CS 335: Bottom-up Parsing

Swarnendu Biswas

Semester 2022-2023-II

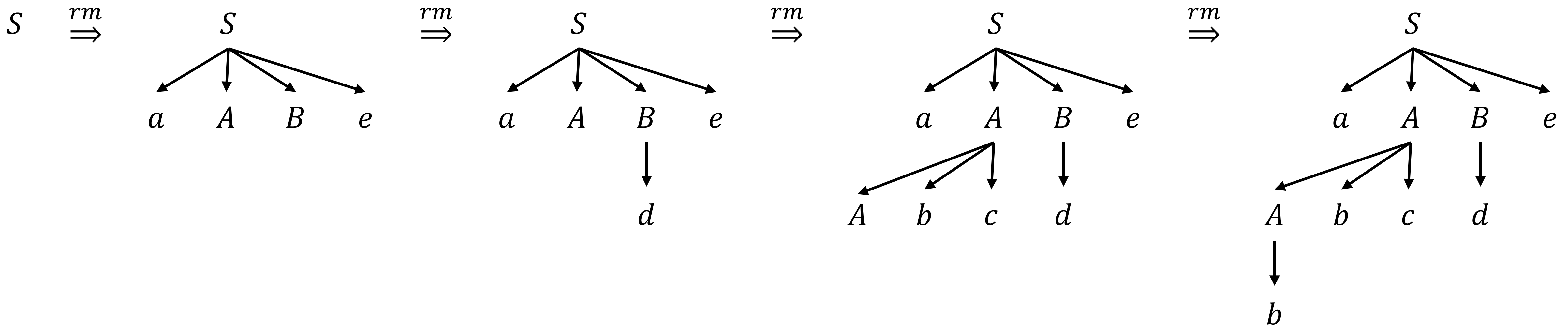
CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

Rightmost Derivation of $abbcde$

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

Input string: $abbcde$
$S \rightarrow aABe$
$\rightarrow aAde$
$\rightarrow aAbcde$
$\rightarrow abbcde$



Bottom-up Parsing

Constructs the parse tree starting from the leaves and working up toward the root

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

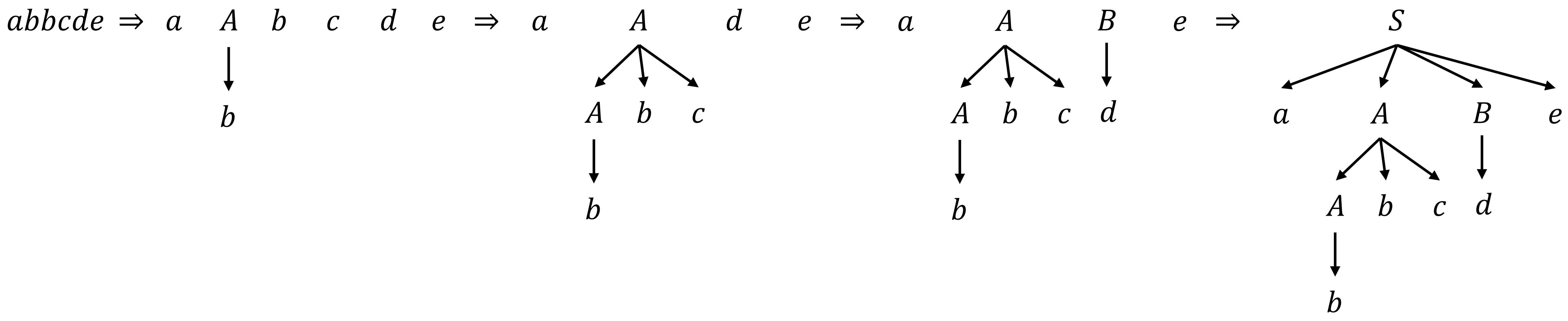
Input string: $abbcde$	
$S \rightarrow aABe$	$abbcde$
$\rightarrow aAde$	$\rightarrow aAbcde$
$\rightarrow aAbcde$	$\rightarrow aAde$
$\rightarrow abbcde$	$\rightarrow aABe$
	$\rightarrow S$

reverse of rightmost derivation

Bottom-up Parsing

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

Input string: $abbcde$
$abbcde$
$\to aAbcde$
$\to aAde$
$\to aABe$
$\to S$



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

$$S \underset{rm}{\Rightarrow} \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \dots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

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

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

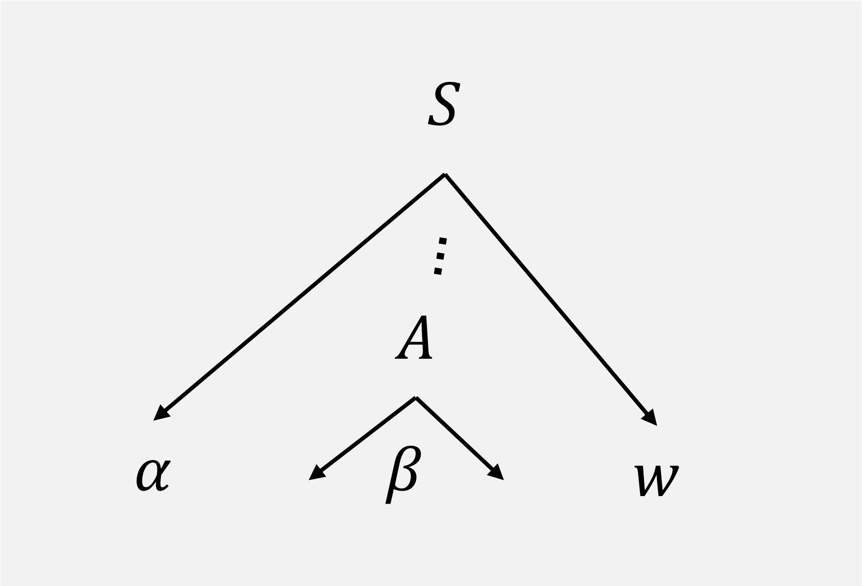
Right Sentential Form	Handle	Reducing Production
$\textbf{id}_1 * \textbf{id}_2$	$\textbf{id}_1$	$F \rightarrow \textbf{id}$
$F * \textbf{id}_2$	F	$T \rightarrow F$
$T * \textbf{id}_2$	$\textbf{id}_2$	$F \rightarrow \textbf{id}$
$T * F$	$T * F$	$T \rightarrow T * F$
T	T	$E \rightarrow T$

Although T is the body of the production $E \rightarrow T$, 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 $S \underset{rm}{\Rightarrow}^* \alpha A w \underset{rm}{\Rightarrow} \alpha\beta w$, 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 $\alpha\beta w$

Handle

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 (i.e., being parsed) consists of two parts
  - Left part is a string of terminals and nonterminals, and is stored in stack
  - Right part is a string of terminals read from an input buffer
  - Bottom of the stack and end of input are represented by \$
- Type of bottom-up parsing with two primary actions, shift and reduce
  - Other obvious actions are accept and error
- **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

Shift-Reduce Parsing

- **Initial**

Stack	Input
$\$$	$w\$$

Shift $\Downarrow *$ Reduce

- **Final goal**

Stack	Input
$\$S$	$\$$

Shift-Reduce Parsing

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

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

Or report an error in case of a syntax error

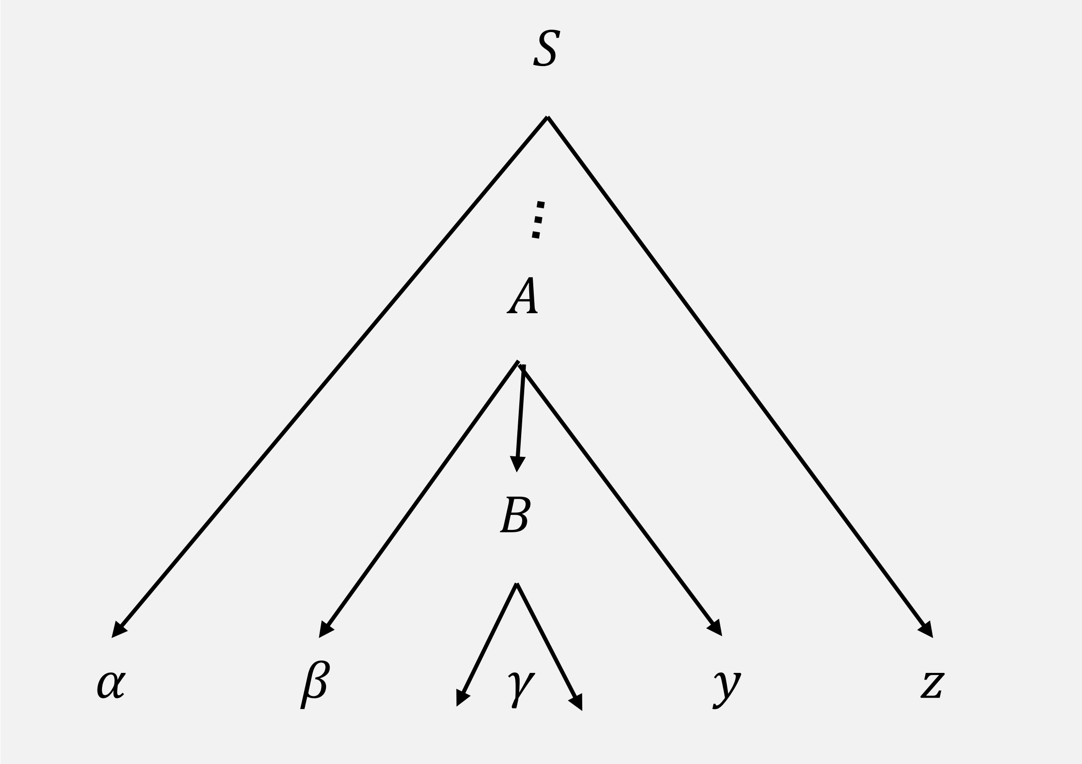
Handle on Top of the Stack

- Is the following scenario possible?

