parser uses the *follow set* of a terminal to decide if a reduction should be performed.
- This parsing approach is powerful enough to parse almost all common programming language constructs.

The $SLR(1)$ parsing algorithm

Let s be the current state, i.e. the state on top of the stack.

1. If s contains any item of the form $A \to \alpha.X\beta$, where X is the next terminal in the input stream, then *shift* X onto the stack and push the state containing the item $A \to \alpha X.\beta$
2. If s contains the complete item $A \to \gamma.$, and the next token in the input stream is in $follow(A)$, then *reduce* by the rule $A \to \gamma$
3. If the next input token is not accommodated by (1) or (2), then an *error* is declared.

SLR(1) grammar

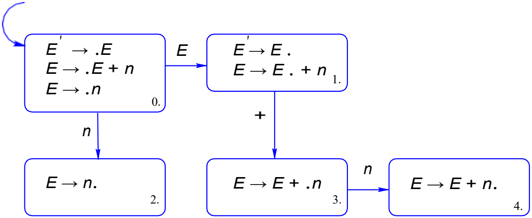
A grammar is an *SLR*(1) *grammar* if the application of the *SLR*(1) parsing rules do not result in an ambiguity.

Thus a grammar is an *SLR*(1) *grammar* $\Leftarrow\Rightarrow$

1. For any item $A \rightarrow \alpha.X\beta$, where X is a terminal there is no complete item $B \rightarrow \gamma.$ in s with $X \in follow(B)$.
   A violation of this condition is a *shift-reduce* conflict.
2. For any two complete items $A \rightarrow \alpha. \in s$ and $A \rightarrow \beta. \in s$, $follow\ (A) \cap follow\ (B) = \emptyset$.
   A violation of this condition is a *reduce-reduce* conflict.

$SLR(1)$ grammar

- The grammar with $E' \rightarrow E$, $E \rightarrow E + n | n$
  is not $LR(0)$ but is $SLR(1)$.
  Its *DFA* of sets of LR(0) items is:



- $follow(E') = \{\$\}$, and $follow(E) = \{\$, +\}$
- SLR(1) Parsing Table:

State	*Input*			*Goto*
	n	$+$	$\$$	E
0	$s2$			1
1		$s3$	*accept*	
2	$r(E \rightarrow n)$		$r(E \rightarrow n)$	
3	$s4$			
4		$r(E \rightarrow E + n)$	$r(E \rightarrow E + n)$	

$SLR(1)$ parse of $n + n + n$

- SLR(1) Parsing Table:

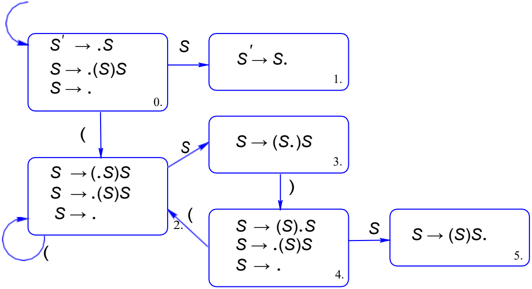
State	Input			Goto
	n	$+$	$\$$	E
0	$s2$			1
1		$s3$	*accept*	
2		$r\,(E \to n)$	$r\,(E \to n)$	
3	$s4$			
4		$r\,(E \to E + n)$	$r\,(E \to E + n)$	

- SLR(1) Parsing actions with input $n + n + n$

	Parsing stack	Input	Action
1	$\$\ 0$	$n + n + n\$$	*shift* 2
2	$\$\ 0\ n\ 2$	$+ n + n\$$	*reduce* $E \to n$
3	$\$\ 0\ E\ 1$	$+ n + n\$$	*shift* 3
4	$\$\ 0\ E\ 1\ +\ 3$	$n + n\$$	*shift* 4
5	$\$\ 0\ E\ 1\ +\ 3\ n\ 4$	$+ n\$$	*reduce* $E \to E + n$
6	$\$\ 0\ E\ 1$	$+ n\$$	*shift* 3
7	$\$\ 0\ E\ 1\ +\ 3$	$n\$$	*shift* 4
8	$\$\ 0\ E\ 1\ +\ 3\ n\ 4$	$\$$	*reduce* $E \to E + n$
9	$\$\ 0\ E\ 1$	$\$$	*accept*

