

CS 335: Top-Down Parsing

Swarnendu Biswas

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

Sem 2023-24-II



Example Expression Grammar

Start → *Expr*

Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*

Term → *Term* × *Factor* | *Term* ÷ *Factor* | *Factor*

Factor → (*Expr*) | **num** | **name**

↓ priority

Derivation of $\text{name} + \text{name} \times \text{name}$ with Oracular Knowledge

Sentential Form	Input
Expr	$\uparrow \text{name} + \text{name} \times \text{name}$
$\text{Expr} + \text{Term}$	$\uparrow \text{name} + \text{name} \times \text{name}$
$\text{Term} + \text{Term}$	$\uparrow \text{name} + \text{name} \times \text{name}$
$\text{Factor} + \text{Term}$	$\uparrow \text{name} + \text{name} \times \text{name}$
$\text{name} + \text{Term}$	$\uparrow \text{name} + \text{name} \times \text{name}$
$\text{name} + \text{Term}$	$\text{name} \uparrow + \text{name} \times \text{name}$
$\text{name} + \text{Term}$	$\text{name} + \uparrow \text{name} \times \text{name}$
$\text{name} + \text{Term} \times \text{Factor}$	$\text{name} + \uparrow \text{name} \times \text{name}$
$\text{name} + \text{Factor} \times \text{Factor}$	$\text{name} + \uparrow \text{name} \times \text{name}$
$\text{name} + \text{name} \times \text{Factor}$	$\text{name} + \uparrow \text{name} \times \text{name}$
$\text{name} + \text{name} \times \text{Factor}$	$\text{name} + \text{name} \uparrow \times \text{name}$
$\text{name} + \text{name} \times \text{Factor}$	$\text{name} + \text{name} \times \uparrow \text{name}$
$\text{name} + \text{name} \times \text{name}$	$\text{name} + \text{name} \times \uparrow \text{name}$
$\text{name} + \text{name} \times \text{name}$	$\text{name} + \text{name} \times \text{name} \uparrow$

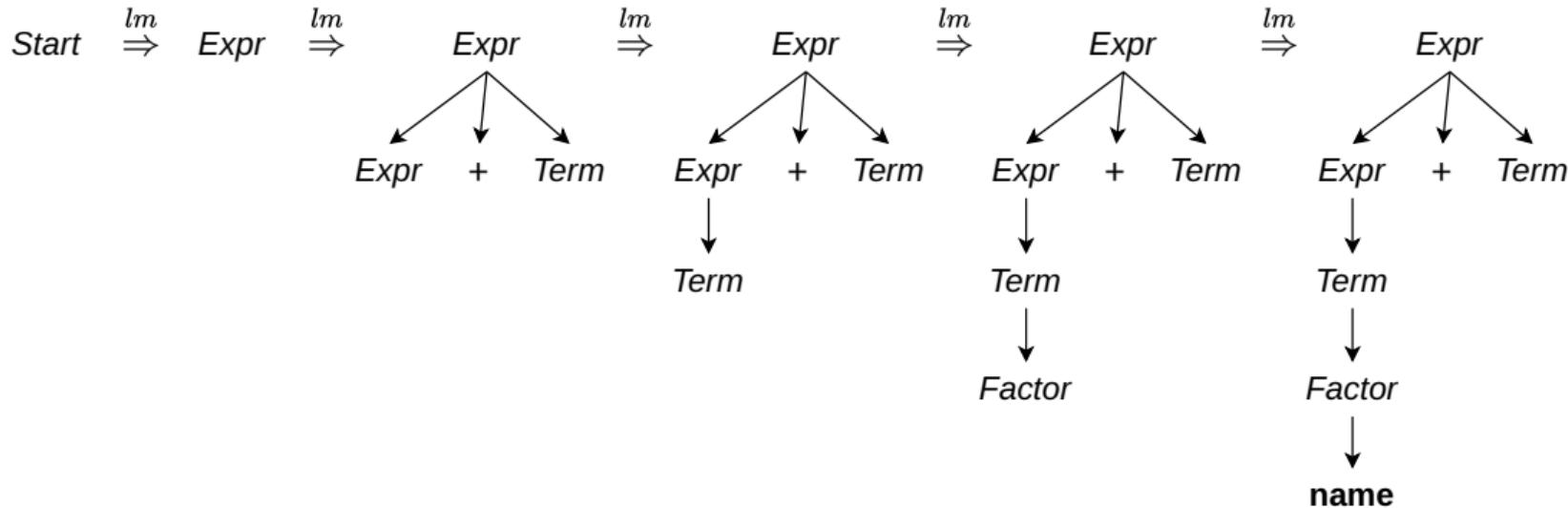
Derivation of **name + name × name** with Oracular Knowledge

Sentential Form	Input
<i>Expr</i>	↑ name + name × name
<i>Expr + Term</i>	↑ name + name × name
<i>Term + Term</i>	↑ name + name × name
<i>Factor + Term</i>	↑ name + name × name
<i>name + Term</i>	↑ name + name × name

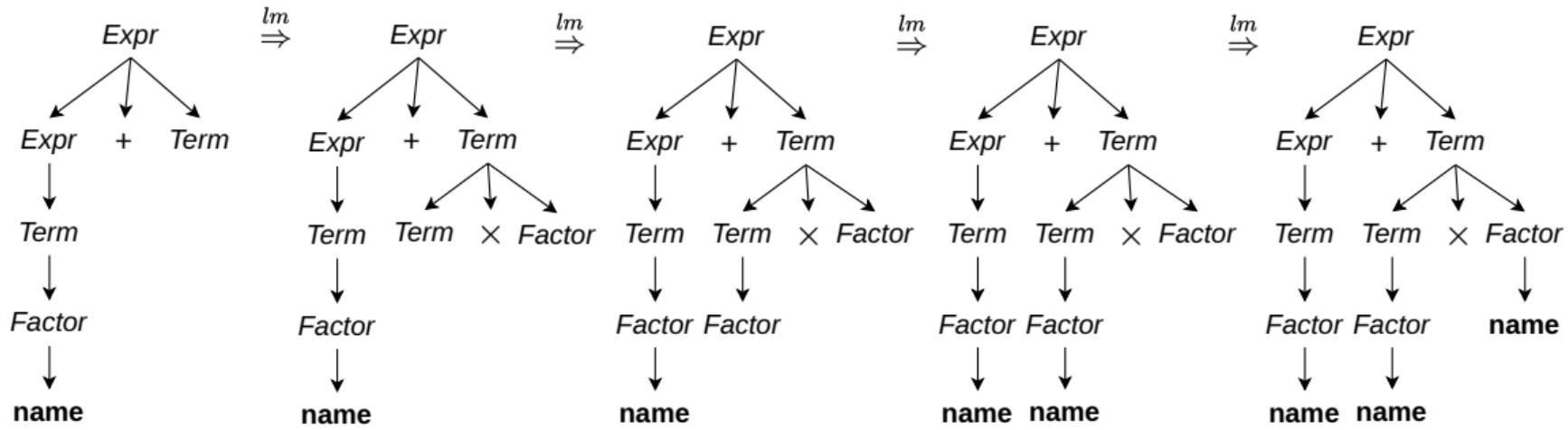
The current input terminal being scanned is called the lookahead symbol

name + Factor × Factor	name + ↑ name × name
name + name × Factor	name + ↑ name × name
name + name × Factor	name + name ↑ × name
name + name × Factor	name + name × ↑ name
name + name × name	name + name × ↑ name
name + name × name	name + name × name ↑

Derivation of **name + name** × **name** with Oracular Knowledge



Derivation of **name + name × name** with Oracular Knowledge



Top-Down Parsing

High-level idea in top-down parsing

- (i) Start with the root (i.e., start symbol) of the parse tree
- (ii) Grow the tree downwards by expanding the production at the lower levels of the tree
 - ▶ Select a nonterminal and extend it by adding children corresponding to the right side of some production for the nonterminal
- (iii) Repeat till the lower fringe consists only of terminals and the input is consumed

- Top-down parsing finds a **leftmost derivation** for an input string
- Expands the parse tree with a **preorder depth-first** traversal

