CS 335: Bottom-Up Parsing

Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2023-24-II

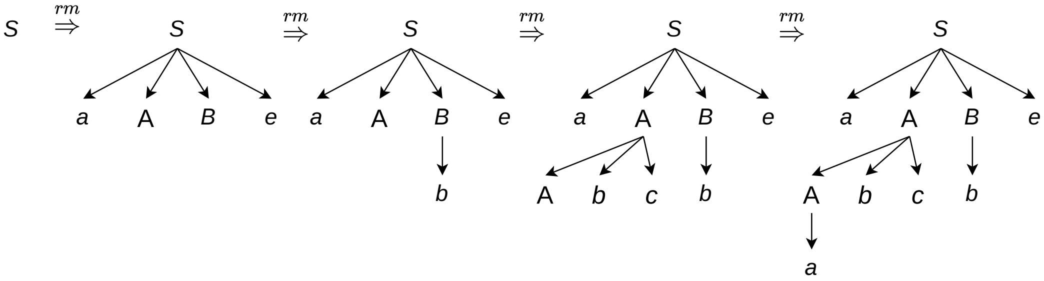
Rightmost Derivation of *abbcde*

Grammar

$$S \rightarrow aABe$$
$$A \rightarrow Abc \mid b$$
$$B \rightarrow d$$

Input string: *abbcde*

$$S \rightarrow aABe$$
$$\rightarrow aAde$$
$$\rightarrow aAbcde$$
$$\rightarrow abbcde$$



Bottom-Up Parsing

Definition

Bottom-up parsing constructs the parse tree starting from the leaves and working up toward the root

Grammar

$$S \rightarrow aABe$$
$$A \rightarrow Abc \mid b$$
$$B \rightarrow d$$

Input string: *abbcde*

$S \rightarrow aABe$	$abbcde$
$\rightarrow aAde$	$\rightarrow aAbcde$
$\rightarrow aAbcde$	$\rightarrow aAde$
$\rightarrow abbcde$	$\rightarrow aABe$
	$\rightarrow S$

↑ reverse of rightmost derivation

Bottom-Up Parsing

Grammar

$S \rightarrow aABe$

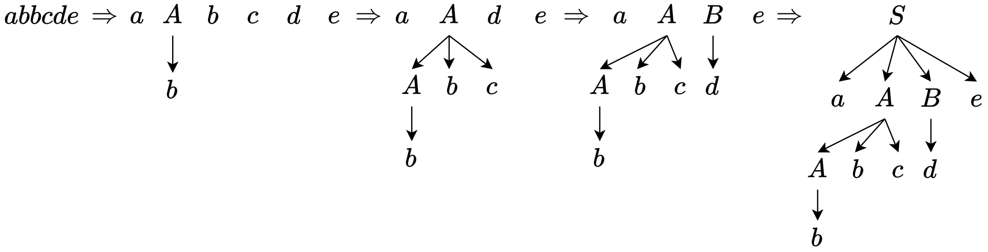
$A \rightarrow Abc \mid b$

$B \rightarrow d$

Input string: *abbcde*

$S \rightarrow aABe$	$abbcde$
$\rightarrow aAde$	$\rightarrow aAbcde$
$\rightarrow aAbcde$	$\rightarrow aAde$
$\rightarrow abbcde$	$\rightarrow aABe$
	$\rightarrow S$

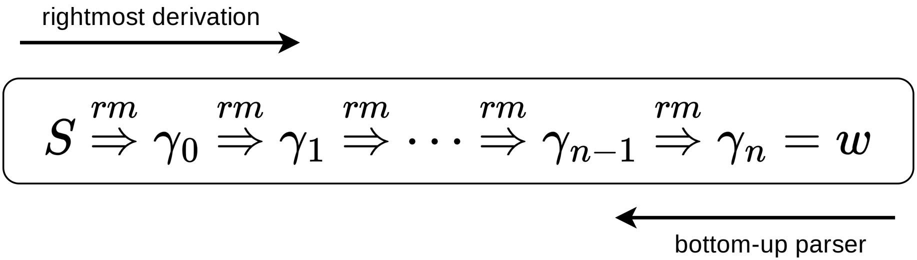
↑ reverse of rightmost derivation



Reduction

Bottom-up parsing reduces a string *w* to the start symbol *S*

At each reduction step, a chosen substring that is the RHS (or body) of a production is replaced by the LHS (or head) nonterminal



rightmost derivation

$$\boxed{S \overset{rm}{\Rightarrow} \gamma_0 \overset{rm}{\Rightarrow} \gamma_1 \overset{rm}{\Rightarrow} \cdots \overset{rm}{\Rightarrow} \gamma_{n-1} \overset{rm}{\Rightarrow} \gamma_n = w}$$

bottom-up parser

Handle

- Handle is a substring that matches the body of a production
- Reducing the handle is one step in the reverse of the rightmost derivation

$$E \longrightarrow E + T \mid T$$
$$T \longrightarrow T * F \mid F$$
$$F \longrightarrow (E) \mid \textbf{id}$$

Right sentential form	Handle	Reducing Production
$\textbf{id}_1 * \textbf{id}_2$	$\textbf{id}_1$	$F \longrightarrow \textbf{id}$
$F * \textbf{id}_2$	F	$T \longrightarrow F$
$T * \textbf{id}_2$	$\textbf{id}_2$	$F \longrightarrow \textbf{id}$
$T * F$	$T * F$	$T \longrightarrow T * F$
T	T	$E \longrightarrow T$
E		

Handle

- Handle is a substring that matches the body of a production
- Reducing the handle is one step in the reverse of the rightmost derivation

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid \textbf{id}$$

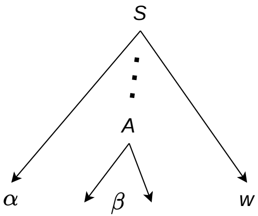
Right sentential form	Handle	Reducing Production
$\textbf{id}_1 * \textbf{id}_2$	$\textbf{id}_1$	$F \to \textbf{id}$
$F * \textbf{id}_2$	F	$T \to F$
$T * \textbf{id}_2$	$\textbf{id}_2$	$F \to \textbf{id}$
$T * F$	$T * F$	$T \to T * F$
T	T	$E \to T$
E		

Although T is the body of the production $E \to T$, T is not a handle in the sentential form $T * \textbf{id}_2$

The leftmost substring that matches the body of some production need not be a handle

Handle

- If $S \overset{*}{\underset{\text{rm}}{\Longrightarrow}} \alpha A w \underset{\text{rm}}{\Longrightarrow} \alpha\beta w$, then $A \rightarrow \beta$ is a handle of $\alpha\beta w$
- String w right of a handle must contain only terminals



A handle $A \rightarrow \beta$ in the parse tree for $\alpha\beta w$

- If grammar G is unambiguous, then every right sentential form has only one handle
- If G is ambiguous, then there can be more than one rightmost derivation of $\alpha\beta w$

Shift-Reduce Parsing

Shift-Reduce Parsing

- The input string being parsed consists of two parts
  - Left part is a string of terminals and nonterminals, and is stored in a stack
  - Right part is a string of terminals to be read from an input buffer
  - Bottom of the stack and end of the input are represented by $

- Shift-reduce parsing is a type of bottom-up parsing with **two primary actions**, shift and reduce
  - Shift-Reduce actions

    Shift Shift the next input symbol from the right string onto the top of the stack

    Reduce Identify a string on top of the stack that is the body of a production and replace the body with the head
  - Other actions are accept and error

Shift-Reduce Parsing

- Initial

Stack	**Input**
\$	w\$

Shift $\Downarrow^*$ Reduce

- Goal

Stack	**Input**
\$$S$	\$

Example of Shift-Reduce Parsing

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

Stack	Input	Action
$\$$	$\mathbf{id}_1 * \mathbf{id}_2\$$	Shift
$\$\mathbf{id}_1$	$*\mathbf{id}_2\$$	Reduce by $F \rightarrow \mathbf{id}$
$\$F$	$*\mathbf{id}_2\$$	Reduce by $T \rightarrow F$
$\$T$	$*\mathbf{id}_2\$$	Shift
$\$T*$	$\mathbf{id}_2\$$	Shift
$\$T * \mathbf{id}_2$	$\$$	Reduce by $F \rightarrow \mathbf{id}$
$\$T * F$	$\$$	Reduce by $T \rightarrow T * F$
$\$T$	$\$$	Reduce by $E \rightarrow T$
$\$E$	$\$$	Accept

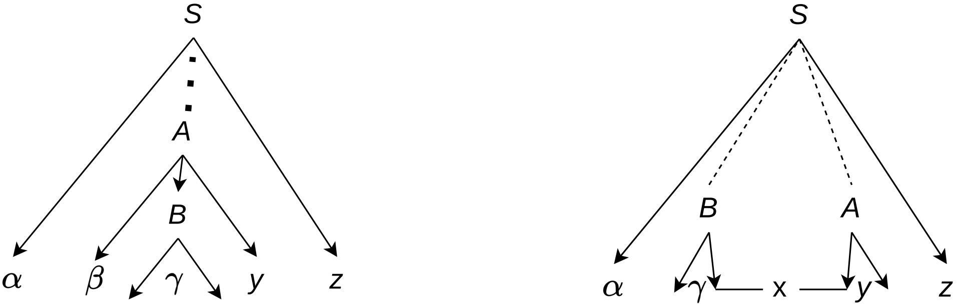
or report an error in case of syntax error

Handle on Top of Stack

Is the following scenario possible?

Stack	Input	Action
$\ldots$		
$\$\alpha\beta\gamma$	$w\$$	Reduce by $A \rightarrow \gamma$
$\$\alpha\beta A$	$w\$$	Reduce by $B \rightarrow \beta$
$\$\alpha B A$	$w\$$	$\ldots$
$\ldots$		

Possible Choices in Rightmost Derivation



1. $S \underset{rm}{\Longrightarrow} \alpha Az \underset{rm}{\Longrightarrow} \alpha\beta Byz \underset{rm}{\Longrightarrow} \alpha\beta\gamma yz$

2. $S \underset{rm}{\Longrightarrow} \alpha BxAz \underset{rm}{\Longrightarrow} \alpha Bxyz \underset{rm}{\Longrightarrow} \alpha\gamma xyz$

Handle on Top of Stack

Is the following scenario possible?