$SLR(1)$ parsing example

- Consider the grammar $S' \rightarrow S \quad S \rightarrow (\; S \;) \; S \mid \epsilon$.
- The *DFA* of sets of LR(0) items is given by:



- Note that follow(S) = {) ,\$ }

SLR(1) parse of ()()

- Parsing Table:

State	*Input*			*Goto*
	(	)	$	*S*
0	*s*2	$r\,(S \to \varepsilon)$	$r\,(S \to \varepsilon)$	1
1			*accept*	
2	*s*2	$r\,(S \to \varepsilon)$	$r\,(S \to \varepsilon)$	3
3		*s*4		
4	*s*2	$r\,(S \to \varepsilon)$	$r\,(S \to \varepsilon)$	5
5		$r\,(S \to (S)S\,)$	$r\,(S \to (S)S\,)$	

- Parsing actions with input ()()

	Parsing stack	*Input*	*Action*
1	$ 0	()()$	*shift* 2
2	$ 0 (2	)()$	*reduce* $S \to \varepsilon$
3	$ 0 (2 *S* 3	()$	*shift* 4
4	$ 0 (2 *S* 3) 4	()$	*shift* 2
5	$ 0 (2 *S* 3) 4 (2	)$	*reduce* $S \to \varepsilon$
6	$ 0 (2 *S* 3) 4 (2 *S* 3	$	*shift* 4
7	$ 0 (2 *S* 3) 4 (2 *S* 3) 4	$	*reduce* $S \to \varepsilon$
8	$ 0 (2 *S* 3) 4 (2 *S* 3) 4 *S* 5	$	*reduce* $S \to (S)S$
9	$ 0 (2 *S* 3) 4 *S* 5	$	*reduce* $S \to (S)S$
10	$ 0 *S* 1	$	*accept*

Disambiguating rules for parsing conflicts

- *shift-reduce* have a natural disambiguating rule: prefer the *shift* over the *reduce*.
- *reduce-reduce* conflicts are more complex to resolve – they usually require the grammar to be altered.
- Preferring the *shift* over the *reduce* in the dangling-else ambiguity, leads to incorporating the most-closely-nested-if rule.
- The grammar with the following productions is ambiguous:

$$\begin{array}{lcl} \textit{statement} & \rightarrow & \textit{if-statement} \mid \texttt{other} \\ \textit{if-statement} & \rightarrow & \texttt{if}\ (\textit{exp})\ \textit{statement} \mid \\ & & \texttt{if}(\textit{exp})\textit{statement}\ \texttt{else}\ \textit{statement} \\ \textit{exp} & \rightarrow & 0 \mid 1 \end{array}$$

- We will consider the simpler grammar:

$$\begin{array}{lcl} S & \rightarrow & I \mid \texttt{other} \\ I & \rightarrow & \texttt{if}\ S \mid \texttt{if}\ S\ \texttt{else}\ S \end{array}$$

Disambiguating a *shift-reduce* conflict

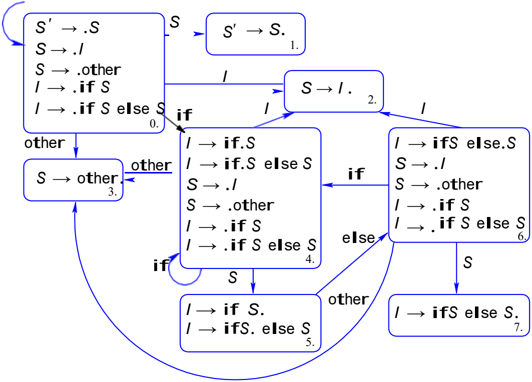
- Consider the grammar:

$$
\begin{array}{lcl}
S & \rightarrow & I \mid \texttt{other} \\
I & \rightarrow & \texttt{if}\ S \mid \texttt{if}\ S\ \texttt{else}\ S
\end{array}
$$

- Since $follow(I) = \{\$, \texttt{else}\}$, there is a shift-reduce conflict in state 5 in the DFA of LR(0) items below.