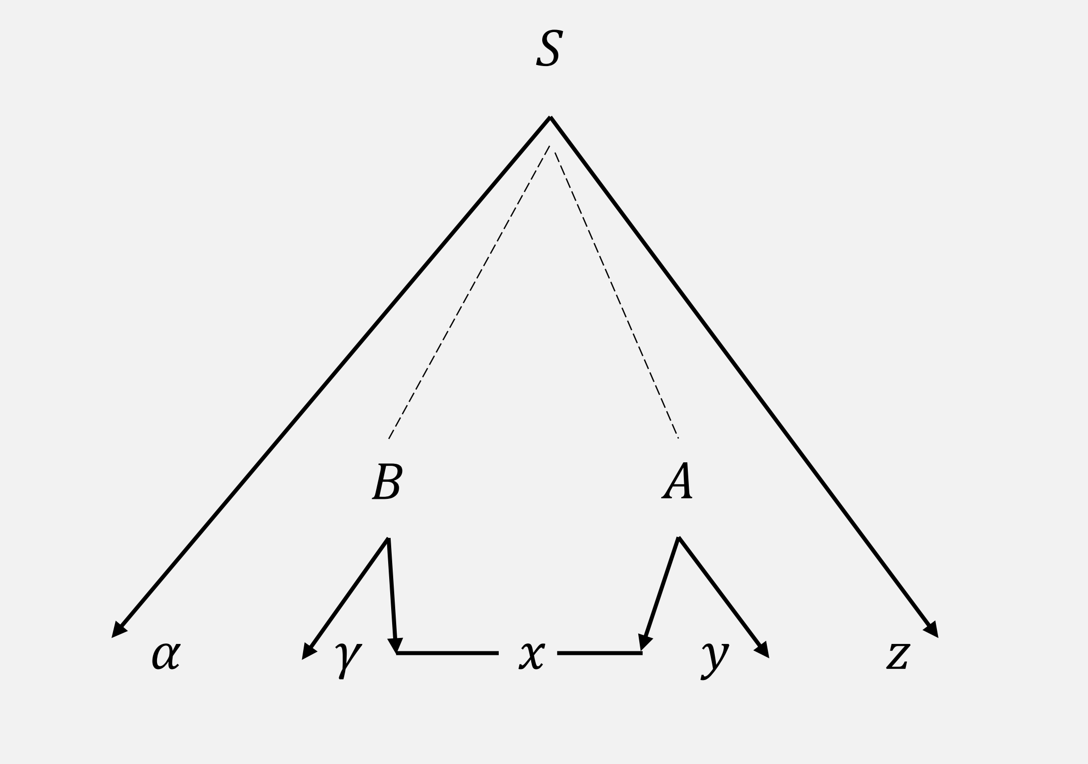
Stack	Input	Action
…		
$\$ \alpha\beta\gamma$	$w\$$	Reduce by $A \rightarrow \gamma$
$\$ \alpha\beta A$	$w\$$	Reduce by $B \rightarrow \beta$
$\$\alpha B A$	$w\$$	
…		

Possible Choices in Rightmost Derivation

$$1.\ S \underset{rm}{\Rightarrow} \alpha A z \underset{rm}{\Rightarrow} \alpha\beta B y z \underset{rm}{\Rightarrow} \alpha\beta\gamma y z$$



$$2.\ S \underset{rm}{\Rightarrow} \alpha B x A z \underset{rm}{\Rightarrow} \alpha B x y z \underset{rm}{\Rightarrow} \alpha\gamma x y z$$



Handle on Top of the Stack

- Is the following scenario possible?

Stack	Input	Action
…		

Handle always eventually appears 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 **detecting 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 there is a choice?	• Both shift and reduce are valid, implies a **shift-reduce conflict**
Which rule to use if reduction is possible by more than one rule?	• **Reduce-reduce conflict**

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

$\mathbf{id} + \mathbf{id} * \mathbf{id}$		
Stack	**Input**	**Action**
$\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	Shift
…		
$\$E + E$	$* \mathbf{id}\$$	**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 $E \rightarrow E + E$
$\$E$	$\$$	

Shift-Reduce Conflict

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

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

What is a correct thing to do for this grammar – shift or reduce? E.g., 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 $R \rightarrow R + R$
$\$M$	$\$$	

$c + c$

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

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

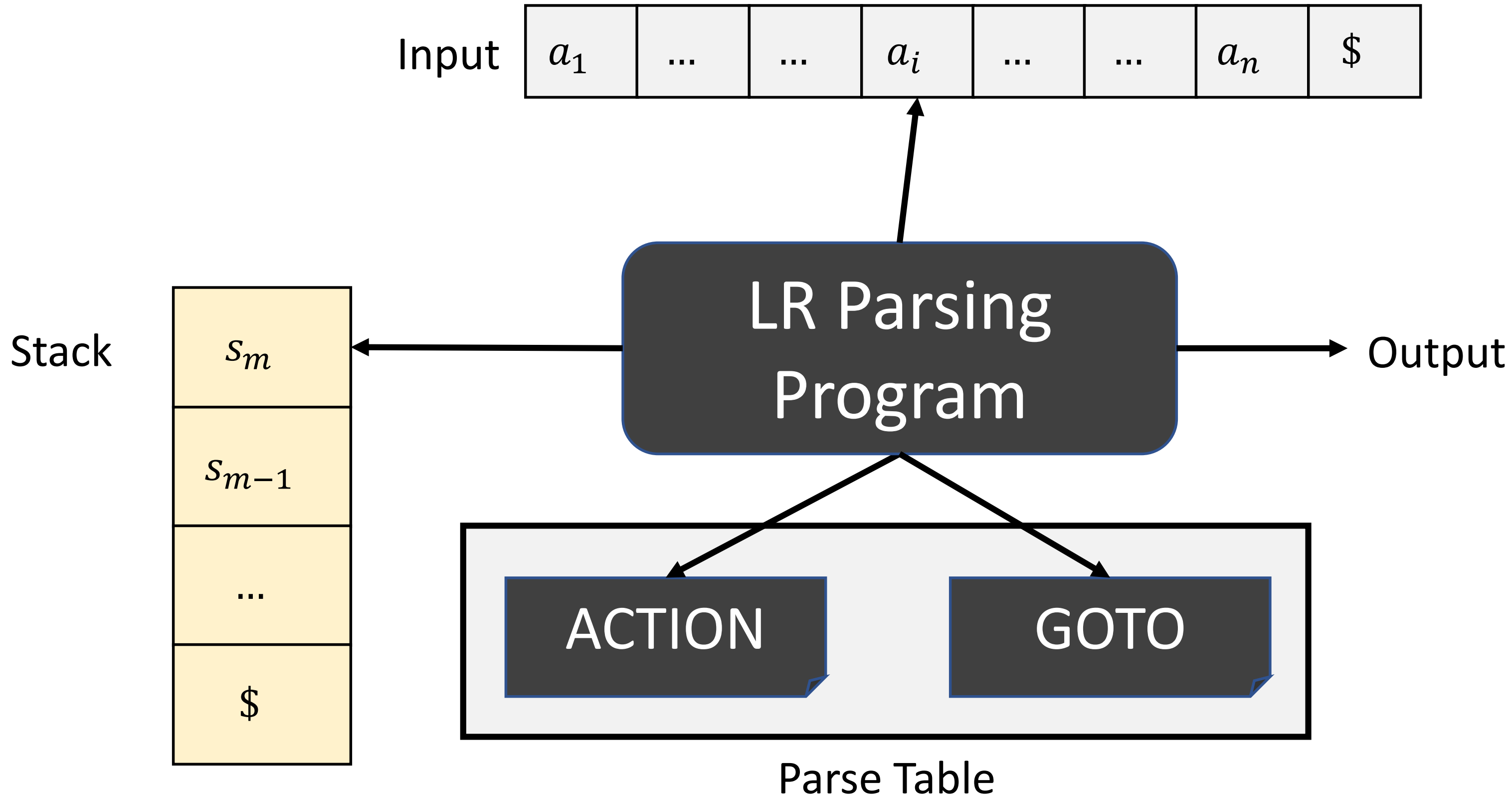
Can recognize almost all language constructs with CFGs

Most general nonbacktracking shift-reduce parsing method

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 table (including ACTION and GOTO) changes across parser types