Stack	Input	Action
…		
$\$\alpha\beta\gamma$	$w\$$	Reduce by $A \rightarrow \gamma$

Handle will always eventually appear on **top of the stack**, never inside

Shift-Reduce Actions

Shift	shift the next input symbol from the right string onto the top of the stack
Reduce	identify a string on top of the stack that is the body of a production, and replace the body with the head

How do you decide when to shift and when to reduce?

Steps in Shift-Reduce Parsers

General shift-reduce technique

- If there is no handle on the stack, then shift
- If there is a handle on the stack, then reduce

Bottom-up parsing is essentially the process of identifying handles and reducing them

- Different bottom-up parsers differ in the way they **detect** handles

Challenges in Bottom-up Parsing

Which action do you pick when both shift and reduce are valid?

Implies a shift-reduce conflict

Which rule to use if reduction is possible by more than one rule?

Implies a reduce-reduce conflict

Example of a Shift-Reduce Conflict

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

$\mathbf{id} + \mathbf{id} * \mathbf{id}$

Stack	Input	Action
$\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	Shift
$\ldots$		
$\$E + E$	$*\mathbf{id}\$$	Reduce by $E \rightarrow E + E$
$\$E$	$*\mathbf{id}\$$	Shift
$\$E*$	$\mathbf{id}\$$	Shift
$\$E * \mathbf{id}$	$\$$	Reduce by $E \rightarrow \mathbf{id}$
$\$E * E$	$\$$	Reduce by $E \rightarrow E * E$
$\$E$	$\$$	

$c + C$

Stack	Input	Action
$\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	Shift
$\ldots$		
$\$E + E$	$*\mathbf{id}\$$	Shift
$\$E + E*$	$\mathbf{id}\$$	Shift
$\$E + E * \mathbf{id}$	$\$$	Reduce by $E \rightarrow \mathbf{id}$
$\$E + E * E$	$\$$	Reduce by $E \rightarrow E * E$
$\$E + E$	$\$$	Reduce by $E \rightarrow E + E$
$\$E$	$\$$	

Resolving Shift-Reduce Conflict

$$
\begin{aligned}
\mathit{Stmt} \longrightarrow\ & \textbf{if}\ \mathit{Expr}\ \textbf{then}\ \mathit{Stmt} \\
\mid\ & \textbf{if}\ \mathit{Expr}\ \textbf{then}\ \mathit{Stmt}\ \textbf{else}\ \mathit{Stmt} \\
\mid\ & \mathit{other}
\end{aligned}
$$

Stack	**Input**	**Action**
...		
$ . . . **if** *Expr* **then** *Stmt*	**else** . . .	

What is a correct thing to do for this grammar – shift or reduce? We can prioritize shifts.

Reduce-Reduce Conflict

$$M \rightarrow R + R \mid R + c \mid R$$
$$R \rightarrow c$$

$c + c$

Stack	Input	Action
$	$c + c\$$	Shift
$\$c$	$+c\$$	Reduce by $R \rightarrow c$
$\$R$	$+c\$$	Shift
$\$R+$	$c\$$	Shift
$\$R + c$	$	Reduce by $R \rightarrow c$
$\$R + R$	$	Reduce by $R \rightarrow R + R$
$\$M$	$	

id $+$ **id** $*$ **id**

Stack	Input	Action
$	$c + c\$$	Shift
$\$c$	$+c\$$	Reduce by $R \rightarrow c$
$\$R$	$+c\$$	Shift
$\$R+$	$c\$$	Shift
$\$R + c$	$	Reduce by $M \rightarrow R + c$
$\$M$	$	

LR Parsing

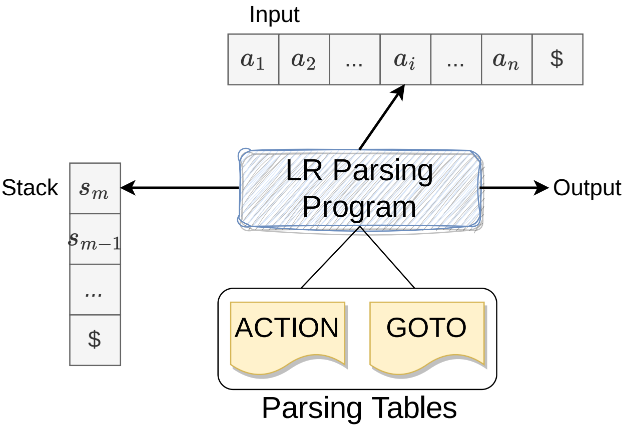
LR(k) Parsing

- Popular bottom-up parsing scheme
  - L is for left-to-right scan of input, R is for **reverse of rightmost** derivation, k is the number of lookahead symbols
- LR parsers are table-driven, like the non-recursive LL parser
- LR grammar is one for which we can construct an LR parsing table
- Popularity of LR Parsing
  - + Most general non-backtracking shift-reduce parsing method
  - + Can recognize almost all language constructs with CFGs
  - + Works for a superset of grammars parsed with predictive or LL parsers

LR(k) Parsing

- Popular bottom-up parsing scheme
  - L is for left-to-right scan of input, R is for **reverse of rightmost** derivation, k is the number of lookahead symbols
- LR parsers are table-driven, like the non-recursive LL parser
- LR grammar is one for which we can construct an LR parsing table
- Popularity of LR Parsing
  - + Most general non-backtracking shift-reduce parsing method
  - + Can recognize almost all language constructs with CFGs
  - + Works for a superset of grammars parsed with predictive or LL parsers
    - LL(k) parsing predicts which production to use having seen only the first k tokens of the right-hand side
    - LR(k) parsing can decide after it has seen input tokens corresponding to the entire right-hand side of the production

Block Diagram of LR Parser



The LR parsing driver is the same for all LR parsers, only the parsing tables (i.e., ACTION and GOTO) change across parser types

Steps in LR Parsing

- Remember the basic questions: **when to shift** and **when to reduce**!
- An LR parser makes shift-reduce decisions by maintaining states
- Information is encoded in a DFA constructed using a canonical LR(0) collection
  1. Augmented grammar G' with new start symbol S' and rule $S' \rightarrow S$
  2. Define helper functions Closure() and Goto()

LR(0) Item

- An LR(0) item of a grammar G is a production of G with a dot (•) at some position in the body
- An item indicates how much of a production we have seen
  - Symbols on the left of "•" are already on the stack
  - Symbols on the right of "•" are expected in the input
- $A \rightarrow \bullet XYZ$ indicates that we expect a string derivable from XYZ next in the input
- $A \rightarrow X \bullet YZ$ indicates that we saw a string derivable from X in the input, and we expect a string derivable from YZ next in the input
- $A \rightarrow \epsilon$ generates only one item $A \rightarrow \bullet$

Production	**Items**
$A \rightarrow XYZ$	$A \rightarrow \bullet XYZ$ $A \rightarrow X \bullet YZ$ $A \rightarrow XY \bullet Z$ $A \rightarrow XYZ \bullet$

Closure Operation

- Let I be a set of items for a grammar G
- Closure(I) is constructed as follows
  - (i) Add every item in I to Closure(I)
  - (ii) If $A \longrightarrow \alpha \bullet B\beta$ is in Closure(I) and $B \longrightarrow \gamma$ is a rule in G, then add $B \longrightarrow \bullet\gamma$ to Closure(I) if not already added
  - (iii) Repeat until no more new items can be added to Closure(I)

A substring derivable from $B\beta$ will have a prefix derivable from B by applying one the B productions

$$\begin{aligned} E^{'} &\longrightarrow E \\ E &\longrightarrow E + T \mid T \\ T &\longrightarrow T * F \mid F \\ F &\longrightarrow (E) \mid \textbf{id} \end{aligned}$$

Suppose $I = \{E^{'} \longrightarrow \bullet E\}$

$$\begin{aligned} \text{Closure}(I) = \{ & E^{'} \longrightarrow \bullet E, \\ & E \longrightarrow \bullet E + T, \\ & E \longrightarrow \bullet T, \\ & T \longrightarrow \bullet T * F, \\ & T \longrightarrow \bullet F, \\ & F \longrightarrow \bullet\, (E)\,, \\ & F \longrightarrow \bullet\textbf{id}\} \end{aligned}$$

Goto Operation

- Suppose I is a set of items and X is a grammar symbol
- Goto(I, X) is the closure of set all items $[A \rightarrow \alpha X \bullet \beta]$ such that $[A \rightarrow \alpha \bullet X\beta]$ is in I
  - If I is a set of items for some valid prefix α, then Goto(I, X) is the set of valid items for prefix αX

Intuitively, Goto(I, X) gives the transition of the state I under input X in the LR(0) automaton

$$\begin{aligned} E' &\rightarrow E \\ E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \textbf{id} \end{aligned}$$

Suppose

$$\begin{aligned} I = \{E' &\rightarrow E\bullet, \\ E &\rightarrow E \bullet +T\} \end{aligned}$$

$$\begin{aligned} \text{Goto}(I, +) = \{E &\rightarrow E + \bullet T, \\ T &\rightarrow \bullet T * F, \\ T &\rightarrow \bullet F, \\ F &\rightarrow \bullet (E), \\ F &\rightarrow \bullet \textbf{id}\} \end{aligned}$$

Algorithm to Compute LR(0) Canonical Collection

```
$C = \text{Closure}\left(\{[S' \rightarrow \bullet S]\}\right)$
repeat
  for each set of items $I \in C$
    for each grammar symbol $X$
      if $\text{Goto}(I, X) \neq \phi$ and $\text{Goto}(I, X) \notin C$
        add $\text{Goto}(I, X)$ to $C$
until no new sets of items are added to $C$
```