- The complete item $I \rightarrow \texttt{if}\ S.$ implies a reduction if the next input is **else** or \$, while the item $I \rightarrow \texttt{if}\ S.\texttt{else}\ S$ implies a shift when the next input is **else**

- The DFA of LR(0) items:



SLR(1) table without conflicts

- The rules are numbered:

(1)	$S \rightarrow I$
(2)	$S \rightarrow$ **other**
(3)	$I \rightarrow$ **if** S
(4)	$I \rightarrow$ **if** S **else** S

- The *SLR*(1) parse table in which we prefer the shift over the reduce in state 5:

State	*Input*				*Go to*	
	if	**else**	**other**	$	*S*	*I*
0	*s*4		*s*3		1	2
1				*accept*		
2		*r*1	*r*1			
3		*r*2	*r*2			
4	*s*4		*s*3		5	2
5		*s*6		*r*3		
6	*s*4		*s*3		7	2
7		*r*4		*r*4		

Limits of *SLR*(1) parsing power

- Consider the grammar:

$$\begin{array}{l} stmt \longrightarrow call\text{-}stmt \mid assign\text{-}stmt \\ call\text{-}stmt \longrightarrow \texttt{identifier} \\ assign\text{-}stmt \longrightarrow var \ \texttt{:=} \ exp \\ var \longrightarrow var \ \texttt{[} \ exp \ \texttt{]} \mid \texttt{identifier} \\ exp \longrightarrow var \mid \texttt{number} \end{array}$$

- We will show that the following simplified version of the previous grammar is not SLR(1):

$$\begin{array}{l} S \longrightarrow \texttt{id} \mid V \ \texttt{:=} \ E \\ V \longrightarrow \texttt{id} \\ E \longrightarrow V \mid \texttt{n} \end{array}$$

Limits of $SLR(1)$ parsing power

- Simplified grammar:

$$\begin{aligned} S &\to \mathbf{id} \mid V := E \\ V &\to \mathbf{id} \\ E &\to V \mid \mathrm{n} \end{aligned}$$

- The start state of the *DFA* of sets of LR(0) items contains:

$$\begin{aligned} S' &\to .S \\ S &\to .\mathbf{id} \\ S &\to .V := E \\ V &\to .\mathbf{id} \end{aligned}$$

- The start state has a *shift* transition on `id` to the state:

$$\begin{aligned} S &\to \mathbf{id}. \\ V &\to \mathbf{id}. \end{aligned}$$

- $follow(S) = \{\$\}$ and $follow(V) = \{:=, \$\}$. On getting the input token \$ the $SLR(1)$ parser will try to reduce by both the rules $S \to \mathtt{id}$ and $V \to \mathtt{id}$ – this is a *reduce-reduce* conflict.
- We conclude that the above grammar is not SLR(1).

Finite automata of $LR(1)$ items

- $LR(1)$ parsing uses a *DFA* of LR(1) items.
- The items are called $LR(1)$ items because they include a single lookahead token.
- $LR(1)$ items are written:

$$[A \to \alpha.\beta, a]$$

where $A \to \alpha.\beta$ is an $LR(0)$ item, and a is the lookahead token.

Transitions between $LR(1)$ items

- There are several similarities with *DFA*s of $LR(0)$ items. The *DFA* states are also built from ϵ-closures.
- However, transitions between $LR(1)$ items must keep track of the lookahead token.
- Normal, i.e. non-ϵ-transitions, are quite similar to those in *DFA*s of $LR(0)$ items.
- The major difference lies in the definition of ϵ-*transitions*.
- Given an $LR(1)$ item, $[A \rightarrow \alpha.X\gamma, a]$, where X is a terminal or a nonterminal, there is a transition on X to the item $[A \rightarrow \alpha X .\gamma, a]$.
- Given an $LR(1)$ item, $[A \rightarrow \alpha.B\gamma, a]$, where B is a nonterminal, there are ϵ-transitions to items $[B \rightarrow .\beta, b]$ for every production $B \rightarrow \beta$ and for every token $b \in first(\gamma a)$.
- Only ϵ-transitions create new lookaheads.