LR Parsing

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

LR(0) Item

- An LR(0) item (also called 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

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

- $A \rightarrow \bullet XYZ$ 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 $A \rightarrow \bullet$

Closure Operation

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

$$E' \rightarrow E$$
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \mathbf{id}$$

Suppose $I = \{E' \rightarrow \bullet E\ \}$

Closure(I) = {

$E' \rightarrow \bullet E$,

$E \rightarrow \bullet E + T$,

$E \rightarrow \bullet T$,

$T \rightarrow \bullet T * F$,

$T \rightarrow \bullet F$,

$F \rightarrow \bullet(E)$,

$F \rightarrow \bullet$**id**

}

Kernel and Nonkernel Items

- If one B-production is added to Closure(I) with the dot at the left end, then all B-productions will be added to the closure
- Kernel items
  - Initial item $S' \rightarrow \bullet S$, and all items whose dots are not at the left end
- Nonkernel items
  - All items with their dots at the left end, except for $S' \rightarrow \bullet S$

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 set of valid items for prefix αX

- Intuitively, Goto(I, X) defines the transitions in the LR(0) automaton
  - Goto(I, X) gives the transition from state I under input X

Example of Goto

$$E' \rightarrow E$$
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \mathbf{id}$$

Suppose $I = \{$

$E' \rightarrow E\bullet,$

$E \rightarrow E \bullet + T$

$\}$

Goto(I, +) = {

$E \rightarrow E + \bullet T,$

$T \rightarrow \bullet T * F,$

$T \rightarrow \bullet F,$

$F \rightarrow \bullet(E),$

$F \rightarrow \bullet\mathbf{id}$

}

Canonical Collection of Sets of LR(0) Items

C = Closure($\{S' \rightarrow \bullet S\}$)
repeat
    for each set of items I in C
        for each grammar symbol X
            if Goto(I, X) is not empty and not in C
                add Goto(I, X) to C
until no new sets of items are added to C

Canonical Collection of Sets of LR(0) Items

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

- Compute the canonical collection for the expression grammar

Canonical Collection of Sets of LR(0) Items

$I_0 =$ Closure($\{E' \rightarrow \bullet E\}$)$= \{$
$E' \rightarrow \bullet E,$
$E \rightarrow \bullet E + T,$
$E \rightarrow \bullet T,$
$T \rightarrow \bullet T * F,$
$T \rightarrow \bullet F,$
$F \rightarrow \bullet (E),$
$F \rightarrow \bullet \mathbf{id},$
$\}$

$I_1 =$ Goto(I_0, E) $= \{$
$E' \rightarrow E\bullet,$
$E \rightarrow E\bullet + T$
$\}$

$I_2 =$ Goto(I_0, T)$= \{$
$E \rightarrow T\bullet,$
$T \rightarrow T\bullet * F$
$\}$

$I_3 =$ Goto(I_0, F)$= \{$
$T \rightarrow F\bullet$
$\}$

$I_5 =$ Goto($I_0, \mathbf{id}$)$= \{$
$F \rightarrow \mathbf{id}\bullet$
$\}$

$I_4 =$ Goto(I_0, "(")$= \{$
$F \rightarrow (\bullet E),$
$E \rightarrow \bullet E + T,$
$E \rightarrow \bullet T,$
$T \rightarrow \bullet T * F,$
$T \rightarrow \bullet F,$
$F \rightarrow \bullet (E),$
$F \rightarrow \bullet \mathbf{id},$
$\}$

$I_7 =$ Goto($I_2, *$) $= \{$
$T \rightarrow T * \bullet F,$
$F \rightarrow \bullet (E),$
$F \rightarrow \bullet \mathbf{id}$
$\}$

Canonical Collection of Sets of LR(0) Items

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

$I_8 = \text{Goto}(I_4, E) = \{$
$E \rightarrow E \bullet + T,$
$F \rightarrow (E \bullet)$
$\}$

$I_9 = \text{Goto}(I_6, T) = \{$
$E \rightarrow E + T \bullet,$
$T \rightarrow T \bullet * F$
$\}$

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

$I_{11} = \text{Goto}(I_8, ")") = \{$
$F \rightarrow (E) \bullet$
$\}$

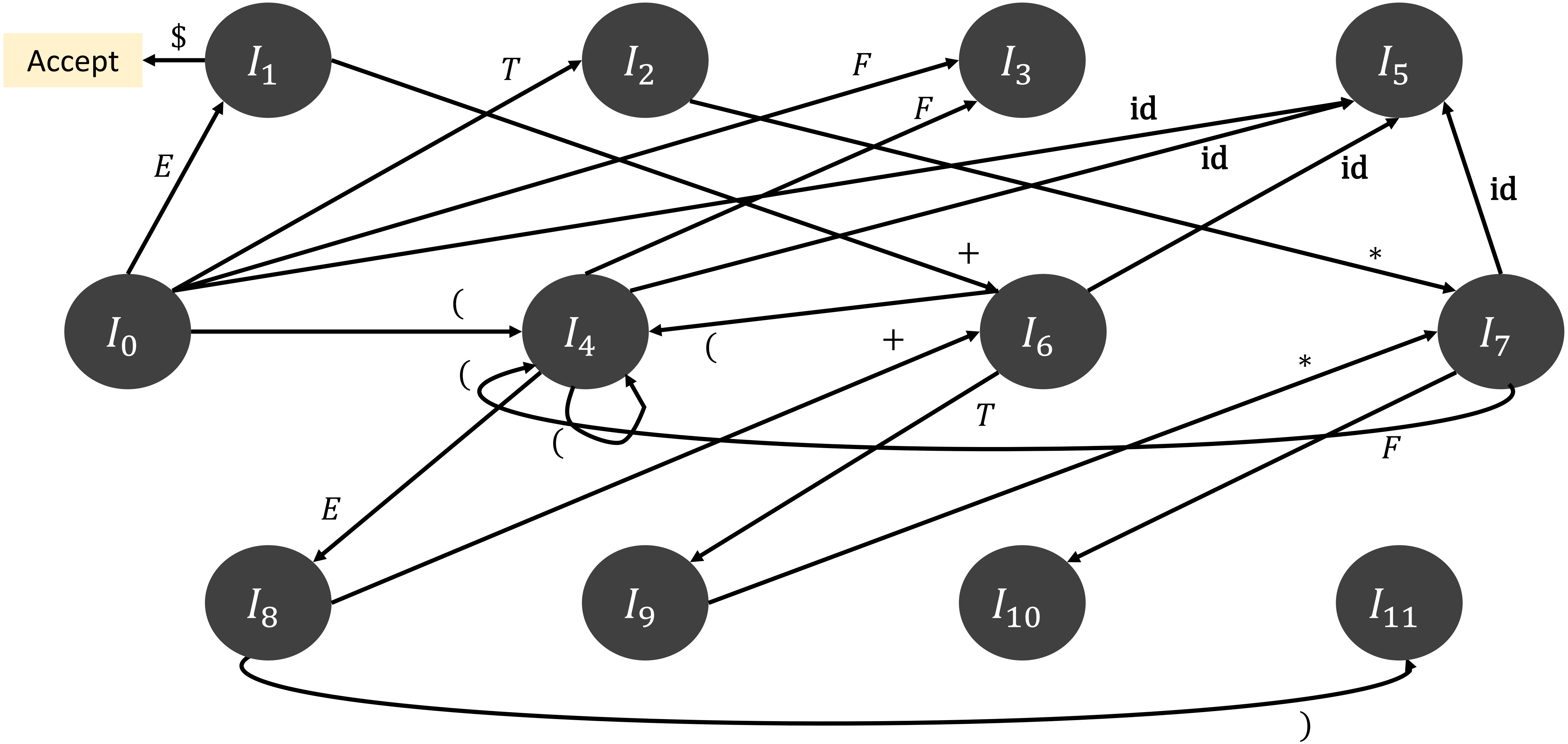
$I_2 = \text{Goto}(I_4, T)$
$I_3 = \text{Goto}(I_4, F)$
$I_4 = \text{Goto}(I_4, "(")$
$I_5 = \text{Goto}(I_4, \textbf{id})$
$I_3 = \text{Goto}(I_6, F)$
$I_4 = \text{Goto}(I_6, "(")$
$I_5 = \text{Goto}(I_6, \textbf{id})$
$I_4 = \text{Goto}(I_7, "(")$
$I_5 = \text{Goto}(I_7, \textbf{id})$
$I_6 = \text{Goto}(I_8, +)$
$I_7 = \text{Goto}(I_9, *)$

LR(0) Automaton

- An LR parser makes shift-reduce decisions by maintaining states
- Canonical LR(0) collection is used for constructing a DFA for parsing
- States represent sets of LR(0) items in the canonical LR(0) collection
  - Start state is 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