Example Computation of LR(0) Canonical Collection

$$\begin{aligned} I_0 &= \text{Closure}(E' \rightarrow \bullet E) \\ &= \{E' \rightarrow \bullet E, \\ &\quad E \rightarrow \bullet E + T, \\ &\quad E \rightarrow \bullet T, \\ &\quad T \rightarrow \bullet T * F, \\ &\quad T \rightarrow \bullet F, \\ &\quad F \rightarrow \bullet (E), \\ &\quad F \rightarrow \bullet \textbf{id}\} \end{aligned}$$

$$\begin{aligned} I_1 &= \text{Goto}(I_0, E) \\ &= \{E' \rightarrow E \bullet, \\ &\quad E \rightarrow E \bullet + T\} \end{aligned}$$

$$\begin{aligned} I_2 &= \text{Goto}(I_0, T) \\ &= \{E \rightarrow T \bullet, \\ &\quad T \rightarrow T \bullet * F\} \end{aligned}$$

$$\begin{aligned} I_3 &= \text{Goto}(I_0, F) \\ &= \{T \rightarrow F \bullet\} \end{aligned}$$

$$\begin{aligned} I_4 &= \text{Goto}(I_0, \text{`('}) \\ &= \{F \rightarrow (\bullet E), \\ &\quad E \rightarrow \bullet E + T, \\ &\quad E \rightarrow \bullet T, \\ &\quad T \rightarrow \bullet T * F, \\ &\quad T \rightarrow \bullet F, \\ &\quad F \rightarrow \bullet (E), \\ &\quad F \rightarrow \bullet \textbf{id}\} \end{aligned}$$

$$\begin{aligned} I_5 &= \text{Goto}(I_0, \textbf{id}) \\ &= \{F \rightarrow \textbf{id} \bullet\} \end{aligned}$$

$$\begin{aligned} I_6 &= \text{Goto}(I_1, +) \\ &= \{E \rightarrow E + \bullet T, \\ &\quad T \rightarrow \bullet T * F, \\ &\quad T \rightarrow \bullet F, \\ &\quad F \rightarrow \bullet (E), \\ &\quad F \rightarrow \bullet \textbf{id}\} \end{aligned}$$

$$\begin{aligned} I_7 &= \text{Goto}(I_2, *) \\ &= \{T \rightarrow T * \bullet F, \\ &\quad F \rightarrow \bullet (E), \\ &\quad F \rightarrow \bullet \textbf{id}\} \end{aligned}$$

$$\begin{aligned} I_8 &= \text{Goto}(I_4, E) \\ &= \{E \rightarrow E \bullet + T, \\ &\quad F \rightarrow (E \bullet)\} \end{aligned}$$

$$\begin{aligned} I_9 &= \text{Goto}(I_6, T) \\ &= \{E \rightarrow E + T \bullet, \\ &\quad T \rightarrow T \bullet * F\} \end{aligned}$$

$$\begin{aligned} I_{10} &= \text{Goto}(I_7, F) \\ &= \{T \rightarrow T * F \bullet\} \end{aligned}$$

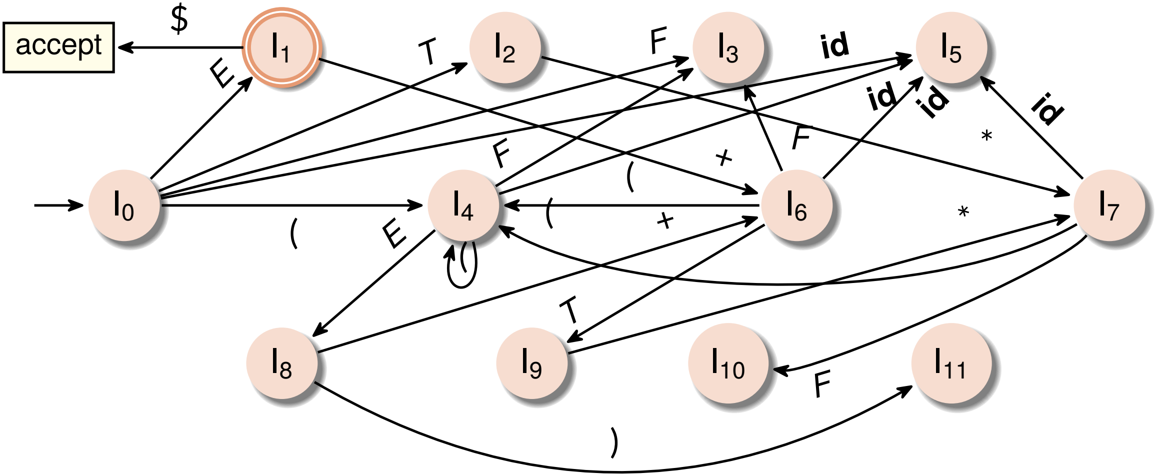
$$\begin{aligned} I_{11} &= \text{Goto}(I_8, \text{`)'}) \\ &= \{F \rightarrow (E) \bullet\} \end{aligned}$$

$$\begin{aligned} I_2 &= \text{Goto}(I_4, T) \\ I_3 &= \text{Goto}(I_4, F) \\ I_4 &= \text{Goto}(I_4, \text{`('}) \\ I_5 &= \text{Goto}(I_4, \textbf{id}) \\ I_3 &= \text{Goto}(I_6, F) \\ I_4 &= \text{Goto}(I_6, \text{`('}) \\ I_5 &= \text{Goto}(I_6, \textbf{id}) \\ I_4 &= \text{Goto}(I_7, \text{`('}) \\ I_5 &= \text{Goto}(I_7, \textbf{id}) \\ I_6 &= \text{Goto}(I_8, +) \\ I_7 &= \text{Goto}(I_9, *) \end{aligned}$$

LR(0) Automaton

- Canonical LR(0) collection is used for constructing the LR(0) automaton for parsing
- States represent sets of LR(0) items in the canonical LR(0) collection
    - Start state is $\text{Closure}(\{[S^{'} \rightarrow \bullet S]\})$, where $S^{'}$ is the start symbol of the augmented grammar
    - State j refers to the state corresponding to the set of items I_j
- By construction, all transitions to state j is for the same symbol X
    - Each state, except the start state, has a unique grammar symbol associated with it

LR(0) Automaton



Use of LR(0) Automaton

- How can the LR(0) automaton help with shift-reduce decisions?
- Suppose string γ of grammar symbols takes the automaton from start state S_0 to state S_j
    - Shift on next input symbol a if S_j has a transition on a
    - Otherwise, reduce
        - Items in state S_j help decide which production to use

Structure of LR Parsing Table

- Assume S_i is top of the stack and a_i is the current input symbol
- Parsing table consists of two parts: an ACTION and a GOTO function
- ACTION table is indexed by state and terminal symbols; ACTION$[S_i, a_i]$ can have four values
    - (i) Shift a_i to the stack, go to state S_j
    - (ii) Reduce by rule k
    - (iii) Accept
    - (iv) Error (empty cell in the table)
- GOTO table is indexed by state and nonterminal symbols

Constructing LR(0) Parsing Table

(i) Construct LR(0) canonical collection $C = \{I_0, I_1, \ldots, I_n\}$ for grammar G'

(ii) State i is constructed from I_i
  (a) If $[A \rightarrow \alpha \bullet A\beta] \in I_i$ and $\text{GOTO}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ = "Shift j"
    - *sj* means shift and stack state j
  (b) If $[A \rightarrow \alpha\bullet] \in I_i$, then set $\text{ACTION}[i, a]$ = "Reduce by $A \rightarrow \alpha$" for all a
    - *rj* means reduce by rule \$$j$
  (c) If $[S' \rightarrow S\bullet] \in I_i$, then set $\text{ACTION}[i, \$]$ = "Accept"

(iii) If $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}[i, A] = j$

(iv) All entries left undefined are "errors"

LR(0) Parsing Table

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

Rule #	Rule
0	$E' \rightarrow E$
1	$E \rightarrow E + T$
2	$E \rightarrow T$
3	$T \rightarrow T * F$
4	$T \rightarrow F$
5	$F \rightarrow (E)$
6	$F \rightarrow \textbf{id}$

LR Parser Configurations

- A LR parser configuration is a pair $\langle s_0 s_1 \ldots s_m,\ a_i a_{i+1} \ldots a_n \$ \rangle$
  - The left half is stack content, and the right half is the remaining input
- Configuration represents the right sentential form $X_1 X_2 \ldots X_m a_i a_{i+1} \ldots a_n$

LR Parsing Algorithm

(i) If $\text{ACTION}[s_m, a_i] = sj$, then the new configuration is $\langle s_0 s_1 \ldots s_m s_j,\ a_{i+1} \ldots a_n \rangle$

(ii) If $\text{ACTION}[s_m, a_i] =$ reduce $A \rightarrow \beta$, then the new configuration is $\langle s_0 s_1 \ldots s_{m-r} s,\ a_i a_{i+1} \ldots a_n \rangle$, where $r = |\beta|$ and $s = \text{GOTO}[s_{m-r}, A]$

(iii) If $\text{ACTION}[s_m, a_i] =$ Accept, then parsing is successful

(iv) If $\text{ACTION}[s_m, a_i] =$ error, then parsing has discovered an error

LR Parsing Program

```
Let $a$ be the first symbol in $w\$$
while (1)
  Let $s$ be the top of the stack
  if $\text{ACTION}[s,a]$ == shift $t$
    push $t$ onto the stack
    let $a$ be the next input symbol
  else if $\text{ACTION}[s,a]$ = reduce $A \rightarrow \beta$
    // Reduce with the production $A \rightarrow \beta$
    pop $|\beta|$ symbols of the stack
    let state $t$ now be the top of the stack
    push $\text{GOTO}[t,A]$ onto the stack
  else if $\text{ACTION}[s,a] == \text{Accept}$
    break // parsing is complete
  else
    invoke error recovery
```

Shift-Reduce Parser with LR(0) Automaton

Stack	Input	Action
$\$0$	$\textbf{id} * \textbf{id}\$$	Shift
$\$0\, \textbf{id}\, 5$	$* \textbf{id}\$$	Reduce by $F \longrightarrow \textbf{id}$
$\$0\, F\, 3$	$* \textbf{id}\$$	Reduce by $T \longrightarrow F$
$\$0\, T\, 2$	$* \textbf{id}\$$	Shift
$\$0\, T\, 2 * 7$	$\textbf{id}\$$	Shift
$\$0\, T\, 2 * 7\, \textbf{id}\, 5$	$\$$	Reduce by $F \longrightarrow \textbf{id}$
$\$0\, T\, 2 * 7\, F\, 10$	$\$$	Reduce by $T \longrightarrow T * F$
$\$0\, T\, 2$	$\$$	Reduce by $E \longrightarrow T$
$\$0\, E\, 1$	$\$$	Accept

popped 5 and pushed 3 because $I_3 = \text{Goto}(I_0, F)$

While the stack consisted of only symbols in the shift-reduce parser, here the stack also contains states from the LR(0) automaton