DFA of sets of *LR*(1) items for $A \rightarrow (A) \mid a$

$$\boxed{\begin{array}{l} [A' \rightarrow .A,\ \$] \\ [A \rightarrow .(A),\ \$] \\ [A \rightarrow .a,\ \$] \qquad\qquad 0. \end{array}}$$

- *State* 2: There is a transition on '(' leaving *State* 0 to the *LR*(1) item $[A \rightarrow (.A),\ \$]$.
- There are ϵ-transitions from the item $[A \rightarrow (.A), \$]$ to $[A \rightarrow .(A),)]$ and to $[A \rightarrow .\mathrm{a},)]$, since *first*( )$) = {)}.
- The complete *State* 2 is:

$$\boxed{\begin{array}{l} [A \rightarrow (.A),\ \$] \\ [A \rightarrow .(A),\ )] \\ [A \rightarrow .a,\ )] \qquad\qquad 2. \end{array}}$$

DFA of sets of $LR(1)$ items for $A \rightarrow (A) | a$

- *State* 3: We get this state by using a transition on 'a', from *State* 0 on $[A \rightarrow .\mathrm{a}, \$]$ to $[A \rightarrow \mathrm{a}., \$]$

$[A \rightarrow a., \$]$ 3.

- This completes the states that we obtain by transitions from *State* 0.
- *State* 4: We have a transition on A from *State* 2 to the state containing $[A \rightarrow (A.), \$]$.

$[A \rightarrow (.A), \$]$
$[A \rightarrow .(A),)]$
$[A \rightarrow .a,)]$ 2.

$[A \rightarrow (A.), \$]$ 4.

DFA of sets of $LR(1)$ items for $A \rightarrow (A) | a$

$[A \longrightarrow (.A), \$]$

$[A \longrightarrow .(A),)]$

$[A \longrightarrow .a,)]$

2.

- *State* 5: We obtain this state by a transition on '(' from state 2 to $[A \rightarrow (.A),)]$

$[A \longrightarrow (.A),)]$

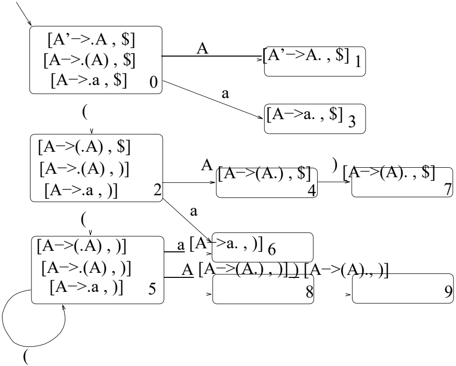
$[A \longrightarrow .(A),)]$

$[A \longrightarrow .a,)]$

5.

DFA of sets of *LR*(1) items for $A \to (A) | a$

- By completing the calculations, we obtain the following DFA of sets of LR(1) items:



The general $LR(1)$ parsing algorithm

Let s be the current state, i.e. the state on top of the stack.
The actions are defined as follows:

1. If s contains a $LR(1)$ item of the form $[A \to \alpha.X\beta, \mathrm{a}]$, where X is the next terminal in the input stream, then *shift* X onto the stack and push the state containing the $LR(1)$ item $[A \to \alpha X.\beta, \mathrm{a}]$.
2. If s contains the complete $LR(1)$ item $[A \to \gamma., \mathrm{a}]$ and the next terminal in the input stream is a, then *reduce* by the rule $A \to \gamma$
3. If the next input token is not accommodated by (1) or (2), then an *error* is declared.

$LR(1)$ grammar

A grammar is an $LR(1)$ *grammar* if the application of the $LR(1)$ parsing rules do not result in an ambiguity.

Thus a grammar is an $LR(1)$ *grammar* $\Longleftrightarrow$

1. For any nonterminal X, we do not have two items of the form $[A \to \alpha.X\beta, \mathrm{a}]$ and $[B \to \gamma., X]$ in the same state of the DFA of $LR(1)$ items.
   A violation of this condition is a *shift-reduce* conflict.

2. It is not the case that there are two complete $LR(1)$ items of the form $[A \to \alpha., \mathrm{a}]$ and $[A \to \beta., \mathrm{a}]$ in the same state of the DFA of $LR(1)$ items, otherwise it would lead to a *reduce-reduce* conflict.