LR(0) Automaton

Each state is associated with a unique grammar symbol



Use of LR(0) Automaton

- How can LR(0) automata 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

1) Construct LR(0) canonical collection $C = \{I_0, I_1, \dots, I_n\}$ for grammar G'
2) State i is constructed from I_i
   a) If $[A \rightarrow \alpha \bullet a\beta]$ is in I_i and Goto(I_i, a) = I_j, then set ACTION[i, a] = "Shift j"
   b) If $[A \rightarrow \alpha \bullet]$ is in I_i, then set Action [i, a] = "Reduce $A \rightarrow \alpha$" for all a
   c) If $[S' \rightarrow S \bullet]$ is in I_i, then set Action [i, \$] = "Accept"
3) If Goto(I_i, A) = I_j, then GOTO[i, A] = j
4) All entries left undefined are "errors"

LR(0) Parsing Table

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

Shift-Reduce Parser with LR(0) Automaton

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

Popped 5, pushed 3 since $I_3 = \text{Goto}(I_0, F)$

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

Viable Prefix

- Consider $$E \rightarrow T \rightarrow T * F \rightarrow T * \mathbf{id} \rightarrow F * \mathbf{id} \rightarrow \mathbf{id} * \mathbf{id}$$
- $\mathbf{id} *$ is a prefix of a right sentential form, but it can never appear on the stack
  - Always reduce by $F \rightarrow \mathbf{id}$ before shifting $*$ (see previous slide)
- Not all prefixes of a right sentential form can appear on the stack
- A viable prefix is a prefix of a right sentential form that can appear on the stack of a shift-reduce parser
  - 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

Example of a Viable Prefix

$S \rightarrow X_1X_2X_3X_4$
$A \rightarrow X_1X_2$

Let $w = X_1X_2X_3$

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 $A \rightarrow \beta_1\beta_2$, and $\alpha\beta_1$is on the stack.

- $\beta_2 \neq \epsilon$ implies the handle $\beta_1\beta_2$ is not at the top of the stack yet, so **shift**
- $\beta_2 = \epsilon$ implies then the 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

$\{ L \rightarrow L, S\bullet \}$	$\{ L \rightarrow L, S\bullet$, $S \rightarrow S\bullet, L \}$	$\{ L \rightarrow S, L\bullet$, $L \rightarrow S\bullet \}$
Ok	Shift-reduce conflict	Reduce-reduce conflict

- Takes shift/reduce decisions **without any lookahead token**
  - Lacks the power to parse programming language grammars

Challenges with LR(0) Parsing

- Consider the following grammar for adding numbers

$$S \to S + E \mid E$$
$$E \to \textbf{num}$$

Left associative

$$S \to E + S \mid E$$
$$E \to \textbf{num}$$

Right associative

Not LR(0)

$$S \to E \bullet + S$$
$$S \to E \bullet$$

Shift-reduce conflict

Canonical Collection of Sets of LR(0) Items

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

$I_0 = \text{Closure}(\{S' \rightarrow \bullet S\}) = \{$
$S' \rightarrow \bullet S,$
$S \rightarrow \bullet E + S,$
$S \rightarrow \bullet E,$
$E \rightarrow \bullet\mathbf{num}$
$\}$

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

Not LR(0)

$I_3 = \text{Goto}(I_0, \mathbf{num}) = \{$
$E \rightarrow \mathbf{num}\bullet$
$\}$

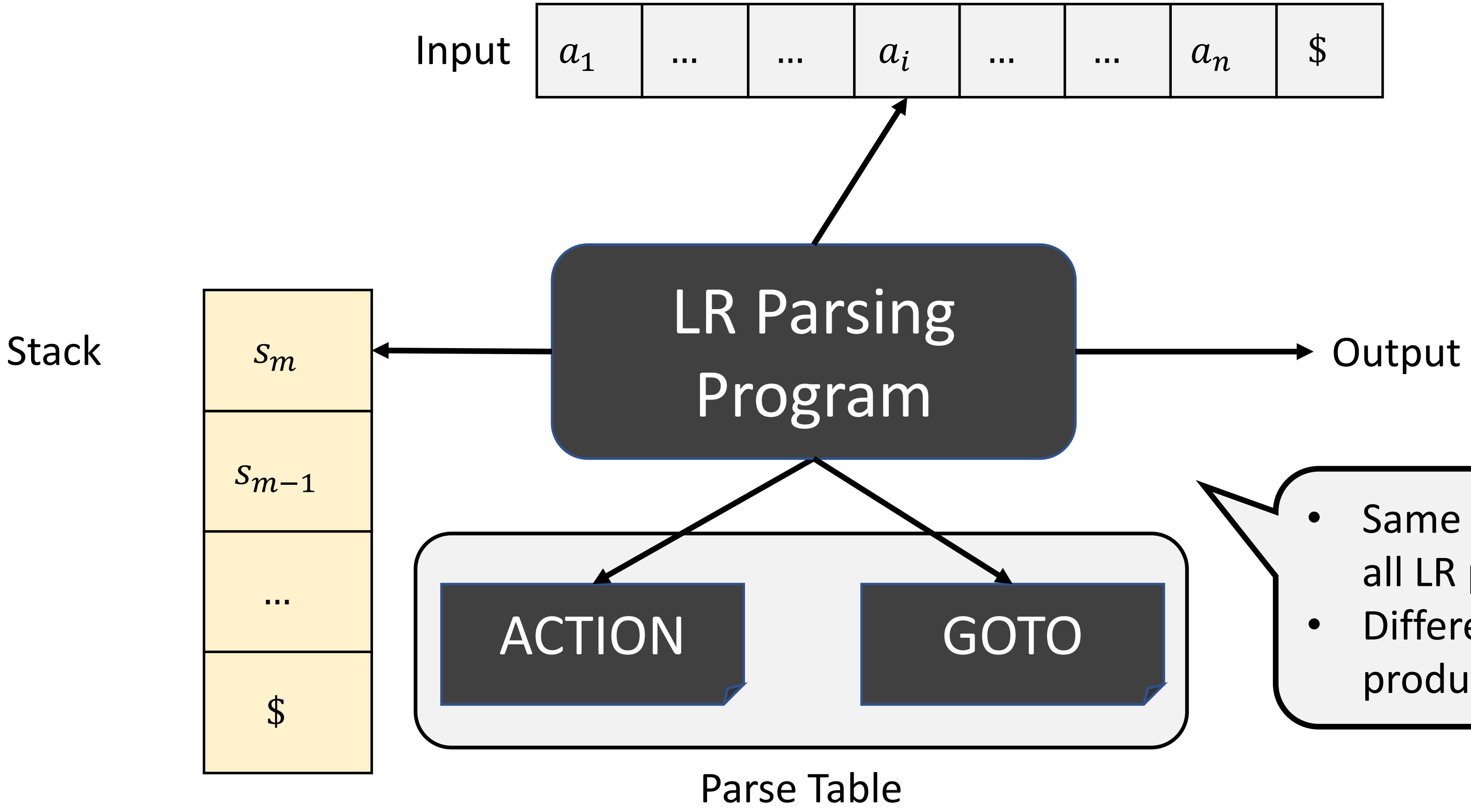
$I_4 = \text{Goto}(I_2, +) = \{$
$S \rightarrow E + \bullet S$
$\}$

$I_2 = \text{Goto}(I_0, E) = \{$
$\boldsymbol{S \rightarrow E \bullet + S,}$
$\boldsymbol{S \rightarrow E\bullet}$
$\}$

Simple LR Parsing

SLR(1)

Block Diagram of LR Parser



- Same driver program is used for all LR parsers
- Different LR parsing techniques produce different parse tables

LR Parsing Algorithm

- The parser driver is same for all LR parsers
  - Only the parsing table changes across parsers
- A shift-reduce parser shifts a symbol, and an LR parser shifts a state

- 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

SLR(1) Parsing

- Uses LR(0) items and LR(0) automaton, **extends LR(0) parser to eliminate a few conflicts**
  - For each reduction $A \rightarrow \beta$, look at the next symbol c
  - Apply reduction **only if** $c \in \mathrm{FOLLOW}(A)$ or $c = \epsilon$ and $S \overset{*}{\Rightarrow} \gamma A$

Constructing SLR Parsing Table

1) Construct LR(0) canonical collection $C = \{I_0, I_1, \dots, I_n\}$ for grammar G'
2) State i is constructed from I_i
   a) If $[A \rightarrow \alpha \bullet a \beta]$ is in I_i and Goto(I_i, a) = I_j, then set ACTION[i, a] = “Shift j”
   b) If $[A \rightarrow \alpha \bullet]$ is in I_i, then set ACTION[i, a] = “Reduce $A \rightarrow \alpha$” **for all $\boldsymbol{a}$ in $\mathbf{FOLLOW}(\boldsymbol{A})$**
   c) If $[S' \rightarrow S \bullet]$ is in I_i, then set Action [i, \$] = “Accept”
3) If Goto(I_i, A) = I_j, then GOTO[i, A] = j
4) All entries left undefined are “errors”

Constraints on when reductions are applied

SLR Parsing for Expression Grammar

Rule #	Rule
1	$E \rightarrow E + T$
2	$E \rightarrow T$
3	$T \rightarrow T * F$
4	$T \rightarrow F$
5	$F \rightarrow (E)$
6	$F \rightarrow$ id

- sj means shift and stack state i
- rj means reduce by rule $\#j$
- acc means accept
- blank means error