Top-Down Parsing

High-level idea in top-down parsing

- (i) Start with the root (i.e., start symbol) of the parse tree
- (ii) Grow the tree downwards by expanding the production at the lower levels of the tree
 - ▶ Select a nonterminal and extend it by adding children corresponding to the right side of some production for the nonterminal
- (iii) Repeat till the lower fringe consists only of terminals and the input is consumed

Mismatch in the lower fringe and the remaining input stream implies

- (i) Wrong choice of productions while expanding nonterminals, selection of a production may involve trial-and-error
- (ii) Input character stream is not part of the language

Top-Down Parsing Algorithm

```
root = node for the Start symbol
curr = root
push(null) // Stack

word = getNextWord()
while (true)
    if curr ∈ Nonterminal
        pick next rule  $A \rightarrow \beta_1\beta_2\dots\beta_n$  to expand curr
        create nodes for  $\beta_1, \beta_2, \dots, \beta_n$  as children of curr
        push( $\beta_n\beta_{n-1}\dots\beta_1$ ) // reverse order
        curr =  $\beta_1$ 
    if curr == word
        word = getNextWord()
        curr = pop() // Consumed
    if word == EOF and curr == null
        accept input
    else
        backtrack
```

Derivation of $\text{name} + \text{name} \times \text{name}$

Rule #	Production
0	$\text{Start} \rightarrow \text{Expr}$
1	$\text{Expr} \rightarrow \text{Expr} + \text{Term}$
2	$\text{Expr} \rightarrow \text{Expr} - \text{Term}$
3	$\text{Expr} \rightarrow \text{Term}$
4	$\text{Term} \rightarrow \text{Term} \times \text{Factor}$
5	$\text{Term} \rightarrow \text{Term} \div \text{Factor}$
6	$\text{Term} \rightarrow \text{Factor}$
7	$\text{Factor} \rightarrow (\text{Expr})$
8	$\text{Factor} \rightarrow \text{num}$
9	$\text{Factor} \rightarrow \text{name}$

Rule #	Sentential Form	Input
	Expr	$\uparrow \text{name} + \text{name} \times \text{name}$
1	$\text{Expr} + \text{Term}$	$\uparrow \text{name} + \text{name} \times \text{name}$
3	$\text{Term} + \text{Term}$	$\uparrow \text{name} + \text{name} \times \text{name}$
6	$\text{Factor} + \text{Term}$	$\uparrow \text{name} + \text{name} \times \text{name}$
9	$\text{name} + \text{Term}$	$\uparrow \text{name} + \text{name} \times \text{name}$
	$\text{name} + \text{Term}$	$\text{name} \uparrow + \text{name} \times \text{name}$
	$\text{name} + \text{Term}$	$\text{name} + \uparrow \text{name} \times \text{name}$
4	$\text{name} + \text{Term} \times \text{Factor}$	$\text{name} + \uparrow \text{name} \times \text{name}$
4	$\text{name} + \text{Factor} \times \text{Factor}$	$\text{name} + \uparrow \text{name} \times \text{name}$
9	$\text{name} + \text{name} \times \text{Factor}$	$\text{name} + \uparrow \text{name} \times \text{name}$
	$\text{name} + \text{name} \times \text{Factor}$	$\text{name} + \text{name} \uparrow \times \text{name}$
	$\text{name} + \text{name} \times \text{Factor}$	$\text{name} + \text{name} \times \uparrow \text{name}$
9	$\text{name} + \text{name} \times \text{name}$	$\text{name} + \text{name} \times \uparrow \text{name}$
	$\text{name} + \text{name} \times \text{name}$	$\text{name} + \text{name} \times \text{name} \uparrow$

Derivation of $\text{name} + \text{name} \times \text{name}$

Rule #	Production	Rule #	Sentential Form	Input
0	$\text{Start} \rightarrow \text{Expr}$		Expr	$\uparrow \text{name} + \text{name} \times \text{name}$
1	$\text{Expr} \rightarrow \text{Expr} + \text{Term}$	1	$\text{Expr} + \text{Term}$	$\uparrow \text{name} + \text{name} \times \text{name}$
2	$\text{Expr} \rightarrow \text{Expr} - \text{Term}$	3	$\text{Term} + \text{Term}$	$\uparrow \text{name} + \text{name} \times \text{name}$
3	$\text{Expr} \rightarrow \text{Term}$	6	$\text{Factor} + \text{Term}$	$\uparrow \text{name} + \text{name} \times \text{name}$
4	$\text{Term} \rightarrow \text{Term} \times \text{Factor}$	9	$\text{name} + \text{Term}$	$\uparrow \text{name} + \text{name} \times \text{name}$
5	$\text{Term} \rightarrow \text{Term} + \text{Factor}$			$+ \text{name} \times \text{name}$
6	$\text{Term} \rightarrow \text{Term} - \text{Factor}$			$\uparrow \text{name} \times \text{name}$
7	$\text{Factor} \rightarrow \text{num}$			$\uparrow \text{name} \times \text{name}$
8	$\text{Factor} \rightarrow \text{name}$	4	$\text{name} + \text{Factor} \times \text{Factor}$	$\text{name} + \uparrow \text{name} \times \text{name}$
9		9	$\text{name} + \text{name} \times \text{Factor}$	$\text{name} + \uparrow \text{name} \times \text{name}$
			$\text{name} + \text{name} \times \text{Factor}$	$\text{name} + \text{name} \uparrow \times \text{name}$
			$\text{name} + \text{name} \times \text{Factor}$	$\text{name} + \text{name} \times \uparrow \text{name}$
		9	$\text{name} + \text{name} \times \text{name}$	$\text{name} + \text{name} \times \uparrow \text{name}$
			$\text{name} + \text{name} \times \text{name}$	$\text{name} + \text{name} \times \text{name} \uparrow$

How does a top-down parser choose which rule to apply?

Deterministically Selecting a Production in Expression Grammar

Rule #	Production
0	$Start \rightarrow Expr$
1	$Expr \rightarrow Expr + Term$
2	$Expr \rightarrow Expr - Term$
3	$Expr \rightarrow Term$
4	$Term \rightarrow Term \times Factor$
5	$Term \rightarrow Term \div Factor$
6	$Term \rightarrow Factor$
7	$Factor \rightarrow (Expr)$
8	$Factor \rightarrow num$
9	$Factor \rightarrow name$

Rule #	Sentential Form	Input
	$Expr$	$\uparrow name + name \times name$
1	$Expr + Term$	$\uparrow name + name \times name$
1	$Expr + Term + Term$	$\uparrow name + name \times name$
1	$Expr + Term + Term + \dots$	$\uparrow name + name \times name$
1	\dots	$\uparrow name + name \times name$
1	\dots	$\uparrow name + name \times name$

Deterministically Selecting a Production in Expression Grammar

Rule #	Production
0	$Start \rightarrow Expr$
1	$Expr \rightarrow Expr + Term$
2	$Expr \rightarrow Expr - Term$
3	$Expr \rightarrow Term$
4	$Term \rightarrow Term \times Factor$
5	$Term \rightarrow name$
6	$Term \rightarrow num$
7	$Factor \rightarrow name$
8	$Factor \rightarrow num$
9	$Factor \rightarrow name$

Rule #	Sentential Form	Input
	$Expr$	$\uparrow name + name \times name$
1	$Expr + Term$	$\uparrow name + name \times name$
1	$Expr + Term + Term$	$\uparrow name + name \times name$
1	$Expr + Term + Term + \dots$	$\uparrow name + name \times name$
1	$name + name \times name$	$+ name \times name$

A top-down parser can loop indefinitely
with left-recursive grammar

Left Recursion

A grammar is left-recursive if it has a nonterminal A such that there is a derivation $A \xrightarrow{+} A\alpha$ for some string α

Direct There is a production of the form $A \rightarrow A\alpha$

Indirect The first symbol on the right-hand side of a rule can derive the symbol on the left

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow Ac \mid Sd \mid \epsilon \end{aligned}$$

We can often reformulate a grammar to avoid left recursion

Remove Direct Left Recursion

Grammar with left recursion

$$A \rightarrow A\alpha_1 | A\alpha_2 | \dots | A\alpha_m | \beta_1 | \dots | \beta_n$$

Grammar without left recursion

$$A \rightarrow \beta_1 A' | \beta_2 A' | \dots | \beta_n A'$$

$$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \dots | \alpha_m A' | \epsilon$$