Viable Prefix

- Consider $E \overset{rm}{\Longrightarrow} T \overset{rm}{\Longrightarrow} T * F \overset{rm}{\Longrightarrow} T * \textbf{id} \overset{rm}{\Longrightarrow} F * \textbf{id} \overset{rm}{\Longrightarrow} \textbf{id} * \textbf{id}$
- Not all prefixes of a right sentential form can appear on the stack
  - **id**$*$ is a prefix of a right sentential form but can never appear on the stack
    - LR parser will not shift past the handle
    - Always reduce by $F \rightarrow \textbf{id}$ before shifting $*$ (see previous slide)
- A **viable prefix** is a prefix of a right sentential form that can appear on the stack of a shift-reduce parser
  - If the stack contains α, then α is a viable prefix if $\exists w$ such that αw is a right sentential form
- There is no error as long as the parser has viable prefixes on the stack
  - The parser has not yet read past the handle, and expects that the remaining input could form a valid sentential form leading to a successful parse

Example of a Viable Prefix

$S \rightarrow X_1X_2X_3X_4$

$A \rightarrow X_1X_2$

Stack	Input
$\$$	$X_1X_2X_3\$$
$\$X_1$	$X_2X_3\$$
$\$X_1X_2$	$X_3\$$
$\$A$	$X_3\$$
$\$AX_3$	$\$$

$X_1X_2X_3$ can never appear on the stack

- Suppose there is a production $A \rightarrow \beta_1\beta_2$, $\alpha\beta_1$ is on the stack, and there is a derivation $S' \overset{*}{\underset{rm}{\Longrightarrow}} \alpha A w \overset{*}{\underset{rm}{\Longrightarrow}} \alpha\beta_1\beta_2 w$
  - $\beta_2 \neq \epsilon$ implies that the handle $\beta_1\beta_2$ is not at the top of the stack yet, so shift
  - $\beta_2 = \epsilon$ implies that the LR parser can reduce by the handle $A \rightarrow \beta_1$

Challenges with LR(0) Parsing

An LR(0) parser works only if each state with a reduce action has only one possible reduce action and no shift action

Ok

$\{L \rightarrow L, S\bullet\}$

Shift-Reduce Conflict

$\{L \rightarrow L, S\bullet,$
$L \rightarrow S\bullet, L\}$

Reduce-Reduce Conflict

$\{L \rightarrow S, L\bullet,$
$L \rightarrow S\bullet\}$

Takes shift/reduce decisions without any lookahead token

Lacks the power to parse programming language grammars

Canonical Collection of Sets of LR(0) Items

Consider the following grammar for adding numbers

Left associative	Right associative	Shift-Reduce Conflict
$S \rightarrow S + E \mid E$ $E \rightarrow \textbf{num}$	$S \rightarrow E + S \mid E$ $E \rightarrow \textbf{num}$	$\{S \rightarrow E \bullet +S,$ $S \rightarrow E\bullet\}$

$$\text{FIRST}\,(S) = \{\textbf{num}\}$$
$$\text{FIRST}\,(E) = \{\textbf{num}\}$$
$$\text{FOLLOW}\,(S) = \{\$\}$$
$$\text{FOLLOW}\,(E) = \{+, \$\}$$

$$\begin{aligned} I_0 &= \text{Closure}(\{S^{'} \rightarrow \bullet S\}) \\ &= \{S^{'} \rightarrow \bullet S, \\ &\quad S \rightarrow \bullet E + S, \\ &\quad S \rightarrow \bullet E, \\ &\quad E \rightarrow \bullet \textbf{num}\} \end{aligned}$$

$$\begin{aligned} I_1 &= \text{Goto}(I_0, S) \\ &= \{S^{'} \rightarrow S\bullet\} \end{aligned}$$

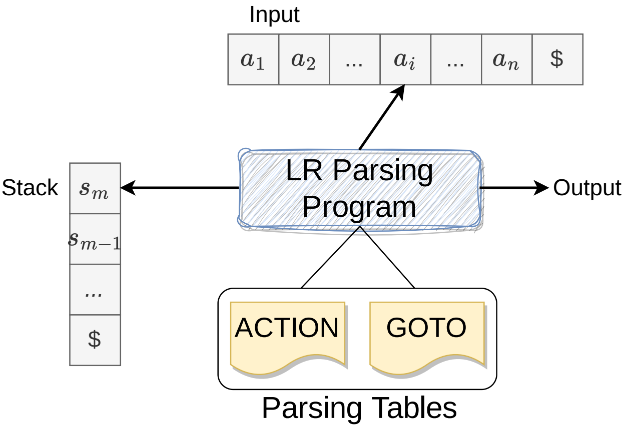
$$\begin{aligned} I_2 &= \text{Goto}(I_0, E) \\ &= \{S \rightarrow E \bullet +S, \\ &\quad S \rightarrow E\bullet\} \end{aligned}$$

$$\begin{aligned} I_3 &= \text{Goto}(I_0, \textbf{num}) \\ &= \{E \rightarrow \textbf{num}\bullet\} \end{aligned}$$

$$\begin{aligned} I_4 &= \text{Goto}(I_2, +) \\ &= \{S \rightarrow E + \bullet S\} \end{aligned}$$

Simple LR Parsing

Block Diagram of LR Parser



The LR parsing driver is the same for all LR parsers, only the parsing tables (i.e., ACTION and GOTO) change across parser types

SLR(1) Parsing

- Uses LR(0) items and LR(0) automaton, extends LR(0) parser to eliminate **a few** conflicts
  - For each reduction $A \longrightarrow \beta$, look at the next symbol c
  - Apply reduction only if $c \in \text{FOLLOW}(A)$

Constructing SLR Parsing Table

(i) Construct LR(0) canonical collection $C = \{I_0, I_1, \ldots, I_n\}$ for grammar $G^{'}$

(ii) State i is constructed from I_i

   (a) If $[A \rightarrow \alpha \bullet A\beta] \in I_i$ and $\text{GOTO}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ = "Shift j"
   (b) If $[A \rightarrow \alpha\bullet] \in I_i$, then set $\text{ACTION}[i, a]$ = "Reduce by $A \rightarrow \alpha$" for all a in $\text{FOLLOW}(A)$
   (c) If $[S^{'} \rightarrow S\bullet] \in I_i$, then set $\text{ACTION}[i, \$]$ = "Accept"

(iii) If $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}[i, A] = j$

(iv) All entries left undefined are "errors"

constraints on when reductions are applied

SLR Parsing for Expression Grammar

Rule #	Rule
1	$E \longrightarrow E + T$
2	$E \longrightarrow T$
3	$T \longrightarrow T * F$
4	$T \longrightarrow F$
5	$F \longrightarrow (E)$
6	$F \longrightarrow \textbf{id}$

$\text{FIRST}\,(E) = \{(, \textbf{id}\}$

$\text{FIRST}\,(T) = \{(, \textbf{id}\}$

$\text{FIRST}\,(F) = \{(, \textbf{id}\}$

$\text{FOLLOW}\,(E) = \{\$, +,)\}$

$\text{FOLLOW}\,(T) = \{\$, +, *,)\}$

$\text{FOLLOW}\,(F) = \{\$, +, *,)\}$

Canonical Collection of Sets of LR(0) Items

$$\begin{aligned} I_0 &= \text{Closure}(E^{'} \rightarrow \bullet E) \\ &= \{E^{'} \rightarrow \bullet E, \\ &\quad E \rightarrow \bullet E + T, \\ &\quad E \rightarrow \bullet T, \\ &\quad T \rightarrow \bullet T * F, \\ &\quad T \rightarrow \bullet F, \\ &\quad F \rightarrow \bullet (E), \\ &\quad F \rightarrow \bullet \mathbf{id}\} \end{aligned}$$

$$\begin{aligned} I_1 &= \text{Goto}(I_0, E) \\ &= \{E^{'} \rightarrow E \bullet, \\ &\quad E \rightarrow E \bullet + T\} \end{aligned}$$

$$\begin{aligned} I_2 &= \text{Goto}(I_0, T) \\ &= \{E \rightarrow T \bullet, \\ &\quad T \rightarrow T \bullet * F\} \end{aligned}$$

$$\begin{aligned} I_3 &= \text{Goto}(I_0, F) \\ &= \{T \rightarrow F \bullet\} \end{aligned}$$

$$\begin{aligned} I_4 &= \text{Goto}(I_0, \text{`('}) \\ &= \{F \rightarrow (\bullet E), \\ &\quad E \rightarrow \bullet E + T, \\ &\quad E \rightarrow \bullet T, \\ &\quad T \rightarrow \bullet T * F, \\ &\quad T \rightarrow \bullet F, \\ &\quad F \rightarrow \bullet (E), \\ &\quad F \rightarrow \bullet \mathbf{id}\} \end{aligned}$$

$$\begin{aligned} I_5 &= \text{Goto}(I_0, \mathbf{id}) \\ &= \{F \rightarrow \mathbf{id} \bullet\} \end{aligned}$$

$$\begin{aligned} I_6 &= \text{Goto}(I_1, +) \\ &= \{E \rightarrow E + \bullet T, \\ &\quad T \rightarrow \bullet T * F, \\ &\quad T \rightarrow \bullet F, \\ &\quad F \rightarrow \bullet (E), \\ &\quad F \rightarrow \bullet \mathbf{id}\} \end{aligned}$$

$$\begin{aligned} I_7 &= \text{Goto}(I_2, *) \\ &= \{T \rightarrow T * \bullet F, \\ &\quad F \rightarrow \bullet (E), \\ &\quad F \rightarrow \bullet \mathbf{id}\} \end{aligned}$$

$$\begin{aligned} I_8 &= \text{Goto}(I_4, E) \\ &= \{E \rightarrow E \bullet + T, \\ &\quad F \rightarrow (E \bullet)\} \end{aligned}$$