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

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

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

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

Canonical Collection of Sets of LR(0) Items

I_0 = Closure($\{E' \rightarrow \bullet E\}$)= {
$E' \rightarrow \bullet E,$
$E \rightarrow \bullet E + T,$
$E \rightarrow \bullet T,$
$T \rightarrow \bullet T * F,$
$T \rightarrow \bullet F,$
$F \rightarrow \bullet (E),$
$F \rightarrow \bullet \mathbf{id},$
}

I_1 = Goto(I_0, E) = {
$E' \rightarrow E\bullet,$
$E \rightarrow E\bullet + T$
}

I_2 = Goto(I_0, T)= {
$E \rightarrow T\bullet,$
$T \rightarrow T\bullet * F$
}

I_3 = Goto(I_0, F)= {
$T \rightarrow F\bullet$
}

I_5 = Goto(I_0, **id**)= {
$F \rightarrow \mathbf{id}\bullet$
}

I_4 = Goto(I_0, "(")= {
$F \rightarrow (\bullet E),$
$E \rightarrow \bullet E + T,$
$E \rightarrow \bullet T,$
$T \rightarrow \bullet T * F,$
$T \rightarrow \bullet F,$
$F \rightarrow \bullet (E),$
$F \rightarrow \bullet \mathbf{id},$
}

I_7 = Goto(I_2,$*$) = {
$T \rightarrow T * \bullet F,$
$F \rightarrow \bullet (E),$
$F \rightarrow \bullet \mathbf{id}$
}

Canonical Collection of Sets of LR(0) Items

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

$I_8 = \text{Goto}(I_4, E) = \{$
$E \rightarrow E \bullet + T,$
$F \rightarrow (E \bullet)$
$\}$

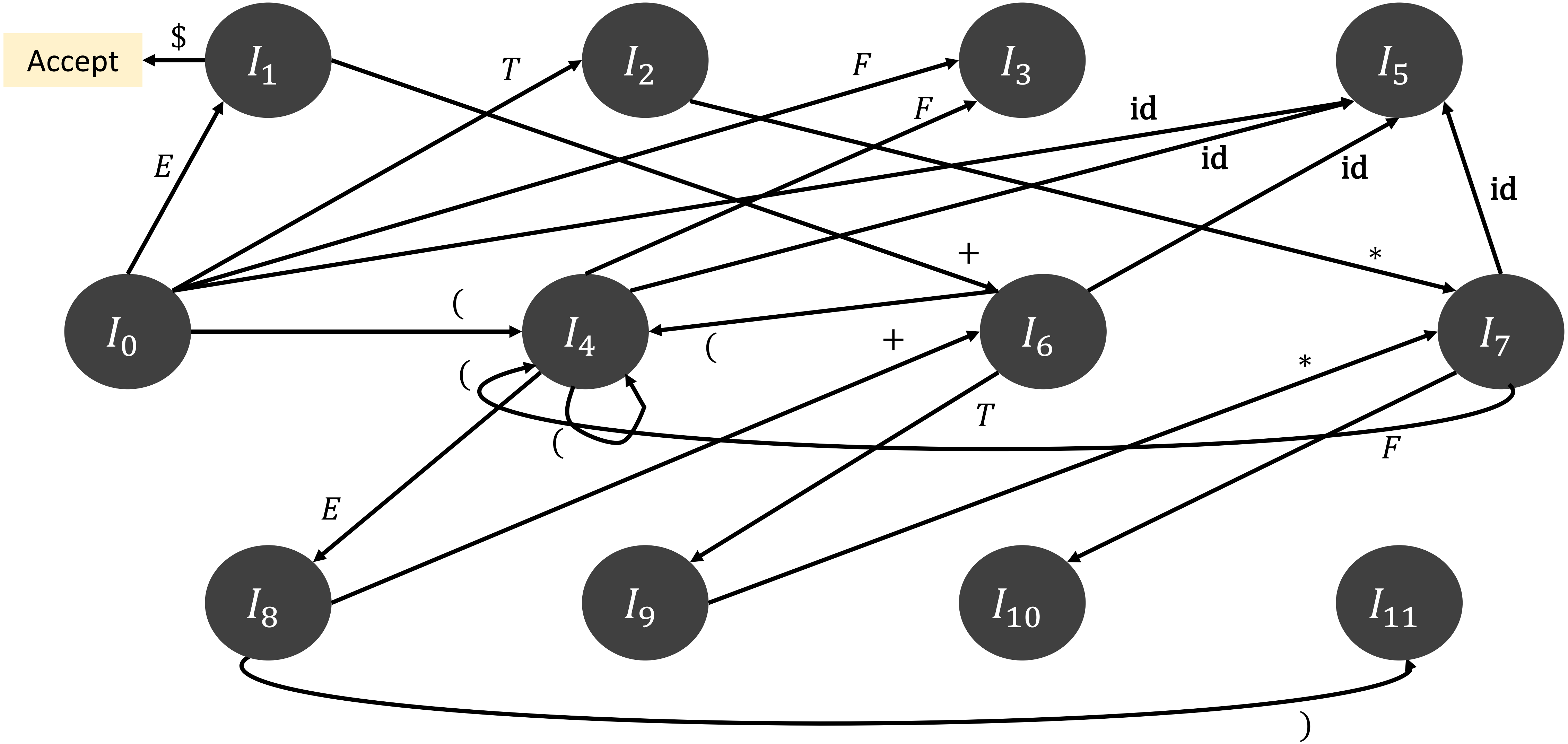
$I_9 = \text{Goto}(I_6, T) = \{$
$E \rightarrow E + T \bullet,$
$T \rightarrow T \bullet * F$
$\}$

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

$I_{11} = \text{Goto}(I_8, ")") = \{$
$F \rightarrow (E) \bullet$
$\}$

$I_2 = \text{Goto}(I_4, T)$
$I_3 = \text{Goto}(I_4, F)$
$I_4 = \text{Goto}(I_4, "(")$
$I_5 = \text{Goto}(I_4, \mathbf{id})$
$I_3 = \text{Goto}(I_6, F)$
$I_4 = \text{Goto}(I_6, "(")$
$I_5 = \text{Goto}(I_6, \mathbf{id})$
$I_4 = \text{Goto}(I_7, "(")$
$I_5 = \text{Goto}(I_7, \mathbf{id})$
$I_6 = \text{Goto}(I_8, +)$
$I_7 = \text{Goto}(I_9, *)$

LR(0) Automaton



SLR Parsing Table

State	ACTION						GOTO		
	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 Parser Configurations

- A LR parser configuration is a pair $<s_0, s_1, \ldots, s_m,\ a_i a_{i+1} \ldots a_n \$>$
  - Left half is stack content, and 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

- If ACTION[s_m, a_i] = shift s, new configuration is $<s_0, s_1, \ldots, s_m s, a_{i+1} \ldots a_n \$>$
- If ACTION[s_m, a_i] = reduce $A \rightarrow \beta$, new configuration is $<s_0, s_1, \ldots, s_{m-r}, a_i a_{i+1} \ldots a_n \$>$, where $r = |\beta|$ and s = GOTO[s_{m-r}, A]
- If ACTION[s_m, a_i] = accept, parsing is successful
- If ACTION[s_m, a_i] = error, parsing has discovered an error

LR Parsing Program

```
Let a be the first symbol of input w$

while (1)
    let s be the top of the stack
    if ACTION[a] == shift t
        push t onto the stack
        let a be the next input symbol
    else if ACTION[s, a] == reduce A → β
        pop |β| symbols off the stack
       push GOTO[t, A] onto the stack
       output production A → β
    else if ACTION[s, a] == accept
       break
    else
       invoke error recovery
```

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

	Stack	Symbols	Input	Action
1	0		$\mathbf{id} * \mathbf{id} + \mathbf{id}\$$	Shift
2	0 5	$\mathbf{id}$	$* \mathbf{id} + \mathbf{id}\$$	Reduce by $F \rightarrow \mathbf{id}$
3	0 3	F	$* \mathbf{id} + \mathbf{id}\$$	Reduce by $T \rightarrow F$
4	0 2	T	$* \mathbf{id} + \mathbf{id}\$$	Shift
5	0 2 7	$T *$	$\mathbf{id} + \mathbf{id}\$$	Shift
6	0 2 7 5	$T * \mathbf{id}$	$+\mathbf{id}\$$	Reduce by $F \rightarrow \mathbf{id}$
7	0 2 7 10	$T * F$	$+\mathbf{id}\$$	Reduce by $T \rightarrow T * F$
8	0 2	T	$+\mathbf{id}\$$	Reduce by $E \rightarrow T$
9	0 1	E	$+\mathbf{id}\$$	Shift
10	0 1 6	$E +$	$\mathbf{id}\$$	Shift

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

	Stack	Symbols	Input	Action
11	$0\ 1\ 6\ 5$	$E + \mathbf{id}$	$\$$	Reduce by $F \rightarrow \mathbf{id}$
12	$0\ 1\ 6\ 3$	$E + F$	$\$$	Reduce by $T \rightarrow F$
13	$0\ 1\ 6\ 9$	$E + T$	$\$$	Reduce by $E \rightarrow E + T$
14	$0\ 1$	E	$\$$	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)