$LR(1)$ parse table for $A \rightarrow (A)|a$

- Number the productions as follows:

$$\begin{array}{ll} (0) & A' \rightarrow A \\ (1) & A \rightarrow (A) \text{ and} \\ (2) & A \rightarrow a \end{array}$$

- The $LR(1)$parse table obtained from the DFA of LR(1) items is given by:

State	*Input*				*Go to*
	(	a	)	\$	*A*
0	*s*2	*s*3			1
1				*accept*	
2	*s*5	*s*6			4
3				*r* 2	
4			*s*7		
5	*s*5	*s*6			8
6			*r* 2		
7				*r* 1	
8			*s*9		
9			*r* 1		

General $LR(1)$ parsing

- The grammar

$$\begin{array}{lll} S & \rightarrow & \mathtt{id} \mid V{:=}E \\ V & \rightarrow & \mathtt{id} \\ E & \rightarrow & V \mid \mathtt{n} \end{array}$$

  is not $SLR(1)$.

- We construct its *DFA* of sets of $LR(1)$ items.
- The start state is the ϵ-closure of the $LR(1)$ item $[S' \rightarrow .S, \$]$. Thus it also contains the $LR(1)$ items $[S \rightarrow .\mathtt{id}, \$]$ and $[S \rightarrow .V{:=}E, \$]$.
- The last item, in turn, gives rise to the $LR(1)$ item $[V \rightarrow .\mathtt{id},\ {:=}]$.

$$\boxed{\begin{array}{l} [S' \longrightarrow .S, \$] \\ [S \longrightarrow .\mathtt{id}, \$] \\ [S \longrightarrow .V{:=}E, \$] \\ [V \longrightarrow .\mathtt{id}, {:=}] \quad 0. \end{array}}$$

General $LR(1)$ parsing

- Consider *state* 0:

$$[S' \to .S, \$]$$
$$[S \to .\mathbf{i}\mathrm{d}, \$]$$
$$[S \to .V := E, \$]$$
$$[V \to .\mathbf{i}\mathrm{d}, :=]$$
0.

A transition from *state* 0 on 'S' goes to *state* 1:

$$[S' \to S., \$]$$
1.

- *State* 0 has a transition on '**i**d' to *state* 2:

$$[S \to \mathbf{i}\mathrm{d}., \$]$$
$$[V \to \mathbf{i}\mathrm{d}., :=]$$
2.

- *State* 0 has a transition on 'V' to *state* 3:

$$[S \to V.:=E, \$]$$
3.

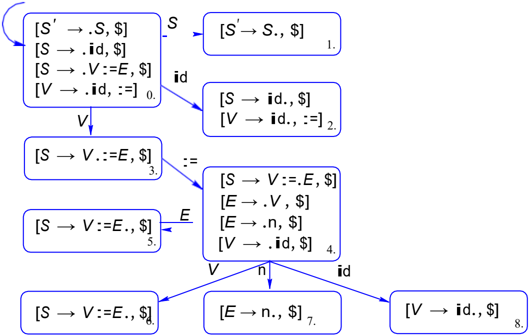
General $LR(1)$ parsing

- The third state has a transition on '$:=$' to the closure of the item $[S \to V := .E, \$]$.
  The two items $[E \to .V, \$]$ and $[E \to .n, \$]$ must be added.
  Since we have $[E \to .V, \$]$, we must also add the item $[V \to .id, \$]$.

$$\boxed{[S \longrightarrow V . := E, \$] \quad 3.}$$

- Each of these items in *state* 4 has the general form $[A \to \alpha.X\beta]$, and each of them transition to a state with the single item $[A \to \alpha X.\beta]$ in it, where $X \in \{E, V, \mathrm{n}, \mathrm{id}\}$.
- *State* 2 gave rise to a parsing conflict in the $SLR(1)$ parser.
  The $LR(1)$ items now clearly distinguish between the two reductions by their lookaheads:
  Select $S \to \mathrm{id}$ on '\$' and $V \to \mathrm{id}$ on '$:=$'.