Example

$$E \rightarrow E + T | T$$

$$T \rightarrow T * F | F$$

$$F \rightarrow (E) | \text{id}$$

\Rightarrow

$$E \rightarrow TE'$$

$$E' \rightarrow +TE'$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT'$$

$$F \rightarrow (E) | \text{id}$$

Non-Left-Recursive Expression Grammar

Expression Grammar with Recursion

Rule #	Production
0	$Start \rightarrow Expr$
1	$Expr \rightarrow Expr + Term$
2	$Expr \rightarrow Expr - Term$
3	$Expr \rightarrow Term$
4	$Term \rightarrow Term \times Factor$
5	$Term \rightarrow Term \div Factor$
6	$Term \rightarrow Factor$
7	$Factor \rightarrow (Expr)$
8	$Factor \rightarrow num$
9	$Factor \rightarrow name$

Expression Grammar without Recursion

Rule #	Production
0	$Start \rightarrow Expr$
1	$Start \rightarrow Term Expr'$
2	$Expr' \rightarrow +Term Expr'$
3	$Expr' \rightarrow -Term Expr'$
4	$Expr' \rightarrow \epsilon$
5	$Term \rightarrow Factor Term'$
6	$Term \rightarrow \times Factor Term'$
7	$Term \rightarrow \div Factor Term'$
8	$Term' \rightarrow \epsilon$
9	$Factor \rightarrow (Expr)$
10	$Factor \rightarrow num$
11	$Factor \rightarrow name$

Eliminating Indirect Left Recursion

- **Input:** Grammar G with no cycles or ϵ -productions
- **Algorithm:**

```
Arrange nonterminals in some order  $A_1, A_2, \dots, A_n$ 
for  $i \leftarrow 1 \dots n$ 
  for  $j \leftarrow 1 \dots i-1$ 
    if  $\exists$  a production  $A_i \rightarrow A_j \gamma$ 
      Replace  $A_i \rightarrow A_j \gamma$  with one or more productions that expand  $A_j$ 
      Eliminate the immediate left recursion among the  $A_i$  productions
```

Loop invariant at the start of the outer iteration i

$\forall k < i$, no production expanding A_k has A_l in its body (i.e., right-hand side) for all $l < k$

The algorithm establishes a topological ordering on nonterminals

Eliminating Indirect Left Recursion

- Input: Grammar G with no cycles or ϵ -productions
- Algorithm:

```
Arrange nonterminals in some order  $A_1, A_2, \dots, A_n$ 
for  $i \leftarrow 1 \dots n$ 
  for  $j \leftarrow 1 \dots i-1$ 
    if  $\exists$  a production  $A_i \rightarrow A_j \gamma$ 
      Replace  $A_i \rightarrow A_j \gamma$  with one or more productions that expand  $A_j$ 
      Eliminate the immediate left recursion among the  $A_i$  productions
```

$$\begin{array}{ccc} S \rightarrow Aa \mid b & \Rightarrow & S \rightarrow Aa \mid b \\ A \rightarrow Ac \mid Sd \mid \epsilon & & A \rightarrow bdA' \mid A' \\ & & A' \rightarrow cA' \mid adA' \mid \epsilon \end{array}$$

Implementing Backtracking

- A top-down parser may need to undo its actions after it detects a mismatch between the parse tree's leaves and the input
 - ▶ Implies a possible expansion with a wrong production
- Steps in backtracking
 - ▶ Set curr to parent and delete the children
 - ▶ Expand the node curr with **untried rules** if any
 - ▶ Create child nodes for each symbol in the right hand of the production
 - ▶ Push those symbols onto the stack in reverse order
 - ▶ Set curr to the first child node
 - ▶ **Move curr up the tree** if there are no untried rules
 - ▶ Report a syntax error when there are no more moves

Backtracking is Expensive

- (i) Parser expands a nonterminal with the wrong rule
- (ii) Mismatch between the lower fringe of the parse tree and the input is detected
- (iii) Parser undoes the last few actions
- (iv) Parser tries other productions (if any)

A large subset of CFGs can be parsed without backtracking

The grammar may require transformations

Avoid Backtracking

- Parser is to select the next rule
 - ▶ Compare the curr symbol and the next input symbol called the lookahead
 - ▶ Use the lookahead to disambiguate the possible production rules
- Intuition
 - ▶ Each alternative for the leftmost nonterminal leads to a **distinct terminal symbol**
 - ▶ Which rules to choose becomes obvious by comparing the next word in the input stream

Definition

Backtrack-free grammar (also called predictive grammar) is a CFG for which a leftmost, top-down parser can always predict the correct rule with a one-word lookahead

FIRST Set

Definition

Given a string γ of terminal and nonterminal symbols, FIRST (γ) is the set of all terminal symbols that can begin any string derived from γ

- We also need to keep track of which symbols can produce the empty string
- $\text{FIRST} : (NT \cup T \cup \{\epsilon, \text{EOF}\}) \rightarrow (T \cup \{\epsilon, \text{EOF}\})$
- Steps to compute FIRST set
 1. If X is a terminal, then $\text{FIRST}(X) = \{X\}$
 2. If $X \rightarrow \epsilon$ is a production, then $\epsilon \in \text{FIRST}(X)$
 3. If X is a nonterminal and $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production, then
 - (i) $\text{FIRST}(X) = \text{FIRST}(Y_1)$ provided $Y_1 \not\rightarrow \epsilon$
 - (ii) If for some $i \leq k$ and $1 \leq j < i$, $a \in \text{FIRST}(Y_i)$, and $\forall j, \epsilon \in \text{FIRST}(Y_j)$, then $a \in \text{FIRST}(X)$
 - (iii) If $\epsilon \in \text{FIRST}(Y_1, \dots, Y_k)$, then $\epsilon \in \text{FIRST}(X)$
- Generalization of FIRST relation to string of symbols
 - $\text{FIRST}(X\gamma) = \text{FIRST}(X)$ if $X \not\rightarrow \epsilon$
 - $\text{FIRST}(X\gamma) = \text{FIRST}(X) \cup \text{FIRST}(\gamma)$ if $X \rightarrow \epsilon$

Example of FIRST Set Computation

Grammar

$Start \rightarrow Expr$

$Expr \rightarrow Term\ Expr'$

$Expr' \rightarrow +\ Term\ Expr' \mid -\ Term\ Expr' \mid \epsilon$

$Term \rightarrow Factor\ Term'$

$Term' \rightarrow \times\ Factor\ Term' \mid \div\ Factor\ Term' \mid \epsilon$

$Factor \rightarrow (Expr) \mid num \mid name$

FIRST Sets

$\text{FIRST}(Start) = \{\text{name}, \text{num}, ()\}$

$\text{FIRST}(Expr) = \{\text{name}, \text{num}, ()\}$

$\text{FIRST}(Expr') = \{+, -, \epsilon\}$

$\text{FIRST}(Term) = \{\text{name}, \text{num}, ()\}$

$\text{FIRST}(Term') = \{\times, \div, \epsilon\}$

$\text{FIRST}(Factor) = \{\text{name}, \text{num}, ()\}$

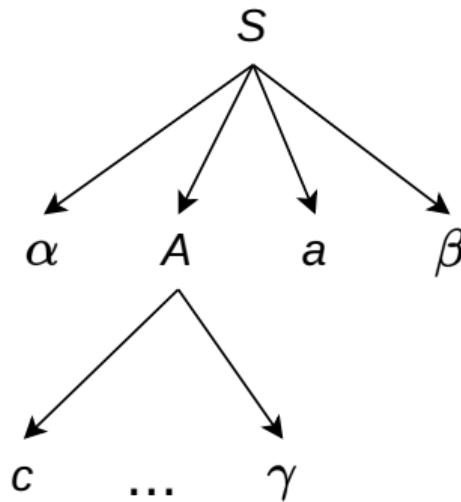
How does a parser decide when to apply the ϵ -production?