Limitations of SLR Parsing

Unambiguous grammar
$S \rightarrow L = R \mid R$ $L \rightarrow *R \mid \mathbf{id}$ $R \rightarrow L$

Example Derivation

$S \Rightarrow L = R \Rightarrow *R = R$

$\text{FIRST}(S) = \text{FIRST}(L) = \text{FIRST}(R) = \{*, \mathbf{id}\}$

$\text{FOLLOW}(S) = \text{FOLLOW}(L) = \text{FOLLOW}(R)$
$= \{=, \$\}$

Canonical LR(0) Collection

I_0 = Closure($S' \rightarrow \bullet S$)= {
$S' \rightarrow \bullet S,$
$S \rightarrow \bullet L = R,$
$S \rightarrow \bullet R,$
$L \rightarrow \bullet * R,$
$L \rightarrow \bullet \mathbf{id},$
$R \rightarrow \bullet L$
}

I_1 = Goto(I_0, S) = {
$S' \rightarrow S\bullet$
}

I_2 = Goto(I_0, L)= {
$\boldsymbol{S \rightarrow L\bullet = R,}$
$\boldsymbol{R \rightarrow L\bullet}$
}

I_3 = Goto(I_0, R)= {
$S \rightarrow R\bullet$
}

I_4 =Goto(I_0, R)= {
$L \rightarrow * \bullet R,$
$R \rightarrow \bullet L,$
$L \rightarrow \bullet * R,$
$L \rightarrow \bullet \mathbf{id}$
}

I_6 = Goto(I_2, '=')= {
$S \rightarrow L = \bullet R,$
$R \rightarrow \bullet L,$
$L \rightarrow \bullet * R,$
$L \rightarrow \bullet \mathbf{id}$
}

I_5 = Goto(I_0, **id**)= {
$L \rightarrow \bullet \mathbf{id}$
}

I_7 =Goto(I_4, R)= {
$L \rightarrow * R\bullet$
}

I_8 = Goto(I_4, L)= {
$R \rightarrow L\bullet$
}

I_9 = Goto(I_6, R)= {
$S \rightarrow L = R\bullet$
}

SLR Parsing Table

State	ACTION				GOTO		
	=	*	id	$	S	L	R
0		$s4$	$s5$		1	2	3
1				acc			
2	$s6, r6$			$r6$			
3							
4		$s4$	$s5$			8	7
5	$r5$			$r5$			
6		$s4$	$s5$			8	9
7	$r4$			$r4$			
8	$r6$			$r6$			
9				$r2$			

Shift-Reduce Conflict with SLR Parsing

I_0 = Closure($S' \rightarrow . S$)= {

$S' \rightarrow \bullet S,$

$S \rightarrow \bullet L = R,$

$S \rightarrow \bullet R$

}

I_1 =

$S' \rightarrow S\bullet$

}

I_2 = Goto(I_0, L)= {

$\boldsymbol{S \rightarrow L\bullet = R,}$

$\boldsymbol{R \rightarrow L\bullet}$

}

I_3 = Goto(I_0, R)= {

$S \rightarrow R\bullet$

}

I_6 = Goto(I_2, ‘=’)= {

$S \rightarrow L = \bullet R,$

$R \rightarrow \bullet L,$

$L \rightarrow \bullet * R,$

$L \rightarrow \bullet \mathbf{id}$

}

I_5 = Goto(I_0, **id**)= {

$L \rightarrow \bullet \mathbf{id}$

}

I_9 = Goto(I_6, R)= {

$S \rightarrow L = R\bullet$

}

1. ACTION[2,=] = Shift 6, or
2. ACTION[2,=] = Reduce $R \rightarrow L$ since $" = " \in \mathrm{FOLLOW}(R)$

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

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

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

Stack	Input	Action
0	$\mathbf{id}{=}\mathbf{id}\$$	Shift 5
$0\ \mathbf{id}\ 5$	$=\mathbf{id}\$$	Reduce by $L \rightarrow \mathbf{id}$
$0\ L\ 2$	$=\mathbf{id}\$$	**Shift 6**
$0\ L\ 2 = 6$	$\mathbf{id}\$$	Shift 5
$0\ L\ 2 = 6\ \mathbf{id}\ 5$	$\$$	Reduce by $L \rightarrow \mathbf{id}$
$0\ L\ 2 = 6\ L\ 8$	$\$$	Reduce by $R \rightarrow L$
$0\ L\ 2 = 6\ R\ 9$	$\$$	Reduce by $S \rightarrow L = R$
$0\ S\ 1$	$\$$	Accept

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

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

- Suppose βA is a viable prefix on top of the stack
- There may be no right sentential form where a follows βA
  - Parser should not reduce by $A \rightarrow \alpha$

Stack	Input	Action
$0\ L\ 2 = 6\ R\ 9$	\$	Reduce by $S \rightarrow L = R$
$0\ S\ 1$	\$	Accept

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

Stack	Input	Action
0	**id=id**$	Shift 5

Stack	Input	Action
0	**id=id**$	Shift 5
[illegible]	$	[illegible]
$0\ L\ 2 = 6\ L\ 8$	$	Reduce by $R \rightarrow L$
$0\ L\ 2 = 6\ R\ 9$	$	Reduce by $S \rightarrow L = R$
$0\ S\ 1$	$	Accept

SLR parsers 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 next input symbol is a
  - Set of possible terminals will always be a subset of $\mathrm{FOLLOW}(A)$, but can be a proper subset

LR(1) Item

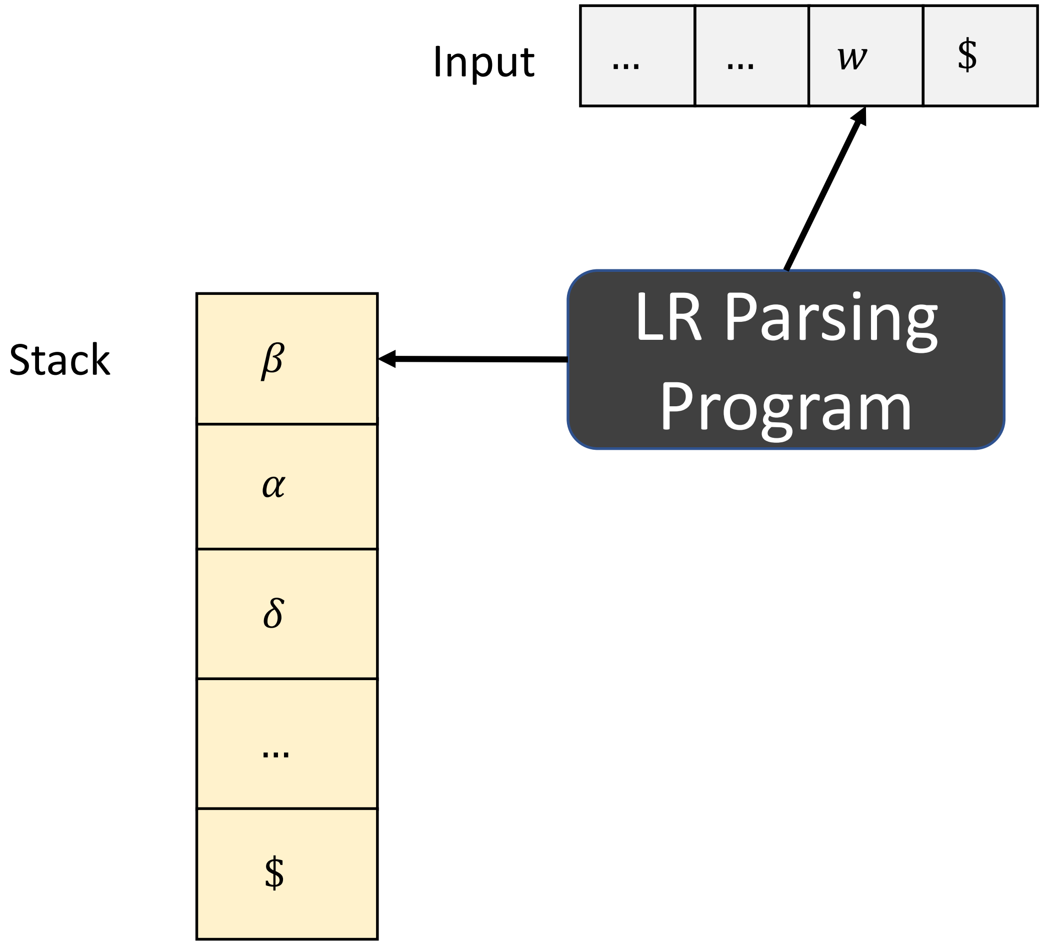
- An LR(1) item $[A \rightarrow \alpha \bullet \beta, a]$ is valid for a viable prefix γ if there is a derivation

$$S \underset{rm}{\Rightarrow}^{*} \delta A w \underset{rm}{\Rightarrow} \delta \alpha \beta w$$

where

i. $\gamma = \delta\alpha$, and

ii. a is the first symbol in w, or, $w = \epsilon$ and $a = \$$



Constructing LR(1) Sets of Items

Closure(I)