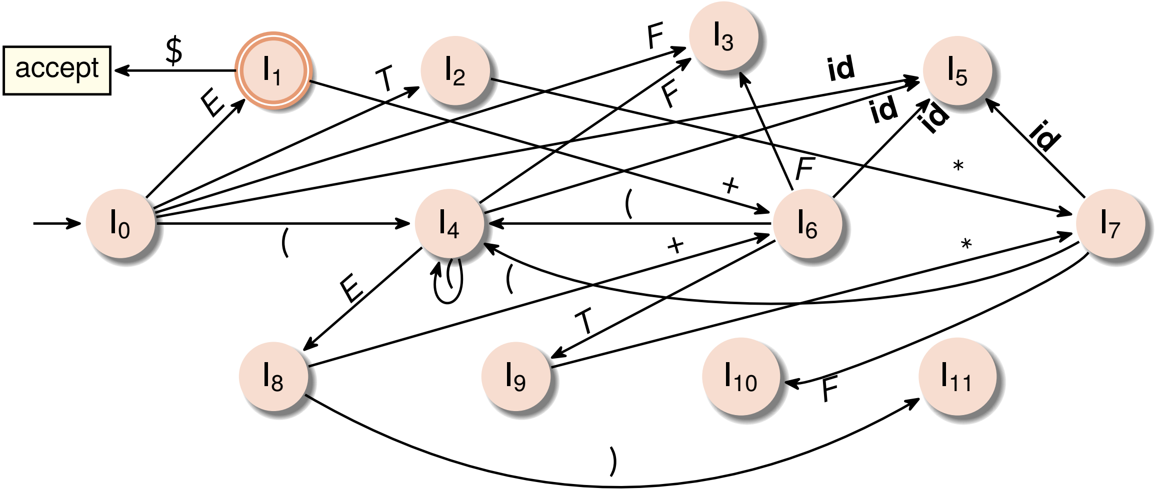
$$\begin{aligned} I_9 &= \text{Goto}(I_6, T) \\ &= \{E \rightarrow E + T \bullet, \\ &\quad T \rightarrow T \bullet * F\} \end{aligned}$$

$$\begin{aligned} I_{10} &= \text{Goto}(I_7, F) \\ &= \{T \rightarrow T * F \bullet\} \end{aligned}$$

$$\begin{aligned} I_{11} &= \text{Goto}(I_8, \text{`)'}) \\ &= \{F \rightarrow (E) \bullet\} \end{aligned}$$

$$\begin{aligned} I_2 &= \text{Goto}(I_4, T) \\ I_3 &= \text{Goto}(I_4, F) \\ I_4 &= \text{Goto}(I_4, \text{`('}) \\ I_5 &= \text{Goto}(I_4, \mathbf{id}) \\ I_3 &= \text{Goto}(I_6, F) \\ I_4 &= \text{Goto}(I_6, \text{`('}) \\ I_5 &= \text{Goto}(I_6, \mathbf{id}) \\ I_4 &= \text{Goto}(I_7, \text{`('}) \\ I_5 &= \text{Goto}(I_7, \mathbf{id}) \\ I_6 &= \text{Goto}(I_8, +) \\ I_7 &= \text{Goto}(I_9, *) \end{aligned}$$

LR(0) Automaton



SLR Parsing Table

State	ACTION						GOTO		
	id	+	$*$	(	)	\$	E	T	F
0	$s5$			$s4$			1	2	3
1		$s6$				Accept			
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		$r5$	$r5$		$r5$	$r5$			

Rule #	Rule
0	$E' \rightarrow E$
1	$E \rightarrow E + T$
2	$E \rightarrow T$
3	$T \rightarrow T * F$
4	$T \rightarrow F$
5	$F \rightarrow (E)$
6	$F \rightarrow \textbf{id}$

Moves of an LR Parser on $\mathbf{id} * \mathbf{id} + \mathbf{id}$

Stack	Input	Action
$\$0$	$\mathbf{id} * \mathbf{id} + \mathbf{id}\$$	Shift 5
$\$0\ \mathbf{id}\ 5$	$* \mathbf{id} + \mathbf{id}\$$	Reduce by $F \rightarrow \mathbf{id}$
$\$0\ F\ 3$	$* \mathbf{id} + \mathbf{id}\$$	Reduce by $T \rightarrow F$
$\$0\ T\ 2$	$* \mathbf{id} + \mathbf{id}\$$	Shift 7
$\$0\ T\ 2 * 7$	$\mathbf{id} + \mathbf{id}\$$	Shift 5
$\$0\ T\ 2 * 7\ \mathbf{id}\ 5$	$+ \mathbf{id}\$$	Reduce by $F \rightarrow \mathbf{id}$
$\$0\ T\ 2 * 7\ F\ 10$	$+ \mathbf{id}\$$	Reduce by $T \rightarrow T * F$
$\$0\ T\ 2$	$+ \mathbf{id}\$$	Reduce by $E \rightarrow T$
$\$0\ E\ 1$	$+ \mathbf{id}\$$	Shift 6
$\$0\ E\ 1 + 6$	$\mathbf{id}\$$	Shift 5
$\$0\ E\ 1 + 6\ \mathbf{id}\ 5$	$\$$	Reduce by $F \rightarrow \mathbf{id}$
$\$0\ E\ 1 + 6\ F\ 3$	$\$$	Reduce by $T \rightarrow F$
$\$0\ E\ 1 + 6\ T\ 9$	$\$$	Reduce by $E \rightarrow E + T$
$\$0\ E\ 1$	$\$$	Accept

Limitations of SLR Parsing

- If an SLR parse table for a grammar does not have multiple entries in any cell, then the grammar is unambiguous
- Every SLR(1) grammar is unambiguous, but there are unambiguous grammars that are not SLR(1)

Example to Highlight Limitations of SLR Parsing

Unambiguous grammar

$$S \rightarrow L = R \mid R$$
$$L \rightarrow * R \mid \textbf{id}$$
$$R \rightarrow L$$

FIRST (S) = $\{*, \textbf{id}\}$
FIRST (L) = $\{*, \textbf{id}\}$
FIRST (R) = $\{*, \textbf{id}\}$
FOLLOW (S) = $\{\$, =\}$
FOLLOW (L) = $\{\$, =\}$
FOLLOW (R) = $\{\$, =\}$

Example derivation

$$S \rightarrow L = R \rightarrow *R = R$$

Canonical LR(0) Collection

$$\begin{aligned} I_0 &= \text{Closure}(S^{'} \rightarrow \bullet S) \\ &= \{S^{'} \rightarrow \bullet S, \\ &\quad S \rightarrow \bullet L = R, \\ &\quad S \rightarrow \bullet R, \\ &\quad L \rightarrow \bullet * R, \\ &\quad L \rightarrow \bullet \textbf{id}, \\ &\quad R \rightarrow \bullet L\} \end{aligned}$$

$$\begin{aligned} I_1 &= \text{Goto}(I_0, S) \\ &= \{S^{'} \rightarrow S\bullet\} \end{aligned}$$

$$\begin{aligned} I_2 &= \text{Goto}(I_0, L) \\ &= \{S \rightarrow L\bullet = R, \\ &\quad R \rightarrow L\bullet\} \end{aligned}$$

$$\begin{aligned} I_3 &= \text{Goto}(I_0, R) \\ &= \{S \rightarrow R\bullet\} \end{aligned}$$

$$\begin{aligned} I_4 &= \text{Goto}(I_0, R) \\ &= \{L \rightarrow * \bullet R, \\ &\quad R \rightarrow \bullet L, \\ &\quad L \rightarrow \bullet * R, \\ &\quad L \rightarrow \bullet \textbf{id}\} \end{aligned}$$

$$\begin{aligned} I_5 &= \text{Goto}(I_0, \textbf{id}) \\ &= \{L \rightarrow \bullet \textbf{id}\} \end{aligned}$$

$$\begin{aligned} I_6 &= \text{Goto}(I_2, =) \\ &= \{S \rightarrow L = \bullet R, \\ &\quad R \rightarrow \bullet L, \\ &\quad L \rightarrow \bullet * R, \\ &\quad L \rightarrow \textbf{id}\} \end{aligned}$$

$$\begin{aligned} I_7 &= \text{Goto}(I_4, R) \\ &= \{L \rightarrow *R\bullet\} \end{aligned}$$

$$\begin{aligned} I_8 &= \text{Goto}(I_4, L) \\ &= \{R \rightarrow L\bullet\} \end{aligned}$$

$$\begin{aligned} I_9 &= \text{Goto}(I_6, R) \\ &= \{S \rightarrow L = R\bullet\} \end{aligned}$$

SLR Parsing Table

State	ACTION				GOTO		
	=	∗	**id**	$	*S*	*L*	*R*
0		*s*4	*s*5		1	2	3
1				Accept			
2	*s*6, *r*6			*r*6			
3							
4		*s*4	*s*5			8	7
5	*r*5			*r*5			
6		*s*4	*s*5			8	9
7	*r*4			*r*4			
8	*r*6			*r*6			
9				*r*2			

Shift-Reduce Conflict with SLR Parsing

$$\begin{aligned} I_0 &= \text{Closure}(S' \rightarrow \bullet S) \\ &= \{S' \rightarrow \bullet S, \\ &\quad S \rightarrow \bullet L = R, \\ &\quad S \rightarrow \bullet R, \\ &\quad L \rightarrow \bullet * R, \\ &\quad L \rightarrow \bullet \textbf{id}, \end{aligned}$$

$$\begin{aligned} I_2 &= \text{Goto}(I_0, L) \\ &= \{S \rightarrow L \bullet = R, \\ &\quad R \rightarrow L \bullet\} \end{aligned}$$

(i) $\text{ACTION}[2, =] = \text{Shift } 6$, or

(ii) $\text{ACTION}[2, =] = \text{Reduce } R \rightarrow L$ because $= \in \text{FOLLOW}(R)$

$$\begin{aligned} I_3 &= \text{Goto}(I_0, R) \\ &= \{S \rightarrow R \bullet\} \end{aligned}$$

$$\begin{aligned} I_4 &= \text{Goto}(I_0, R) \\ &= \{L \rightarrow * \bullet R, \\ &\quad R \rightarrow \bullet L, \end{aligned}$$

$$\begin{aligned} I_6 &= \text{Goto}(I_2, =) \\ &= \{S \rightarrow L = \bullet R, \\ &\quad R \rightarrow \bullet L, \\ &\quad L \rightarrow \bullet * R, \\ &\quad L \rightarrow \textbf{id}\} \end{aligned}$$

$$\begin{aligned} I_7 &= \text{Goto}(I_4, R) \\ &= \{L \rightarrow * R \bullet\} \end{aligned}$$

$$\begin{aligned} I_8 &= \text{Goto}(I_4, L) \\ &= \{R \rightarrow L \bullet\} \end{aligned}$$

$$\begin{aligned} I_9 &= \text{Goto}(I_6, R) \\ &= \{S \rightarrow L = R \bullet\} \end{aligned}$$