FOLLOW Set

Definition

FOLLOW (X) is the set of terminals that can immediately follow X

- That is, $t \in \text{FOLLOW} (X)$ if there is any derivation containing Xt



Terminal c is in FIRST (A) and a is in FOLLOW (A)

Steps to Compute FOLLOW Set

- (i) Place $\$$ in FOLLOW (S) where S is the start symbol and the $\$$ is the end marker
- (ii) If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST (β) except ϵ is in FOLLOW (B)
- (iii) If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST (β) contains ϵ , then everything in FOLLOW (A) is in FOLLOW (B)

Example of FOLLOW Set Computation

Grammar

$Start \rightarrow Expr$

$Expr \rightarrow Term\ Expr'$

$Expr' \rightarrow +\ Term\ Expr' \mid -\ Term\ Expr' \mid \epsilon$

$Term \rightarrow Factor\ Term'$

$Term' \rightarrow \times\ Factor\ Term' \mid \div\ Factor\ Term' \mid \epsilon$

$Factor \rightarrow (Expr) \mid num \mid name$

FOLLOW Sets

$\text{FOLLOW}(Start) = \{\$\}$

$\text{FOLLOW}(Expr) = \{\$,)\}$

$\text{FOLLOW}(Expr') = \{\$,)\}$

$\text{FOLLOW}(Term) = \{\$, +, -,)\}$

$\text{FOLLOW}(Term') = \{\$, +, -,)\}$

$\text{FOLLOW}(Factor) = \{\$, +, -, \times, \div,)\}$

Conditions for Backtrack-Free Grammar

- Consider a production $A \rightarrow \beta$

$$\text{FIRST}^+ (A \rightarrow \beta) = \begin{cases} \text{FIRST}(\beta) & \text{if } \epsilon \notin \text{FIRST}(\beta) \\ \text{FIRST}(\beta) \cup \text{FOLLOW}(A) & \text{otherwise} \end{cases}$$

- For any nonterminal A where $A \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$, a **backtrack-free grammar** has the property

$$\text{FIRST}^+(A \rightarrow \beta_i) \cap \text{FIRST}^+(A \rightarrow \beta_j) = \emptyset, \quad \forall 1 \leq i, j \leq n, i \neq j$$

Expression grammar on the previous slide is backtrack-free

Not All Grammars are Backtrack-Free

$Start \rightarrow Expr$

$Expr \rightarrow Term\ Expr'$

$Expr' \rightarrow +\ Term\ Expr' \mid -\ Term\ Expr' \mid \epsilon$

$Term \rightarrow Factor\ Term'$

$Term' \rightarrow \times\ Factor\ Term' \mid \div\ Factor\ Term' \mid \epsilon$

$Factor \rightarrow (Expr) \mid num \mid name$

$Factor \rightarrow name \mid name[Arglist] \mid name(Arglist)$

$Arglist \rightarrow Expr\ MoreArgs$

$MoreArgs \rightarrow ,\ Expr\ MoreArgs \mid \epsilon$

Not All Grammars are Backtrack-Free

$Start \rightarrow Expr$

$Expr \rightarrow Term\ Expr'$

$Expr' \rightarrow +\ Term\ Expr' \mid -\ Term\ Expr' \mid \epsilon$

$Term \rightarrow Factor\ Term'$

$Term' \rightarrow \times\ Factor\ Term' \mid \div\ Factor\ Term' \mid \epsilon$

$Factor \rightarrow (Expr) \mid num \mid name$

$Factor \rightarrow name \mid name[Arglist] \mid name(Arglist)$

$Arglist \rightarrow Expr\ MoreArgs$

$MoreArgs \rightarrow ,\ Expr\ MoreArgs \mid \epsilon$

Given a finite lookahead, we can always devise a non-backtrack-free grammar such that the lookahead is insufficient

Left Factoring

Definition

Left factoring is the process of extracting and isolating common prefixes in a set of productions

- **Algorithm:**

$$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \dots | \alpha\beta_n | \gamma_1 | \dots | \gamma_j$$



$$\begin{aligned} A &\rightarrow \alpha B | \gamma_1 | \gamma_2 \dots | \gamma_j \\ B &\rightarrow \beta_1 | \beta_2 | \dots | \beta_n \end{aligned}$$

Summarizing Top-down Parsing

- Efficiency depends on the accuracy of selecting the correct production for expanding a nonterminal
 - ▶ Parser may not terminate in the worst-case
- A large subset of the context-free grammars can be parsed without backtracking

Recursive-Descent Parsing

Recursive-Descent Parsing

- Recursive-descent parsing is a form of top-down parsing that **may require** backtracking
 - ▶ Top-down approach is modeled by calls to functions, where there is one function for each nonterminal

```
void A() {  
    Choose an A-production  $A \rightarrow X_1 X_2 \dots X_k$   
    for  $i \leftarrow 1 \dots k$   
        if  $X_i$  is a nonterminal  
            call function  $X_i$   
        else if  $X_i$  equals the current input symbol  $a$   
            advance the input to the next symbol  
        else  
            // error  
}
```

Recursive-Descent Parsing with Backtracking

- Consider a grammar with two productions $X \rightarrow \gamma_1$ and $X \rightarrow \gamma_2$
- Suppose $\text{FIRST}(\gamma_1) \cap \text{FIRST}(\gamma_2) \neq \emptyset$
 - ▶ Let us denote one of the common terminal symbols by a
- The function for X will not know which production to use on the input token a
- To support backtracking
 - ▶ All productions should be tried in some order
 - ▶ Failure for some production implies the parser needs to try the remaining productions
 - ▶ Report an error only when there are no other rules

Predictive Parsing

Definition

Predictive parsing is a special case of recursive-descent parsing that does not require backtracking

- Lookahead symbol unambiguously determines which production rule to use
- Advantage is that the algorithm is simple and the parser can be constructed by hand

$$\begin{aligned}stmt &\rightarrow \mathbf{expr}; \\&| \mathbf{if} (\mathit{expr}) \mathit{stmt} \\&| \mathbf{for} (\mathit{optexpr}; \mathit{optexpr}; \mathit{optexpr}) \mathit{stmt} \\&| \mathbf{other} \\ \mathit{optexpr} &\rightarrow \mathbf{expr} \mid \epsilon\end{aligned}$$

Pseudocode for a Predictive Parser

```
void stmt() {
    switch(lookahead) {
        case expr: { match(expr); match(';'); break; }
        case if: {
            match(if); match('('); match(expr); match(')');
            stmt(); break;
        }
        case for: {
            match(for); match('('); optexpr(); match(';');
            optexpr(); match(')');
            stmt(); break;
        }
        case other: { match(other); break; }
        default: { print("syntax error"); }
    }
}
```

Non-Recursive Predictive Parsing

LL(k) Grammars

Definition

A CFG $G = (T, NT, S, P)$ is LL(1) if and only if for every nonterminal $A \in NT$ where $A \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$ such that $\beta_i \in \Sigma^*$, we have

$$\text{FIRST}^+ (A \rightarrow \beta_i) \cap \text{FIRST}^+ (A \rightarrow \beta_j) = \phi, \quad \forall 1 \leq i, j \leq n, i \neq j$$

- First L stands for left-to-right scan, second L stands for leftmost derivation, and k represents the number of lookahead tokens
- LL(k) grammars are the class of CFGs for which no backtracking is required
 - ▶ Predictive parsers accept LL(k) grammars
- Every LL(1) grammar is a LL(2) grammar
- Many programming language constructs are LL(1)

Definition of LL(k) Grammar

- For a given word $w \in T^*$ and non-negative integer k ,
 - ▶ w/k is w if $|w| \leq k$, or
 - ▶ w/k is a string consisting of the first k symbols of w if $|w| > k$.
- A CFG $G = (T, NT, S, P)$ is LL(k) for some positive integer k if and only if given
 - (i) a word $w \in T^*$ such that $|w| \leq k$,
 - (ii) a nonterminal $A \in NT$, and
 - (iii) a word $w_1 \in T^*$,there is at most one production $p \in P$ such that for some $w_2, w_3 \in T^*$,

1. $S \Rightarrow w_1 Aw_3$,
2. $A \stackrel{+}{\Rightarrow} w_2$ by first applying production p ,
3. $w_2w_3/k = w$.