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

Goto(I, X)

initialize J to be the empty set
for each item $[A \to \alpha \bullet X\beta, a]$ in I
    add item $[A \to \alpha X \bullet \beta, a]$ to set J
return Closure(J)

Constructing LR(1) Sets of Items

Items(G'):

    C = Closure($\{[S' \rightarrow \bullet S, \$]\}$)

    repeat

    for each set of items I in C

        for each grammar symbol X

            if Goto(I, X) $\neq \phi$ and Goto(I, X) $\notin C$

                add Goto(I, X) to C

    until no new sets of items are added to C

Example Construction of LR(1) Items

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

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

I_0 = Closure($[S' \rightarrow \bullet S, \$]$)= {
$S' \rightarrow \bullet S, \$,$
$S \rightarrow \bullet CC, \$,$
$C \rightarrow \bullet \mathbf{c}C, \mathbf{c/d},$
$C \rightarrow \bullet \mathbf{d}, \mathbf{c/d}$
}

I_1 = Goto(I_0, S) = {
$S' \rightarrow S\bullet, \$$
}

Example Construction of LR(1) Items

$I_0 =$ Closure($[S' \rightarrow .S, \$]$)$= \{$
$S' \rightarrow \bullet S, \$,$
$S \rightarrow \bullet CC, \$,$
$C \rightarrow \bullet \mathbf{c}C, \mathbf{c/d},$
$C \rightarrow \bullet \mathbf{d}, \mathbf{c/d}$
$\}$

$I_1 =$ Goto(I_0, S) $= \{$
$S' \rightarrow S\bullet, \$$
$\}$

$I_2 =$ Goto(I_0, C)$= \{$
$S \rightarrow C\bullet C, \$,$
$C \rightarrow \bullet \mathbf{c}C, \$,$
$C \rightarrow \bullet \mathbf{d}, \$$
$\}$

$I_3 =$ Goto($I_0, \mathbf{c}$)$= \{$
$C \rightarrow \mathbf{c}\bullet C, \mathbf{c/d},$
$C \rightarrow \bullet \mathbf{c}C, \mathbf{c/d},$
$C \rightarrow \bullet \mathbf{d}, \mathbf{c/d}$
$\}$

$I_4 =$ Goto(I_0, d) $= \{$
$C \rightarrow \mathbf{d}\bullet, \mathbf{c/d}$
$\}$

$I_5 =$ Goto(I_2, C) $= \{$
$C \rightarrow CC\bullet, \$$
$\}$

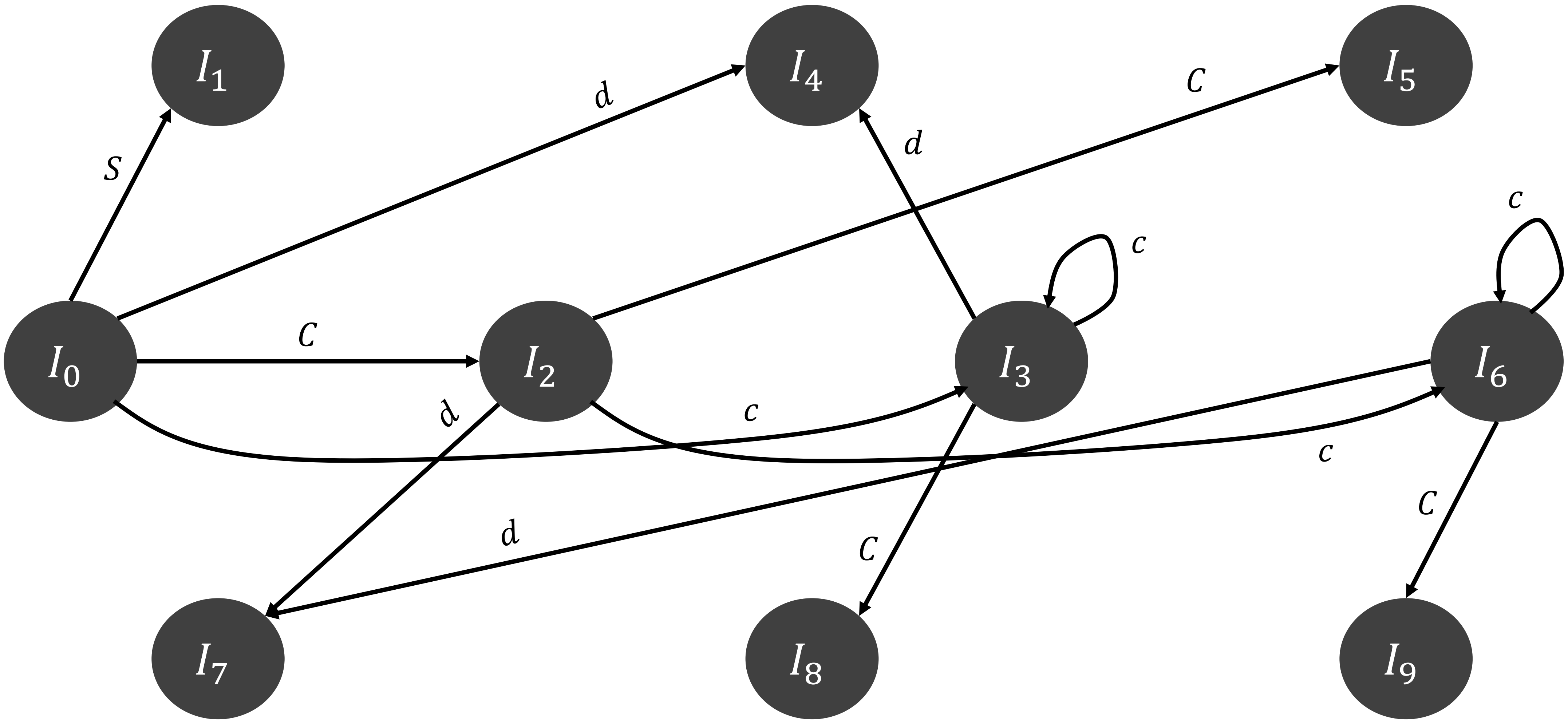
$I_6 =$ Goto($I_2, \mathbf{c}$)$= \{$
$C \rightarrow \mathbf{c}\bullet C, \$,$
$C \rightarrow \bullet \mathbf{c}C, \$,$
$C \rightarrow \bullet \mathbf{d}, \$$
$\}$

$I_7 =$ Goto($I_2, \mathbf{d}$) $= \{$
$C \rightarrow \mathbf{d}\bullet, \$$
$\}$

$I_8 =$ Goto(I_3, C) $= \{$
$C \rightarrow \mathbf{c}C\bullet, \mathbf{c/d}$
$\}$

$I_9 =$ Goto(I_6, C)$= \{$
$C \rightarrow \mathbf{c}C\bullet, \$$
$\}$

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 Goto(I_i,a) = I_j, then set ACTION[i, a]="shift j"
  - If $[A \rightarrow \alpha \bullet, a]$ is in I_i, $A \neq S'$, then set ACTION[i,a]="reduce $A \rightarrow \alpha \bullet$"
  - If $[S' \rightarrow S \bullet, \$]$ is in I_i, then set ACTION[i,\$]="accept"
- If Goto(I_i, A)= I_j, then 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

State	ACTION			GOTO	
	$\mathbf{c}$	$\mathbf{d}$	$\$$	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			$r2$		

Moves of a CLR Parser on **cdcd**

	Stack	Symbols	Input	Action
1	0		**cdcd**\$	Shift
2	0 3	**c**	**dcd**\$	Shift
3	0 3 4	**cd**	**cd**\$	Reduce by $C \rightarrow \mathbf{d}$
4	0 3 8	$\mathbf{c}C$	**cd**\$	Reduce by $C \rightarrow \mathbf{c}C$
5	0 2	C	**cd**\$	Shift
6	0 2 6	$C\mathbf{c}$	**d**\$	Shift
7	0 2 6 7	$C\mathbf{cd}$	\$	Reduce by $C \rightarrow \mathbf{d}$
8	0 2 6 9	$C\mathbf{c}C$	\$	Reduce by $C \rightarrow \mathbf{c}C$
9	0 2 5	CC	\$	Reduce by $S \rightarrow CC$
10	0 1	S	\$	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

I_0 = Closure($[S' \rightarrow .S, \$]$)= {
$S' \rightarrow \bullet S, \$,$
$S \rightarrow \bullet CC, \$,$
$C \rightarrow \bullet cC, c/d,$
$C \rightarrow \bullet d, c/d$
}

I_1 = Goto(I_0, S) = {
$S' \rightarrow S\bullet, \$$
}

I_2 = Goto(I_0, C)= {
$S \rightarrow C\bullet C, \$,$
$C \rightarrow \bullet cC, \$,$
$C \rightarrow \bullet d, \$$
}

I_3 = Goto(I_0, c)= {
$C \rightarrow c\bullet C, c/d,$
$C \rightarrow \bullet cC, c/d,$
$C \rightarrow \bullet d, c/d$
}