Moves of an SLR Parser on **id** = **id**

Stack	Input	Action
$\$0$	$\textbf{id} = \textbf{id}$	Shift 5
$\$0\textbf{id}5$	$= \textbf{id}$	Reduce by $L \rightarrow \textbf{id}$
$\$0L2$	$= \textbf{id}$	Reduce by $R \rightarrow L$
$\$0R3$	$= \textbf{id}$	Error

No right sentential form begins with $R = \dots$

Stack	Input	Action
$\$0$	$\textbf{id} = \textbf{id}\$$	Shift 5
$\$0\textbf{id}5$	$= \textbf{id}\$$	Reduce by $L \rightarrow \textbf{id}$
$\$0L2$	$= \textbf{id}\$$	Shift 6
$\$0L2 = 6$	$\textbf{id}\$$	Shift 5
$\$0L2 = 6\textbf{id}5$	$\$$	Reduce by $L \rightarrow \textbf{id}$
$\$0L2 = 6L8$	$\$$	Reduce by $R \rightarrow L$
$\$0L2 = 6R9$	$\$$	Reduce by $S \rightarrow L = R$
$\$0S1$	$\$$	Accept

Moves of an SLR Parser on **id** = **id**

Stack	Input	Action
$0	**id** = **id**	Shift 5
$0**id**5	= **id**	Reduce by $L \rightarrow$ **id**
0L$2	= **id**	Reduce by $R \rightarrow L$
0R$3	= **id**	Error

Stack	Input	Action
$0	**id** = **id**$	Shift 5
$0**id**5	= **id**$	Reduce by $L \rightarrow$ **id**
0L$2	= **id**$	Shift 6
0L$2 = 6	**id**$	Shift 5
		$\rightarrow$ **id**
		$\rightarrow L$
		$\rightarrow L = R$

State i calls for a reduction by $A \rightarrow \alpha$ if the set of items I_i contains items $[A \rightarrow \alpha\bullet]$ and $a \in \text{FOLLOW}(A)$

- Suppose βA is a viable prefix at the top of the stack
- There may be no right sentential form where a follows βA
  - An LR parser should not reduce by $A \rightarrow \alpha$ in such cases

Moves of an SLR Parser on **id** = **id**

Stack	Input	Action
\$0	**id** = **id**	Shift 5
\$0**id**5	= **id**	Reduce by $L \longrightarrow$ **id**
\$0L2	= **id**	Reduce by $R \longrightarrow L$
\$0R3	= **id**	Error

Stack	Input	Action
\$0	**id** = **id**\$	Shift 5
\$0**id**5	= **id**\$	Reduce by $L \longrightarrow$ **id**
\$0L2	= **id**\$	Shift 6
\$0L2 = 6	**id**\$	Shift 5
\$0L2 = 6**id**5	\$	Reduce by $L \longrightarrow$ **id**
		$\rightarrow L$
		$\rightarrow L = R$

SLR parser **cannot remember** the **left context**

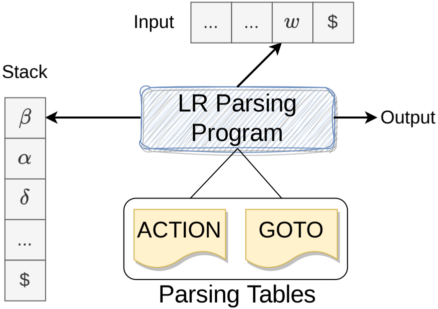
- SLR(1) states only tell us about the sequence on top of the stack, not what is below on the stack

Canonical LR Parsing

LR(1) Item

- An LR(1) item of a CFG G is a string of the form $[A \rightarrow \alpha \bullet \beta, a]$, with a as one symbol lookahead
  - $A \rightarrow \alpha\beta$ is a production in G, and $a \in T \cup \{\$\}$
- Suppose $[A \rightarrow \alpha \bullet \beta, a]$ where $\beta \neq \epsilon$, then the lookahead is not required
- If $[A \rightarrow \alpha\bullet, a]$, reduce **only** if the next input symbol is a
  - Set of possible terminals will always be a subset of A but can be a proper subset

- An LR(1) item $[A \rightarrow \alpha \bullet \beta, a]$ is valid for a viable prefix γ if there is a derivation $S \overset{*}{\underset{rm}{\Longrightarrow}} \delta A w \underset{rm}{\Longrightarrow} \delta\alpha\beta w$, where
  - (i) $\gamma = \delta\alpha$, and
  - (ii) a is the first symbol in w, or $w = \epsilon$ and $a = \$$



Computing Closure and Goto for LR(1) Collection

Closure(I)

```
repeat
  for each item $[A \rightarrow \alpha \bullet B\beta, a] \in I$
    for each production $B \rightarrow \gamma \in G'$
      for each terminal $b \in \text{FIRST}(\beta a)$
        add $[B \rightarrow \bullet\gamma, b]$ to set $I$
until no more items are added to $I$
return $I$
```

Goto(I, X)

```
$J = \phi$
for each item $[A \rightarrow \alpha \bullet X\beta, a] \in I$
  add item $[A \rightarrow \alpha X \bullet \beta, a]$ to set $J$
return Closure($J$)
```

Constructing LR(1) Sets of Items

```
$C = \text{Closure}(\{[S^{'} \rightarrow \bullet S, \$]\})$
repeat
for each set of items $I \in C$
  for each grammar symbol $X$
    if $\text{Goto}(I,X) \neq \phi$ and $\text{Goto}(I,X) \notin C$
      add $\text{Goto}(I,X)$ to $C$
until no new sets of items are added to $C$
```

Example Construction of LR(1) Items

Rule #	Rule
0	$S' \rightarrow S$
1	$S \rightarrow CC$
2	$C \rightarrow \mathbf{c}C$
3	$C \rightarrow \mathbf{d}$

generates the regular language $\mathbf{c}^*\mathbf{d}\mathbf{c}^*\mathbf{d}$

$$\begin{aligned} I_0 &= \text{Closure}(\{[S' \rightarrow \bullet S, \$]\}) \\ &= \{S' \rightarrow \bullet S, \$, \\ &\quad S \rightarrow \bullet CC, \$, \\ &\quad C \rightarrow \bullet \mathbf{c}C, \mathbf{c}/\mathbf{d}, \\ &\quad C \rightarrow \bullet \mathbf{d}, \mathbf{c}/\mathbf{d}\} \end{aligned}$$

$$\begin{aligned} I_1 &= \text{Goto}(I_0, S) \\ &= \{S' \rightarrow S\bullet, \$\} \end{aligned}$$

$$\begin{aligned} I_2 &= \text{Goto}(I_0, C) \\ &= \{S \rightarrow C \bullet C, \$, \\ &\quad C \rightarrow \bullet \mathbf{c}C, \$, \\ &\quad C \rightarrow \bullet \mathbf{d}, \$\} \end{aligned}$$

$$\begin{aligned} I_3 &= \text{Goto}(I_0, \mathbf{c}) \\ &= \{C \rightarrow \mathbf{c} \bullet C, \mathbf{c}/\mathbf{d}, \\ &\quad C \rightarrow \bullet \mathbf{c}C, \mathbf{c}/\mathbf{d}, \\ &\quad C \rightarrow \bullet \mathbf{d}, \mathbf{c}/\mathbf{d}\} \end{aligned}$$

$$\begin{aligned} I_4 &= \text{Goto}(I_0, \mathbf{d}) \\ &= \{C \rightarrow \mathbf{d}\bullet, \mathbf{c}/\mathbf{d}\} \end{aligned}$$

$$\begin{aligned} I_5 &= \text{Goto}(I_2, S) \\ &= \{S \rightarrow CC\bullet, \$\} \end{aligned}$$

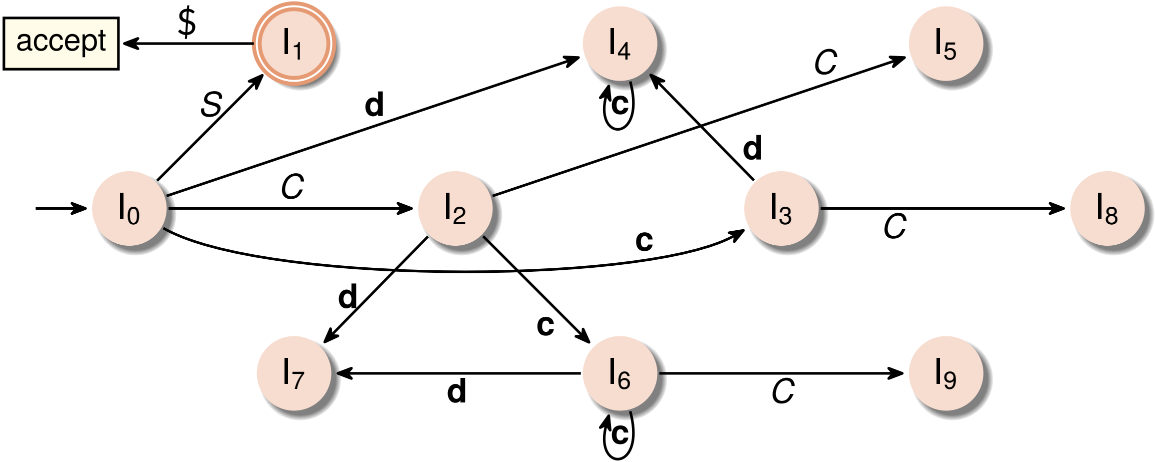
$$\begin{aligned} I_6 &= \text{Goto}(I_2, \mathbf{c}) \\ &= \{C \rightarrow \mathbf{c} \bullet C, \$, \\ &\quad C \rightarrow \bullet \mathbf{c}C, \$, \\ &\quad C \rightarrow \bullet \mathbf{d}, \$\} \end{aligned}$$

$$\begin{aligned} I_7 &= \text{Goto}(I_2, \mathbf{d}) \\ &= \{C \rightarrow \mathbf{d}\bullet, \$\} \end{aligned}$$

$$\begin{aligned} I_8 &= \text{Goto}(I_3, C) \\ &= \{C \rightarrow \mathbf{c}C\bullet, \mathbf{c}/\mathbf{d}\} \end{aligned}$$

$$\begin{aligned} I_9 &= \text{Goto}(I_6, C) \\ &= \{C \rightarrow \mathbf{c}C\bullet, \$\} \end{aligned}$$