Definition of LL(k) Grammar

- For a given word $w \in T^*$ and non-negative integer k ,
 - ▶ w/k is w if $|w| \leq k$, or
 - ▶ w/k is a string consisting of the first k symbols of w if $|w| > k$.
- A CFG $G = (T, NT, S, P)$ is LL(k) for some positive integer k if and only if given

Stated informally in terms of parsing, an LL(k) grammar is a CFG such that for any word in its language, each production in its derivation can be identified with certainty by inspecting the word from its beginning (left end) to the k^{th} symbol beyond the beginning of the production.

2. $A \xrightarrow{+} w_2$ by first applying production p ,
3. $w_2w_3/k = w$.

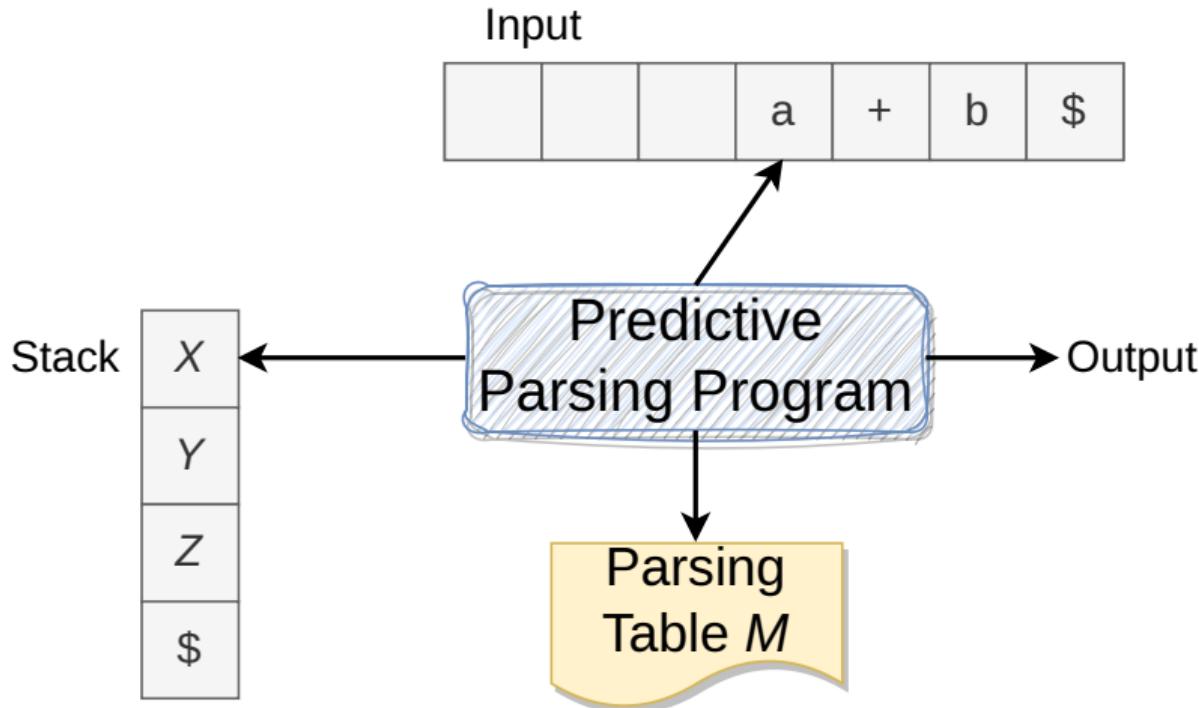
Example LL(2) Parser

$$S \rightarrow AXQ \mid AYR$$

lookahead is the set of 2-sequence tokens that indicate which alternative will succeed

```
void S() {
    if (lookahead(1) == A && lookahead(2) == X) {
        match(A); match(X); match(Q);
    } else if (lookahead(1) == A && lookahead(2) == Y) {
        match(A); match(Y); match(R);
    } else {
        // Raise error
    }
}
```

Nonrecursive Table-Driven LL(1) Parser



LL(1) Parsing Algorithm

- **Input:** String w and parsing table M for grammar G
- **Output:** A leftmost derivation of w if $w \in L(G)$; otherwise, report an error
- **Algorithm:**

```
Let  $a$  be the first symbol in  $w$ 
Let  $X$  be the symbol at the top of the stack
while  $X \neq \$$ 
    if  $X == a$ 
        pop the stack and advance the input
    else if  $X$  is a terminal or  $M[X,a]$  is an error entry
        report error
    else if  $M[X,a] == X \rightarrow Y_1 Y_2 \dots Y_k$ 
        // Expand with the production  $X \rightarrow Y_1 Y_2 \dots Y_k$ 
        pop the stack
        // Simulate depth-first traversal
        push  $Y_k Y_{k-1} \dots Y_1$  onto the stack
     $X \leftarrow$  top stack symbol
```

Construction of a LL(1) Parsing Table

- Input: Grammar G
- Algorithm:

```
for each production  $A \rightarrow \alpha$  in  $G$ 
    for each terminal  $a$  in FIRST( $\alpha$ )
        add  $A \rightarrow \alpha$  to  $M[A,a]$ 
    if  $\epsilon \in \text{FIRST}(\alpha)$ 
        for each terminal  $b$  in FOLLOW( $A$ )
            add  $A \rightarrow \alpha$  to  $M[A,b]$ 
    if  $\epsilon \in \text{FIRST}(\alpha)$  and  $\$ \in \text{FOLLOW}(A)$ 
        add  $A \rightarrow \alpha$  to  $M[A,\$]$ 

// No production in  $M[A,a]$  indicates error
```

LL(1) Parsing Table

Grammar

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \epsilon \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

FIRST Sets

$$\begin{aligned} \text{FIRST}(E) &= \{\text{id}, ()\} \\ \text{FIRST}(E') &= \{+, \epsilon\} \\ \text{FIRST}(T) &= \{\text{id}, ()\} \\ \text{FIRST}(T') &= \{*, \epsilon\} \\ \text{FIRST}(F) &= \{\text{id}, ()\} \end{aligned}$$

FOLLOW Sets

$$\begin{aligned} \text{FOLLOW}(E) &= \{\$\}, ()\} \\ \text{FOLLOW}(E') &= \{\$\}, ()\} \\ \text{FOLLOW}(T) &= \{\$\}, +, ()\} \\ \text{FOLLOW}(T') &= \{\$\}, +, ()\} \\ \text{FOLLOW}(F) &= \{\$\}, +, *, ()\} \end{aligned}$$

Nonterminal	id	+	*	()	\$
E	$E \rightarrow TE'$				$E \rightarrow TE'$	
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$				$T \rightarrow FT'$	
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{id}$				$F \rightarrow (E)$	

Working of a LL(1) Parser

Stack	Input	Remark
\$E	↑ id + id * id\$	Expand $E \rightarrow TE'$
\$E' T	↑ id + id * id\$	Expand $T \rightarrow FT'$
\$E' T' F	↑ id + id * id\$	Expand $F \rightarrow id$
\$E' T' id	↑ id + id * id\$	Match id
\$E' T'	↑ + id * id\$	Expand $T \rightarrow \epsilon$
\$E'	↑ + id * id\$	Expand $E' \rightarrow +TE'$
\$E' T +	↑ + id * id\$	Match +
\$E' T	↑ id * id\$	Expand $T \rightarrow FT'$
\$E' T' F	↑ id * id\$	Expand $F \rightarrow id$
\$E' T' id	↑ id * id\$	Match id
\$E' T'	↑ * id\$	Expand $T' \rightarrow *FT'$
\$E' T' F *	↑ * id\$	Match *
\$E' T' F	↑ id\$	Expand $F \rightarrow id$
\$E' T' id	↑ id\$	Match id
\$E' T'	↑ \$	Expand $T' \rightarrow \epsilon$
\$E'	↑ \$	Expand $E' \rightarrow \epsilon$
\$	↑ \$	

More on LL(1) Parsing

- Grammars whose predictive parsing tables contain no duplicate entries are LL(1)
- No left-recursive or ambiguous grammar can be LL(1)
 - ▶ If grammar G is left-recursive or is ambiguous, then parsing table M will have at least one multiply-defined cell
- Some grammars cannot be transformed into LL(1)