I_4 = Goto(I_0, d) = {
$C \rightarrow d\bullet, c/d$
}

I_5 = Goto(I_2, C) = {
$C \rightarrow CC\bullet, \$$
}

I_6 = Goto(I_2, c)= {
$C \rightarrow c\bullet C, \$,$
$C \rightarrow \bullet cC, \$,$
$C \rightarrow \bullet d, \$$
}

I_7 = Goto(I_2, d) = {
$C \rightarrow d\bullet, \$$
}

I_8 = Goto(I_3, C) = {
$C \rightarrow cC\bullet, c/d$
}

I_9 = Goto(I_6, C)= {
$C \rightarrow cC\bullet, \$$
}

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 a large number of states
- Lookahead LR (LALR) parser
  - Merge sets of LR(1) items that have the **same core** (set of LR(0) items, i.e., first component)
  - LALR parsers have fewer states, same as SLR
- LALR parser is used in many parser generators (e.g., Yacc and Bison)

Construction of LALR Parsing Table

- Construct $C = \{I_0, I_1, \dots, I_n\}$, the collection of sets of LR(1) items
- For each core present in LR(1) items, find all sets having the same core and replace these sets by their union
- Let $C' = \{J_0, J_1, \dots, J_n\}$ 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 $I_1, I_2, \dots, I_k$ are same
  - Cores of Goto(I_1, X), Goto(I_2, X), …, Goto(I_k, X) will also be the same
  - Let $K =$ Goto(I_1, X) ∪ Goto(I_2, X) ∪ … ∪ Goto(I_k, X), then Goto(J,X) = K

LALR Grammar

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

Rule #	Production
0	$S' \rightarrow S$
1	$S \rightarrow CC$
2	$C \rightarrow cC$
3	$C \rightarrow d$

I_{36} = Goto(I_0, c)= {
$C \rightarrow c\bullet C, c/d/\$,$
$C \rightarrow \bullet cC, c/d/\$,$
$C \rightarrow \bullet d, c/d/\$$
}

I_{47} = Goto(I_0, d) = {
$C \rightarrow d\bullet, c/d/\$$
}

I_{89} = Goto(I_3, C) = {
$C \rightarrow cC\bullet, c/d/\$$
}

LALR Parsing Table

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	$r2$	$r2$	$r2$		

Moves of a LALR Parser on **cdcd**

	Stack	Symbols	Input	Action
1	0		**cdcd**\$	Shift
2	0 36	**c**	**dcd**\$	Shift
3	0 36 47	**cd**	**cd**\$	Reduce by $C \rightarrow \mathbf{d}$
4	0 36 89	**c**C	**cd**\$	Reduce by $C \rightarrow \mathbf{c}C$
5	0 2	C	**cd**\$	Shift
6	0 2 36	C**c**	**d**\$	Shift
7	0 2 36 47	C**cd**	\$	Reduce by $C \rightarrow \mathbf{d}$
8	0 2 36 89	C**c**C	\$	Reduce by $C \rightarrow \mathbf{c}C$
9	0 2 5	CC	\$	Reduce by $S \rightarrow CC$
10	0 1	S	\$	Accept

Notes on LALR Parsing Table

- LALR parser behaves like the CLR parser excepting 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 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
$S' \rightarrow S$ $S \rightarrow aAd \mid bBd \mid aBe \mid bAe$ $A \rightarrow c$ $B \rightarrow c$
acd, ace, bcd, bce

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

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

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

Dealing with Errors with LALR Parsing

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

- Consider an erroneous input **ccd**

CLR Parsing Table

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			$r2$		

LALR Parsing Table

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	$r2$	$r2$	$r2$		

Comparing Moves of CLR and LALR Parsers

- Consider an erroneous input **ccd**

CLR Parsing Table			
Stack	**Symbols**	**Input**	**Action**
0		**ccd**\$	Shift
0 3	**c**	**cd**\$	Shift
0 3 3	**cc**	**d**\$	Shift
0 3 3 4	**ccd**	\$	Error

LALR Parsing Table			
Stack	**Symbols**	**Input**	**Action**
0		**ccd**\$	Shift
0 36	**c**	**cd**\$	Shift
0 36 36	**cc**	**d**\$	Shift
0 36 36 47	**ccd**	\$	Reduce by $C \rightarrow \mathbf{d}$
0 36 36 89	**cc**C	\$	Reduce by $C \rightarrow \mathbf{c}C$
0 36 89	**c**C	\$	Reduce by $C \rightarrow \mathbf{c}C$
0 2	C	\$	Error

Comparing Moves of CLR and LALR Parsers

- Consider an erroneous input **ccd**

CLR Parsing Table

LALR Parsing Table

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

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

Using Ambiguous Grammars

Dealing with Ambiguous Grammars

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

I_0 = Closure($\{E' \rightarrow \bullet E\}$) = {
$E' \rightarrow \bullet E,$
$E \rightarrow \bullet E + E,$
$E \rightarrow \bullet E * E,$
$E \rightarrow \bullet (E),$
$E \rightarrow \bullet \textbf{id}$
}

I_1 = Goto(I_0, E) = {
$E' \rightarrow E\bullet,$
$E \rightarrow E \bullet + E,$
$E \rightarrow E \bullet * E$
}

Does not specify the associativity and precedence of the two operators

I_2 = Goto(I_0, '(') = {
$E \rightarrow (\bullet E),$
$E \rightarrow \bullet E + E,$
$E \rightarrow \bullet E * E,$
$E \rightarrow \bullet (E),$
$E \rightarrow \bullet \textbf{id}$
}

I_3 = Goto(I_0, **id**) = {
$E \rightarrow \textbf{id}\bullet$
}

I_4 = Goto(I_0, '+') = {
$E \rightarrow E + \bullet E,$
$E \rightarrow \bullet E + E,$
$E \rightarrow \bullet E * E,$
$E \rightarrow \bullet (E),$
$E \rightarrow \bullet \textbf{id}$
}

I_9 = Goto(I_6, ')') = {
$E \rightarrow (E)\bullet$
}

I_5 = Goto(I_0, '*') = {
$E \rightarrow E * \bullet E,$
$E \rightarrow \bullet E + E,$
$E \rightarrow \bullet E * E,$
$E \rightarrow \bullet (E),$
$E \rightarrow \bullet \textbf{id}$
}

I_6 = Goto(I_2, E) = {
$E \rightarrow (E\bullet),$
$E \rightarrow E \bullet + E,$
$E \rightarrow E \bullet * E,$
}

I_7 = Goto(I_4, E) = {
$\boldsymbol{E \rightarrow E + E\bullet,}$
$\boldsymbol{E \rightarrow E \bullet + E,}$
$\boldsymbol{E \rightarrow E \bullet * E}$
}

I_8 = Goto(I_5, E) = {
$\boldsymbol{E \rightarrow E * E\bullet,}$
$\boldsymbol{E \rightarrow E \bullet + E,}$
$\boldsymbol{E \rightarrow E \bullet * E}$
}

SLR(1) Parsing Table

State	ACTION						GOTO
	id	$+$	$*$	(	)	\$	E
0	$s3$			$s2$			1
1		$s4$	$s5$			acc	
2	$s3$			$s2$			6
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 $\mathbf{id} + \mathbf{id} * \mathbf{id}$

	Stack	Symbols	Input	Action
1	0		$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	Shift 3
2	$0\ 3$	$\mathbf{id}$	$+\mathbf{id} * \mathbf{id}\$$	Reduce by $E \rightarrow \mathbf{id}$
3	$0\ 1$	E	$+\mathbf{id} * \mathbf{id}\$$	Shift 4
4	$0\ 1\ 4$	$E +$	$\mathbf{id} * \mathbf{id}\$$	Shift 3
5	$0\ 1\ 4\ 3$	$E + \mathbf{id}$	$* \mathbf{id}\$$	Reduce by $E \rightarrow \mathbf{id}$
6	$0\ 1\ 4\ 7$	$E + E$	$* \mathbf{id}\$$	

What can the parser do to resolve the ambiguity?

SLR(1) Parsing Table

State	Action						Goto
	id	$+$	$*$	(	)	\$	E
0	$s3$			$s2$			1
1		$s4$	$s5$			acc	
2	$s3$			$s2$			6
3		$r4$	$r4$		$r4$	$r4$	
4	[illegible]			$s2$			7
[illegible]	[illegible]			$s2$			8
6		$s4$	$s5$		$s9$		
7		$s4$, $\mathbf{r1}$	$\mathbf{s5}$, $r1$		$r1$	$r1$	
8		$s4$, $\mathbf{r2}$	$s5$, $\mathbf{r2}$		$r2$	$r2$	
9		$r3$	$r3$		$r3$	$r3$	

Why did the parser make these choices?

Summary

Comparison across LR Parsing Techniques

- 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(1) $\leq$ LALR(1) $\leq$ LR(1)
- SLR(k) $\leq$ LALR(k) $\leq$ LR(k)
- LL(k) $\leq$ LR(k)
- Ambiguous grammars are not LR

Summary

- 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

References

- A. Aho et al. Compilers: Principles, Techniques, and Tools, 2nd edition, Chapter 4.5-4.8.
- K. Cooper and L. Torczon. Engineering a Compiler, 2nd edition, Chapter 3.4.