LR(1) Automaton



Construction of Canonical LR(1) Parsing Tables

- Construct $C^{'} = \{I_0, I_1, \dots, I_n\}$
- State i of the parser is constructed from I_i
  - If $[A \rightarrow \alpha \bullet a\beta, b]$ is in I_i and $\text{Goto}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ = "Shift j"
  - If $[A \rightarrow \alpha\bullet, a]$ is in I_i and $A \neq S^{'}$, then set $\text{ACTION}[i, a]$ = "Reduce by $A \rightarrow \alpha\bullet$"
  - If $[S^{'} \rightarrow S\bullet, \$]$ is in I_i, then set $\text{ACTION}[i, \$]$ = "Accept"
- If $\text{Goto}(I_i, A) = I_j$, then $\text{GOTO}[i, A] = j$
- Initial state of the parser is constructed from the set of items containing $[S^{'} \rightarrow \bullet S, \$]$

Canonical LR(1) Parsing Table and Moves of a CLR Parser on **cdcd**

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

Stack	Input	Action
$0	**cdcd**$	Shift 3
$0**c**3	**dcd**$	Shift 3
$0**c**3**d**4	**cd**$	Reduce by $C \rightarrow$ **d**
$0**c**3$C$8	**cd**$	Reduce by $C \rightarrow$ **c**C
0C$2	**cd**$	Shift 6
0C$2**c**6	**d**$	Shift 7
0C$2**c**6**d**7	$	Reduce by $C \rightarrow$ **d**
0C$2**c**6$C$9	$	Reduce by $C \rightarrow$ **c**C
0C2C$5	$	Reduce by $S \rightarrow CC$
0S$1	$	Accept

Canonical LR(1) Parsing

- If the parsing table has no multiply-defined cells, then the corresponding grammar *G* is LR(1)
- Every SLR(1) grammar is an LR(1) grammar
  - Canonical LR parser may have more states than SLR

LALR Parsing

Example Construction of LR(1) Items

$$\begin{aligned} I_0 &= \text{Closure}(\{[S' \rightarrow \bullet S, \$]\}) \\ &= \{S' \rightarrow \bullet S, \$, \\ &\quad S \rightarrow \bullet CC, \$, \\ &\quad C \rightarrow \bullet \mathbf{c}C, \mathbf{c}/\mathbf{d}, \\ &\quad C \rightarrow \bullet \mathbf{d}, \mathbf{c}/\mathbf{d}\} \end{aligned}$$

$$\begin{aligned} I_1 &= \text{Goto}(I_0, S) \\ &= \{S' \rightarrow S\bullet, \$\} \end{aligned}$$

$$\begin{aligned} I_2 &= \text{Goto}(I_0, C) \\ &= \{S \rightarrow C \bullet C, \$, \\ &\quad C \rightarrow \bullet \mathbf{c}C, \$, \\ &\quad C \rightarrow \bullet \mathbf{d}, \$\} \end{aligned}$$

$$\begin{aligned} I_3 &= \text{Goto}(I_0, \mathbf{c}) \\ &= \{C \rightarrow \mathbf{c} \bullet C, \mathbf{c}/\mathbf{d}, \\ &\quad C \rightarrow \bullet \mathbf{c}C, \mathbf{c}/\mathbf{d}, \\ &\quad C \rightarrow \bullet \mathbf{d}, \mathbf{c}/\mathbf{d}\} \end{aligned}$$

$$\begin{aligned} I_4 &= \text{Goto}(I_0, \mathbf{d}) \\ &= \{C \rightarrow \mathbf{d}\bullet, \mathbf{c}/\mathbf{d}\} \end{aligned}$$

$$\begin{aligned} I_5 &= \text{Goto}(I_2, S) \\ &= \{S \rightarrow CC\bullet, \$\} \end{aligned}$$

$$\begin{aligned} I_6 &= \text{Goto}(I_2, \mathbf{c}) \\ &= \{C \rightarrow \mathbf{c} \bullet C, \$, \\ &\quad C \rightarrow \bullet \mathbf{c}C, \$, \\ &\quad C \rightarrow \bullet \mathbf{d}, \$\} \end{aligned}$$

$$\begin{aligned} I_7 &= \text{Goto}(I_2, \mathbf{d}) \\ &= \{C \rightarrow \mathbf{d}\bullet, \$\} \end{aligned}$$

$$\begin{aligned} I_8 &= \text{Goto}(I_3, C) \\ &= \{C \rightarrow \mathbf{c}C\bullet, \mathbf{c}/\mathbf{d}\} \end{aligned}$$

$$\begin{aligned} I_9 &= \text{Goto}(I_6, C) \\ &= \{C \rightarrow \mathbf{c}C\bullet, \$\} \end{aligned}$$

I_3 and I_6, I_4 and I_7, and I_8 and I_9 only differ in the second components

Lookahead LR (LALR) Parsing

- CLR(1) parser has numerous states
- Lookahead LR (LALR) parser **merges sets** of LR(1) items that have the **same core** (set of LR(0) items, i.e., first component)
  - LALR parsers have fewer states, the same as SLR
- LALR parser is used in many parser generators (e.g., Bison)

Construction of LALR Parsing Table

- Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of set of LR(1) items
- For each core present in LR(1) items, find all sets having the same core and replace these sets with their union
- Let $C^{'} = \{J_0, J_1, \ldots, J_n\}$ be the resulting sets of LR(1) items (also called LALR collection)
- Construct ACTION table as was done earlier, parsing actions for state i is constructed from J_i
- Let $J = I_1 \cup I_2 \cup \cdots \cup I_k$, where the cores of $I_1, I_2, \ldots, I_k$ are the same
  - Cores of $\text{Goto}(I_1, X)$, $\text{Goto}(I_2, X)$, …,$\text{Goto}(I_k, X)$ will also be the same
  - Let $K = \text{Goto}(I_1, X) \cup \text{Goto}(I_2, X) \cup \ldots \text{Goto}(I_k, X)$, then $K = \text{Goto}(J, X)$

LALR Grammar

If there are no parsing action conflicts, then the grammar is LALR(1)

Rule #	Rule
0	$S' \longrightarrow S$
1	$S \longrightarrow CC$
2	$C \longrightarrow \mathbf{c}C$
3	$C \longrightarrow \mathbf{d}$

$$\begin{aligned} I_{36} &= \text{Goto}(I_2, \mathbf{c}) \\ &= \{C \longrightarrow \mathbf{c} \bullet C, \mathbf{c}/\mathbf{d}/\$, \\ &\quad\; C \longrightarrow \bullet \mathbf{c}C, \mathbf{c}/\mathbf{d}/\$, \\ &\quad\; C \longrightarrow \bullet \mathbf{d}, \mathbf{c}/\mathbf{d}/\$\} \end{aligned}$$

$$\begin{aligned} I_{47} &= \text{Goto}(I_0, \mathbf{d}) \\ &= \{C \longrightarrow \mathbf{d}\bullet, \mathbf{c}/\mathbf{d}/\$\} \end{aligned}$$

$$\begin{aligned} I_{89} &= \text{Goto}(I_3, C) \\ &= \{C \longrightarrow \mathbf{c}C\bullet, \mathbf{c}/\mathbf{d}/\$\} \end{aligned}$$

LALR Parsing Table

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

Stack	Input	Action
$\$0$	$\mathbf{cdcd}\$$	Shift 36
$\$0\mathbf{c}36$	$\mathbf{dcd}\$$	Shift 47
$\$0\mathbf{c}36\mathbf{d}47$	$\mathbf{cd}\$$	Reduce by $C \rightarrow \mathbf{d}$
$\$0\mathbf{c}36C89$	$\mathbf{cd}\$$	Reduce by $C \rightarrow \mathbf{c}C$
$\$0C2$	$\mathbf{cd}\$$	Shift 36
$\$0C2\mathbf{c}36$	$\mathbf{d}\$$	Shift 47
$\$0C2\mathbf{c}36\mathbf{d}47$	$\$$	Reduce by $C \rightarrow \mathbf{d}$
$\$0C2\mathbf{c}36C89$	$\$$	Reduce by $C \rightarrow \mathbf{c}C$
$\$0C2C5$	$\$$	Reduce by $S \rightarrow CC$
$\$0S1$	$\$$	Accept

Notes on LALR Parsing

- LALR parser behaves like the CLR parser except for difference in stack states

Merging LR(1) items can never produce shift/reduce conflicts

- Suppose there is a shift-reduce conflict on lookahead a due to items $[B \rightarrow \beta \bullet a\gamma, b]$ and $[A \rightarrow \alpha \bullet, a]$
- But the merged state was formed from states with same cores, which implies $[B \rightarrow \beta \bullet a\gamma, c]$ and $[A \rightarrow \alpha \bullet, a]$ must have already been in the same state, for some value of c

Merging items may produce reduce/reduce conflicts

Reduce-Reduce Conflicts due to Merging

LR(1) grammar

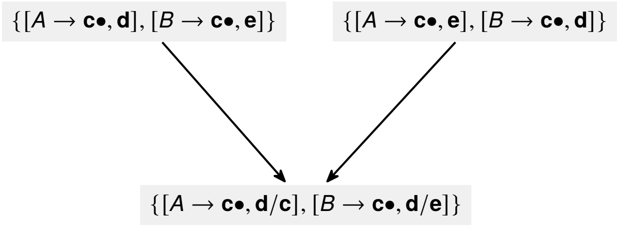
$S' \rightarrow S$

$S \rightarrow \mathbf{a}A\mathbf{d} \mid \mathbf{b}B\mathbf{d} \mid \mathbf{a}B\mathbf{e} \mid \mathbf{b}A\mathbf{e}$