The below grammar is ambiguous

$$S \rightarrow iEtSS' \mid a$$

$$S' \rightarrow eS \mid \epsilon$$

$$E \rightarrow b$$

Limitations with LL(k) Parsing

LL(k) cannot see past **arbitrarily** long constructs from the left edge

$$S \rightarrow A^+ XQ \mid A^+ YR$$

Could left factor, but not always possible and natural

$$S \rightarrow A^+ (XQ \mid YR)$$

Programming language grammars may not be LL(k) (e.g., C function declaration vs definition)

$$\begin{aligned} \text{func} &\rightarrow \text{type ID } '(' \text{ arg* } ')' ';' \\ &\rightarrow \text{type ID } '(' \text{ arg* } ')' '\{ \text{ body } '\} \end{aligned}$$

Using Ambiguous Grammars

LL(1) Parsing Table for an Ambiguous Grammar

Grammar

$$S \rightarrow iEtSS' \mid a$$

$$S' \rightarrow eS \mid \epsilon$$

$$E \rightarrow b$$

FIRST Sets

$$\text{FIRST}(S) = \{i, a\}$$

$$\text{FIRST}(S') = \{e, \epsilon\}$$

$$\text{FIRST}(E) = \{b\}$$

FOLLOW Sets

$$\text{FOLLOW}(S) = \{\$\}, e\}$$

$$\text{FOLLOW}(S') = \{\$\}, e\}$$

$$\text{FOLLOW}(E) = \{t\}$$

Nonterminal	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$		$S' \rightarrow \epsilon$	
E		$E \rightarrow b$				

Detecting Errors in Table-Driven Predictive Parsing

Error conditions

- (i) Terminal on top of the stack does not match the next input symbol
- (ii) Nonterminal A is on top of the stack, a is the next input symbol, and $M[A, a]$ is empty

Choices

- (i) Raise an error and quit parsing
- (ii) Print an error message, try to recover from the error, and continue with the compilation

Error Recovery in Table-Driven Predictive Parsing

Assume A is the nonterminal at the top of the stack

Panic mode recovery skips over symbols until a token in a set of synchronizing (synch) tokens is found

- (i) Add all tokens in FOLLOW (A) to the synch set for A
 - ▶ Parsing can continue if the parser skips all input tokens until it sees an input symbol in FOLLOW (A)
- (ii) Add symbols in FIRST (A) to the synch set for A
 - ▶ Parsing can continue with A if the parser skips all input tokens until it sees an input symbol in FIRST (A)
- (iii) Add keywords that begin constructs
- (iv) Skip input if the table does not have an entry
- (v) ...

Using FOLLOW Sets as Synchronizing Tokens

Grammar

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

$$F \rightarrow (E) \mid \text{id}$$

FOLLOW Sets

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

$$\text{FOLLOW}(T) = \text{FOLLOW}(T') = \{\$, +,)\}$$

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

Nonterminal	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{id}$	synch	synch	$F \rightarrow (E)$	synch	synch

Error Recovery Moves by Table-Driven Predictive Parser

Stack	Input	Remark
$\$E$	$+ \mathbf{id} * + \mathbf{id}\$$	Error, skip $+$
$\$E$	$\mathbf{id} * + \mathbf{id}\$$	Expand $E \rightarrow TE'$
$\$E' T$	$\mathbf{id} * + \mathbf{id}\$$	Expand $T \rightarrow FT'$
$\$E' T' F$	$\mathbf{id} * + \mathbf{id}\$$	Expand $F \rightarrow \mathbf{id}$
$\$E' T' \mathbf{id}$	$\mathbf{id} * + \mathbf{id}\$$	Match \mathbf{id}
$\$E' T'$	$* + \mathbf{id}\$$	Expand $T \rightarrow *FT'$
$\$E' T' F *$	$* + \mathbf{id}\$$	Match $*$
$\$E' T' F$	$+ \mathbf{id}\$$	Error, $M[F, +] = \text{synch}$, pop F
$\$E' T'$	$+ \mathbf{id}\$$	Expand $T \rightarrow \epsilon$
$\$E'$	$+ \mathbf{id}\$$	Expand $E' \rightarrow +TE'$
$\$E' T +$	$+ \mathbf{id}\$$	Match $+$
$\$E' T$	$\mathbf{id}\$$	Expand $T \rightarrow FT'$
$\$E' T' F$	$\mathbf{id}\$$	Expand $F \rightarrow \mathbf{id}$
$\$E' T' \mathbf{id}$	$\mathbf{id}\$$	Match \mathbf{id}
$\$E' T'$	$\$$	Expand $T' \rightarrow \epsilon$
$\$E'$	$\$$	Expand $E' \rightarrow \epsilon$
$\$$	$\$$	

References

-  A. Aho et al. Compilers: Principles, Techniques, and Tools. Sections 2.4, 4.2–4.4, 2nd edition, Pearson Education.
-  K. Cooper and L. Torczon. Engineering a Compiler. Section 3.3, 2nd edition, Morgan Kaufmann.

CS 335: Bottom-up Parsing

Swarnendu Biswas

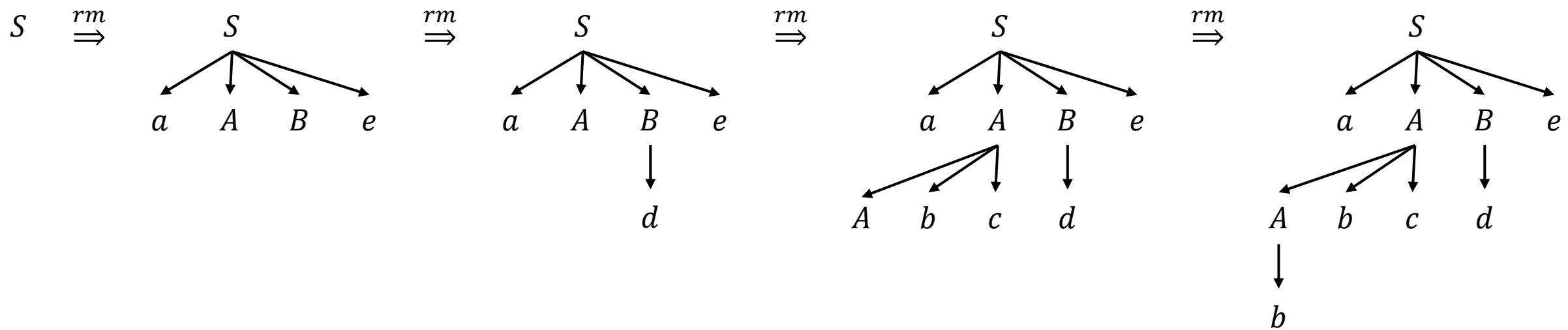
Semester 2022-2023-II
CSE, IIT Kanpur

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: $abbcde$



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

$$\begin{aligned}S &\rightarrow aABe \\A &\rightarrow Abc \mid b \\B &\rightarrow d\end{aligned}$$

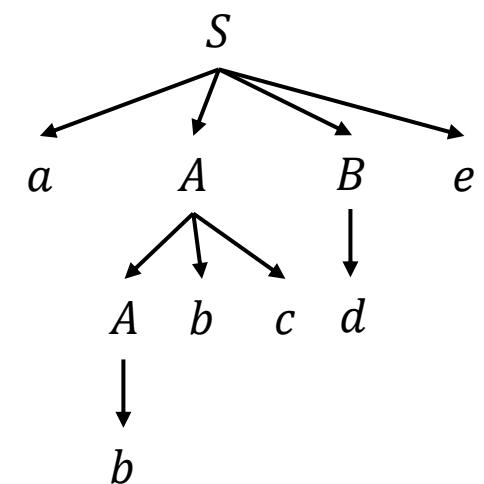
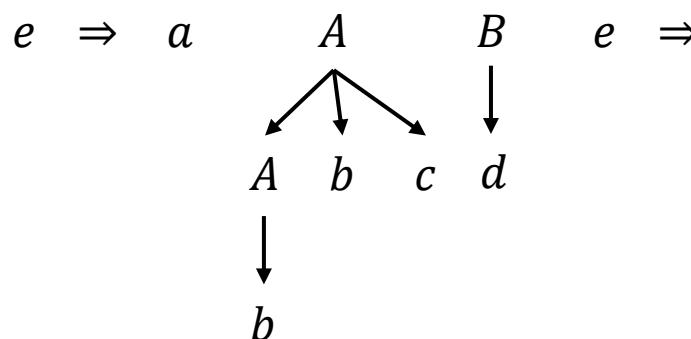
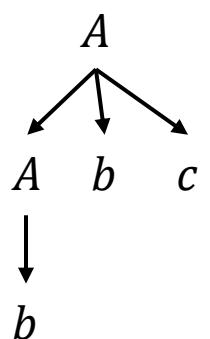
Input string: *abbcde*