General $LR(1)$ parsing



LALR(1) parsing

- In the *DFA* of sets of *LR*(1) items many states differ only in some of the lookaheads of their items.
- The *DFA* of sets of *LR*(0) items of the grammar
  $A' \rightarrow A,\ A \rightarrow (A) \mid \mathrm{a}$
  has only 6 states while its *DFA* of sets of *LR*(1) items has 10 items.
- In the *DFA* of sets of *LR*(1) items states 2 and 5, 4 and 8, 7 and 9, 3 and 6, differ only in lookaheads.
- e.g. the item $[A \rightarrow (.A), \$]$ from *state* 2 differs from the item $[A \rightarrow (.A),)]$ from *state* 5 only in its lookahead.

LALR(1) parsing

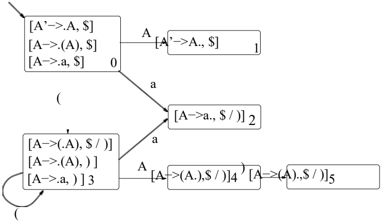
- The *LALR*(1) algorithm combine states that are the same if we ignore the lookahead symbols, by using sets of lookaheads in the items, e.g. $[A \rightarrow (.A), \$/)]$.
- The *DFA* of sets of *LALR*(1) items is identical to the corresponding *DFA* of sets of *LR*(0) items, except that the former includes sets of lookahead items.
- The *LALR*(1) parsing algorithm preserves the benefit of the smaller *DFA* of sets of *LR*(0) items with the advantage of some of the benefit of *LR*(1) parsing over *SLR*(1) parsing.

LALR(1) parsing

- We construct the *DFA* of sets of *LALR*(1) by identifying all states that are identical if we ignore the lookahead symbols.
- Thus each *LALR*(1) item in this *DFA* will have an *LR*(0) item as its first component and a set of lookahead tokens as its second component.
- Multiple lookaheads are separated by '/'.

LALR(1) parsing

- The *DFA* of sets of *LALR*(1) items for $A' \rightarrow A \mid A \rightarrow (A) \mid a$



- The *DFA* is identical to the *DFA* of sets of *LR*(0) items for this grammar, except for lookaheads.

LALR(1) parsing algorithm

- The *LALR*(1) parsing algorithm is identical to the general *LR*(1) parsing algorithm.
- *Definition*: if no parsing conflicts arise when parsing a grammar with the *LALR*(1) parsing algorithm, the grammar is defined to be an *LALR*(1) *grammar* .
- It is possible for the *LALR*(1) construction to create parsing conflicts that do not exist in general *LR*(1) parsing.

LALR(1) parsing

- Combining *LR*(1) states to form the *DFA* of sets of *LALR*(1) items solves the problem of large parsing tables, but it still requires the entire *DFA* of sets of *LR*(1) items to be computed.
- It is possible to compute the *DFA* of sets of *LALR*(1) items directly from the *DFA* of sets of *LR*(0) items by *propagating lookaheads* which is a relatively simple process.
- Consider the grammar $A' \rightarrow A$, $A \rightarrow (\ A\ ) \mid a$
- Begin constructing lookaheads by adding '\$' to the lookahead of the item $A' \rightarrow A$ in *state* 0.
- The '\$' propagates to the two closure items of '.*A*' By following the three transitions leaving *state* 0, the '\$' propagates to *states* 1, 2, and 3.

LALR(1) parsing

- Continuing with *state* 3 the closure items get the lookahead ')' because in $A \to (.A)$, '$.A$' is followed by ')'.
- The transition of '(' from *state* 3 to itself causes the ')' to propagate to the lookahead of $A \to (.A)$, which now has ')' and '\$' in its lookahead set.
- The transition on "a" from *state* 3 to *state* 2 causes the ')' to be propagated to the lookahead of the item in that state.
- Now the lookahead set ')/\$' propagates to *states* 4 and 5.
- Thus we have demonstrated how to build the *DFA* of sets of *LALR*(1) directly from the *DFA* of sets of *LR*(0) items.

The hierarchy of LR grammars

- *LR*(0) grammars are *SLR*(1) and there are *SLR*(1) grammars that are not *LR*(0) grammars.
- *SLR*(1) grammars are *LALR*(1) and there are *LALR*(1) grammars that are not *SLR*(1) grammars.
- *LALR*(1) grammars are *LR*(1) and there are *LR*(1) grammars that are not *LALR*(1).