$A \rightarrow \mathbf{c}$

$B \rightarrow \mathbf{c}$

Example strings: **acd**, **ace**, **bcd**, **bce**



$\{[A \rightarrow \mathbf{c}\bullet, \mathbf{d}], [B \rightarrow \mathbf{c}\bullet, \mathbf{e}]\}$

$\{[A \rightarrow \mathbf{c}\bullet, \mathbf{e}], [B \rightarrow \mathbf{c}\bullet, \mathbf{d}]\}$

$\{[A \rightarrow \mathbf{c}\bullet, \mathbf{d}/\mathbf{c}], [B \rightarrow \mathbf{c}\bullet, \mathbf{d}/\mathbf{e}]\}$

Dealing with Errors with LALR Parsing

CLR Parsing Table

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

LALR Parsing Table

State	ACTION			GOTO	
	c	**d**	$	S	C
0	$s36$	$s47$		1	2
1			Accept		
2	$s36$	$s47$			5
36	$s36$	$s47$			89
47	$r3$	$r3$	$r3$		
5			$r1$		
89	$r2$	$r2$	$r2$		

Rule #	Rule
0	$S' \rightarrow S$
1	$S \rightarrow CC$
2	$C \rightarrow \mathbf{c}C$
3	$C \rightarrow \mathbf{d}$

Comparing Moves of CLR and LALR Parsers

Consider an erroneous input **ccd**

CLR Parser

Stack	Input	Action
\$0	**ccd**\$	Shift 3
\$0**c**3	**d**\$	Shift 3
\$0**c**3**c**3	**d**\$	Shift 4
\$0**c**3**c**3**d**4	\$	Error

LALR Parser

Stack	Input	Action
\$0	**ccd**\$	Shift 36
\$0**c**36	**cd**\$	Shift 36
\$0**c**36**c**36	**d**\$	Shift 47
\$0**c**36**c**36**d**47	\$	Reduce by $C \rightarrow \mathbf{d}$
\$0**c**36**c**36C89	\$	Reduce by $C \rightarrow \mathbf{c}C$
\$0**c**36C89	\$	Reduce by $C \rightarrow \mathbf{c}C$
\$0C2	\$	Error

Comparing Moves of CLR and LALR Parsers

Consider an erroneous input **ccd**

CLR Parser

Stack
$0
$0**c**3
$0**c**3**c**
$0**c**3**c**

LALR Parser

		d
$0**c**36**c**36*C*89	$	Reduce by *C* → **c***C*
$0**c**36*C*89	$	Reduce by *C* → **c***C*
$0*C*2	$	Error

- CLR parser will not even reduce before reporting an error
- SLR and LALR parser may reduce several times before reporting an error, but will never shift an erroneous input symbol onto the stack

Using Ambiguous Grammars

Dealing with Ambiguous Grammars

LR(1) grammar

$$E' \to E$$
$$E \to E + E \mid E * E \mid (E) \mid \textbf{id}$$

Grammar does not distinguish between the associativity and precedence of the two operators

$$\begin{aligned} I_0 &= \text{Closure}(\{[E' \to \bullet E]\}) \\ &= \{E' \to \bullet E, \\ &\quad E \to \bullet E + E, \\ &\quad E \to \bullet E * E, \\ &\quad E \to \bullet (E), \\ &\quad E \to \bullet \textbf{id}\} \end{aligned}$$

$$\begin{aligned} I_1 &= \text{Goto}(I_0, E) \\ &= \{E' \to E \bullet, \\ &\quad E \to E \bullet +E, \\ &\quad E \to E \bullet *E\} \end{aligned}$$

$$\begin{aligned} I_2 &= \text{Goto}(I_0, `(\text{'}) \\ &= \{E \to (\bullet E), \\ &\quad E \to \bullet E + E, \\ &\quad E \to \bullet E * E, \\ &\quad E \to \bullet (E), \\ &\quad E \to \bullet \textbf{id}\} \end{aligned}$$

$$\begin{aligned} I_3 &= \text{Goto}(I_0, \textbf{id}) \\ &= \{E \to \textbf{id} \bullet\} \end{aligned}$$

$$\begin{aligned} I_4 &= \text{Goto}(I_0, +) \\ &= \{E \to E + \bullet E, \\ &\quad E \to \bullet E + E, \\ &\quad E \to \bullet E * E, \\ &\quad E \to \bullet (E), \\ &\quad E \to \bullet \textbf{id}\} \end{aligned}$$

$$\begin{aligned} I_9 &= \text{Goto}(I_6, `)\text{'}) \\ &= \{E \to (E) \bullet\} \end{aligned}$$

$$\begin{aligned} I_5 &= \text{Goto}(I_0, *) \\ &= \{E \to E * \bullet E, \\ &\quad E \to \bullet E + E, \\ &\quad E \to \bullet E * E, \\ &\quad E \to \bullet (E), \\ &\quad E \to \bullet \textbf{id}\} \end{aligned}$$

$$\begin{aligned} I_6 &= \text{Goto}(I_2, E) \\ &= \{E \to (E \bullet), \\ &\quad E \to E \bullet +E, \\ &\quad E \to E \bullet *E\} \end{aligned}$$

$$\begin{aligned} I_7 &= \text{Goto}(I_4, E) \\ &= \{E \to E + E \bullet, \\ &\quad E \to E \bullet +E, \\ &\quad E \to E \bullet *E\} \end{aligned}$$

$$\begin{aligned} I_8 &= \text{Goto}(I_5, E) \\ &= \{E \to E * E \bullet, \\ &\quad E \to E \bullet +E, \\ &\quad E \to E \bullet *E\} \end{aligned}$$

SLR Parsing Table

State	ACTION						GOTO
	id	+	$*$	(	)	\$	E
0	$s3$			$s2$			1
1		$s4$	$s5$			Accept	
2	$s3$			$s2$			
3		$r4$	$r4$		$r4$	$r4$	
4	$s3$			$s2$			7
5	$s3$			$s2$			8
6		$s4$	$s5$		$s9$		
7		$s4, r1$	$s5, r1$		$r1$	$r1$	
8		$s4, r2$	$s5, r2$		$r2$	$r2$	
9		$r3$	$r3$		$r3$	$r3$	

Moves of an SLR Parser on **id** + **id** $*$ **id**

Stack	Input	Action
\$0	**id** + **id** $*$ **id**\$	Shift 3
\$0**id**3	+ **id** $*$ **id**\$	Reduce by $E \longrightarrow$ **id**
\$0E1	+ **id** $*$ **id**\$	Shift 4
\$0E1+4	**id** $*$ **id**\$	Shift 3
\$0E1+4**id**3	$*$ **id**\$	Reduce by $E \longrightarrow$ **id** 3
\$0E1+4E7	$*$ **id**\$	

What can the parser do to resolve the ambiguity?

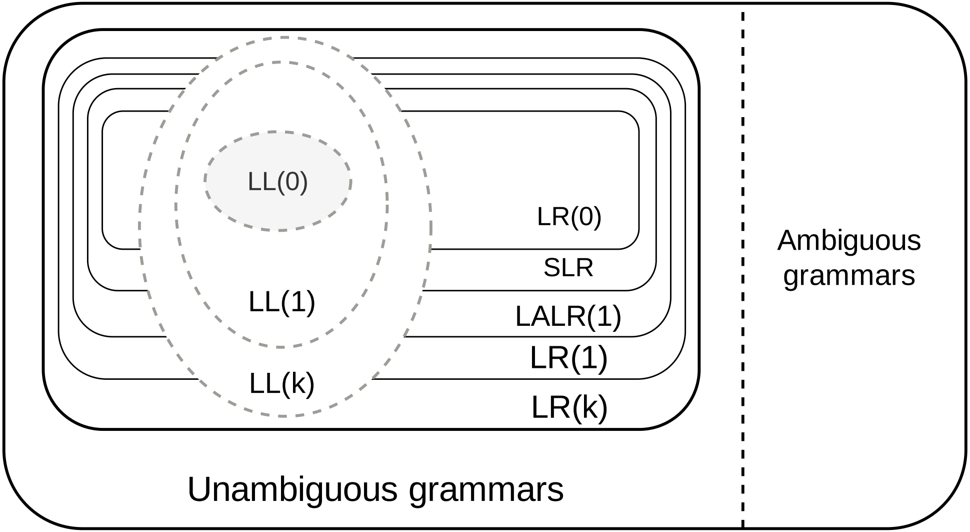
SLR Parsing Table

State	ACTION						GOTO
	id	+	*	(	)	$	*E*
0	*s*3			*s*2			1
1		*s*4	*s*5			Accept	
2	*s*3			*s*2			
3		*r*4	*r*4		*r*4	*r*4	
4	*s*3			*s*2			7
5	*s*3			*s*2			8
6		*s*4	*s*5		*s*9		
7		*s*4, ***r*1**	***s*5**, *r*1		*r*1	*r*1	
8		*s*4, ***r*2**	*s*5, ***r*2**		*r*2	*r*2	
9		*r*3	*r*3		*r*3	*r*3	

Why did the parser make these choices?

Comparison of Parsing Techniques

Relationship Among Grammars



Comparison of Parsing Techniques

- Ambiguous grammars are not LR
- Among grammars,
  - LL(0) ⊂ LL(1) ⊂ . . . ⊂ LL(k)[1]
  - LR(0) ⊂ SLR(1) ⊂ LALR(1) ⊂ LR(1)
    - SLR(1) = LR(0) items + FOLLOW
    - SLR(1) parsers can parse a larger number of grammars than LR(0)
    - Any grammar that can be parsed by an LR(0) parser can be parsed by an SLR(1) parser
  - SLR(k) ⊂ LALR(k) ⊂ LR(k)
  - LL(k) ⊂ LR(k)
    - Bottom-up parsing is a more powerful technique compared to top-down parsing
    - LR grammars can handle left recursion
    - Detects errors as soon as possible, and allows for better error recovery
    - Automated parser generators such as Yacc and Bison implement LALR parsing

---

[1] D. Rosenkrantz and R. Stearns. Properties of Deterministic Top-Down Grammars.

References

- A. Aho et al. Compilers: Principles, Techniques, and Tools. Sections 4.5–4.8, 2nd edition, Pearson Education.
- K. Cooper and L. Torczon. Engineering a Compiler. Sections 3.4–3.6, 2nd edition, Morgan Kaufmann.