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

abbcde \Rightarrow *a A b c d e* \Rightarrow *a*

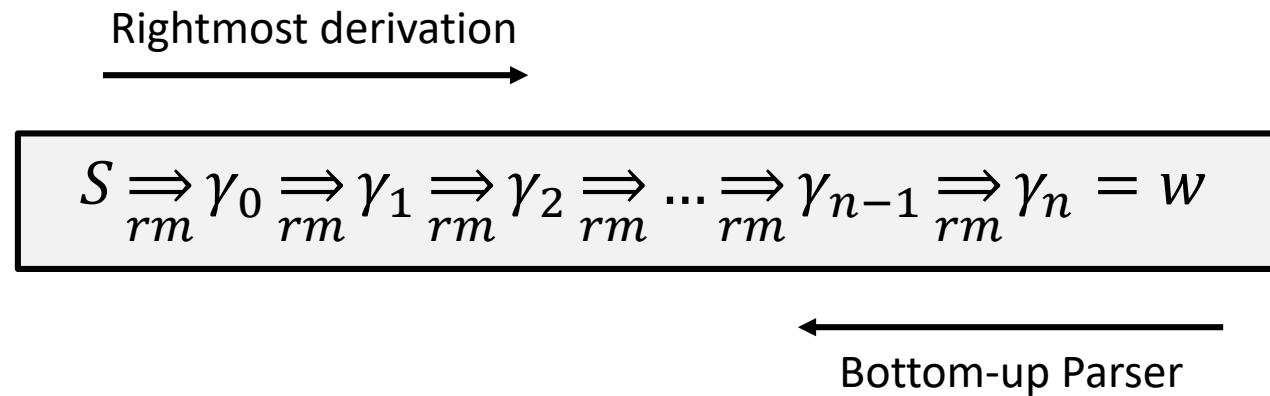


b



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



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

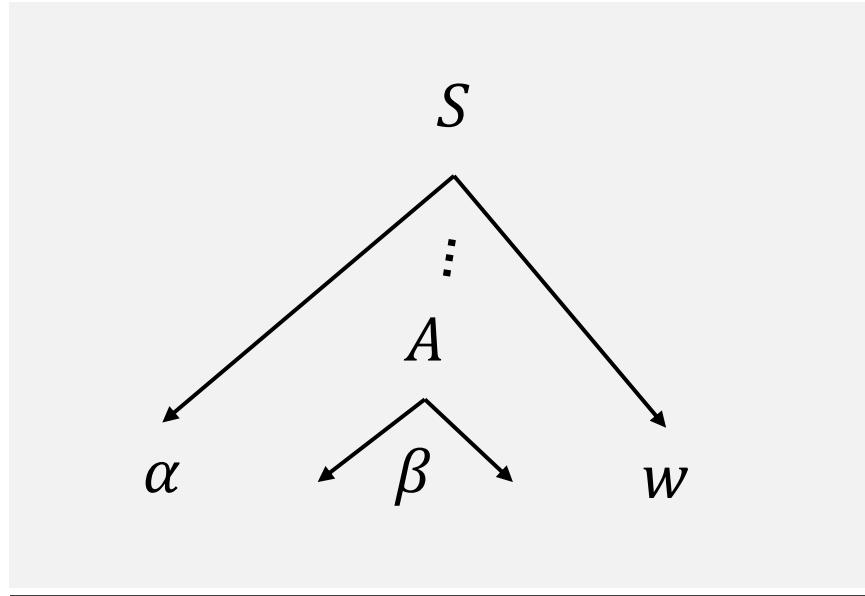
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \text{id}$

Right Sentential Form	Handle	Reducing Production
$\text{id}_1 * \text{id}_2$	id_1	$F \rightarrow \text{id}$
$F * \text{id}_2$	F	$T \rightarrow F$
$T * \text{id}_2$	id_2	$F \rightarrow \text{id}$
$T * F$	$T * F$	$T \rightarrow T * F$
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 $T * \text{id}_2$. The leftmost substring that matches the body of some production need not be a handle.

Handle

- If $S \xrightarrow{r_m}^* \alpha A w \xrightarrow{r_m} \alpha \beta w$, 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

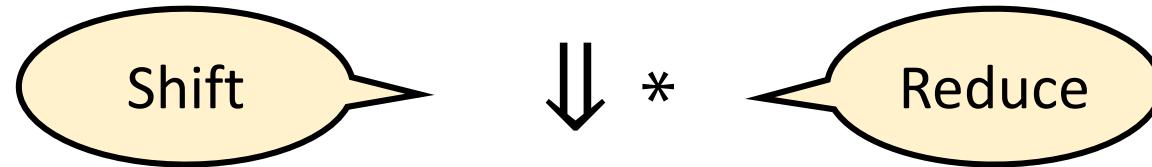
Shift-Reduce Parsing

- 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\$



- Final goal

Stack	Input
\$S	\$

Shift-Reduce Parsing

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

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

Or report an error in
case of a syntax error

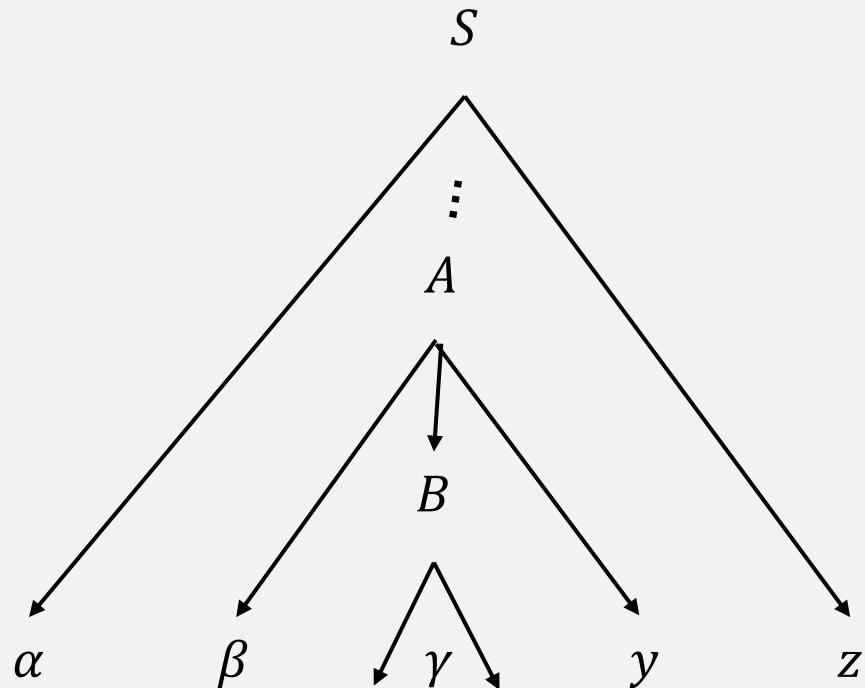
Handle on Top of the Stack

- Is the following scenario possible?

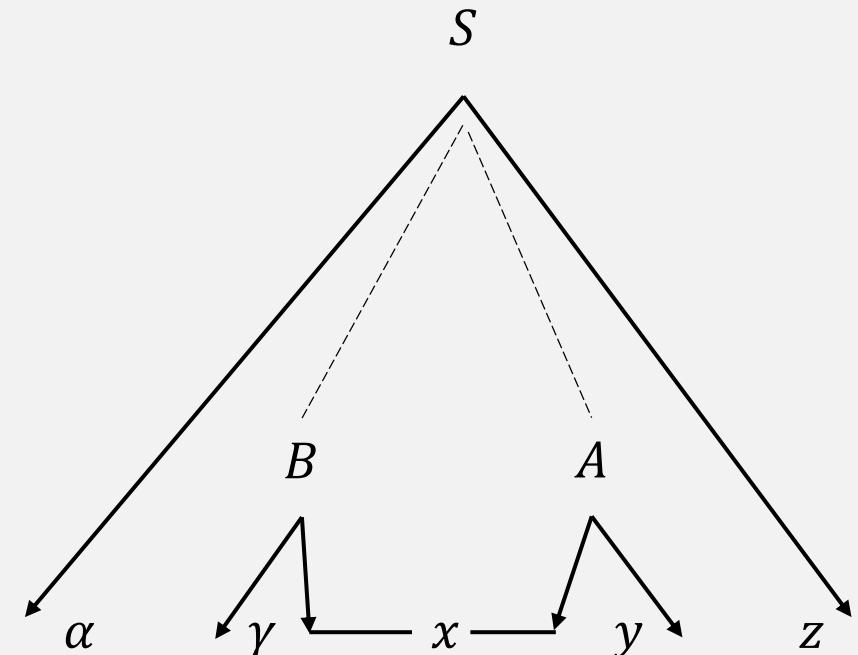
Stack	Input	Action
...		
$\$ \alpha \beta \gamma$	$w \$$	Reduce by $A \rightarrow \gamma$
$\$ \alpha \beta A$	$w \$$	Reduce by $B \rightarrow \beta$
$\$ \alpha B A$	$w \$$	
...		

Possible Choices in Rightmost Derivation

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



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



Handle on Top of the Stack

- Is the following scenario possible?

Stack	Input	Action
Handle	Push	Pop
...		

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

$$E \rightarrow E + E \mid E * E \mid id$$

id + id * id

Stack	Input	Action
\$	id + id * id\$	Shift
...		
\$E + E	* id\$	Reduce by $E \rightarrow E + E$
\$E	* id\$	Shift
\$E *	id\$	Shift
\$E * id	\$	Reduce by $E \rightarrow id$
\$E * E	\$	Reduce by $E \rightarrow E * E$
\$E	\$	

id + id * id

Stack	Input	Action
\$	id + id * id\$	Shift
...		
\$E + E	* id\$	Shift
\$E + E *	id\$	Shift
\$E + E * id	\$	Reduce by $E \rightarrow id$
\$E + E * E	\$	Reduce by $E \rightarrow E * E$
\$E + E	\$	Reduce by $E \rightarrow E + E$
\$E	\$	

Shift-Reduce Conflict

$$\begin{aligned} \text{Stmt} \rightarrow & \text{ if } \text{Expr} \text{ then Stmt} \\ | & \text{ if } \text{Expr} \text{ then Stmt else Stmt} \\ | & \text{ other} \end{aligned}$$

Stack	Input	Action
$\dots \text{if } \text{Expr} \text{ then Stmt}$	$\text{else } \dots \$$	

What is a correct thing to do for this grammar – shift or reduce?
E.g., we can prioritize shifts.

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

$$R \rightarrow c$$

Reduce-Reduce Conflict

$c + c$

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

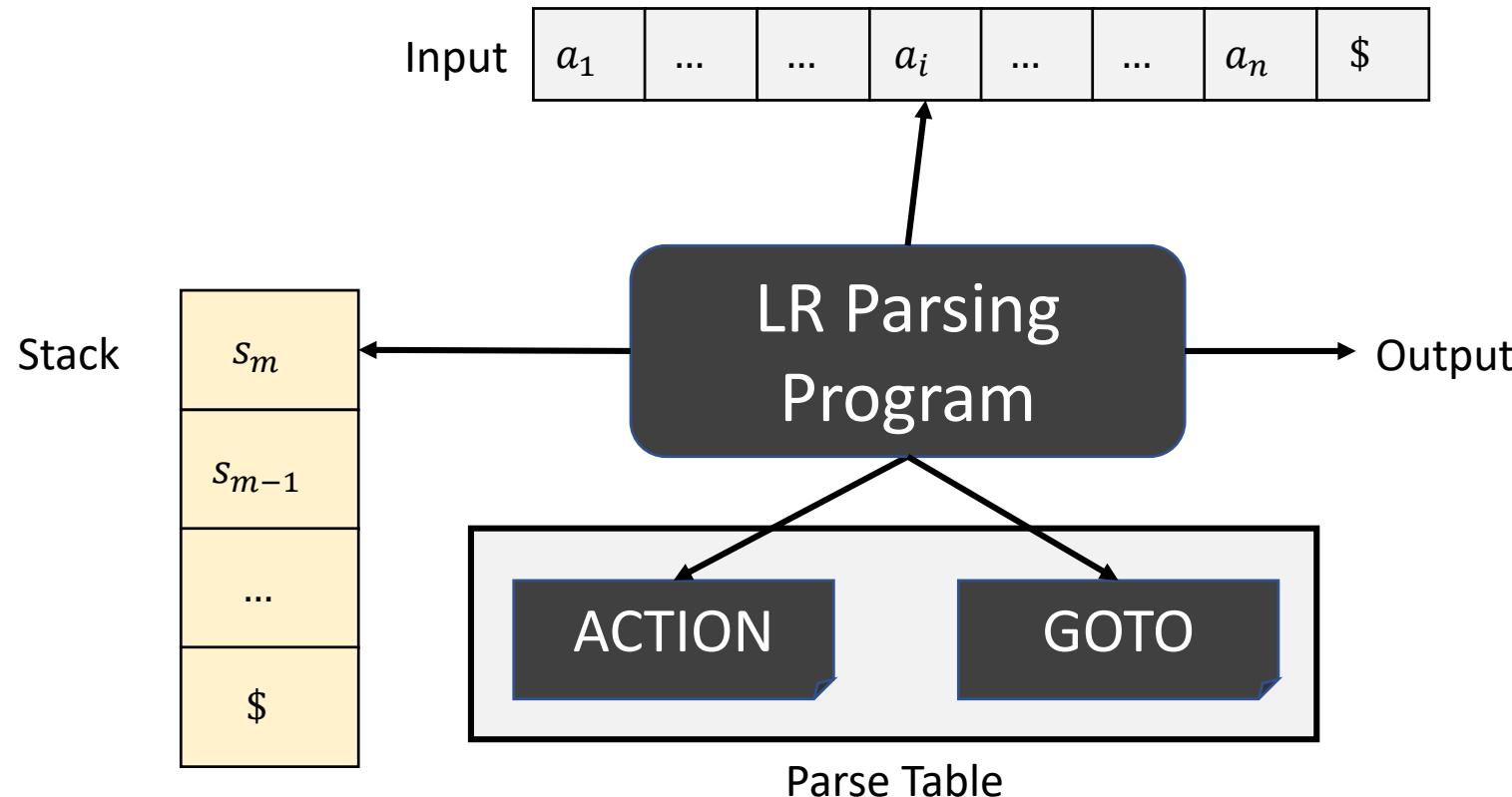
Can recognize almost all language constructs with CFGs

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

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

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

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

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

Kernel and Nonkernel Items

- If one B -production is added to $\text{Closure}(I)$ 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
- $\text{Goto}(I, X)$ 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 $\text{Goto}(I, X)$ is set of valid items for prefix αX
- Intuitively, $\text{Goto}(I, X)$ defines the transitions in the LR(0) automaton
 - $\text{Goto}(I, X)$ gives the transition from state I under input X

Example of Goto

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

Suppose $I = \{$

$E' \rightarrow E\bullet,$

$E \rightarrow E\bullet + T$

}

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

}

Canonical Collection of Sets of LR(0) Items

$C = \text{Closure}(\{S' \rightarrow \bullet S\})$

repeat

 for each set of items I in C

 for each grammar symbol X

 if $\text{Goto}(I, X)$ is not empty and not in C

 add $\text{Goto}(I, X)$ to C

until no new sets of items are added to C

Canonical Collection of Sets of LR(0) Items

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

- Compute the canonical collection for the expression grammar

Canonical Collection of Sets of LR(0) Items

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

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

$$I_2 = \text{Goto}(I_0, T) = \{ E \rightarrow T \bullet, \\ T \rightarrow T \bullet * F \} \\ I_3 = \text{Goto}(I_0, F) = \{ T \rightarrow F \bullet \} \\ I_5 = \text{Goto}(I_0, \text{id}) = \{ F \rightarrow \text{id} \bullet \} \}$$

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

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

Canonical Collection of Sets of LR(0) Items

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

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

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

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

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

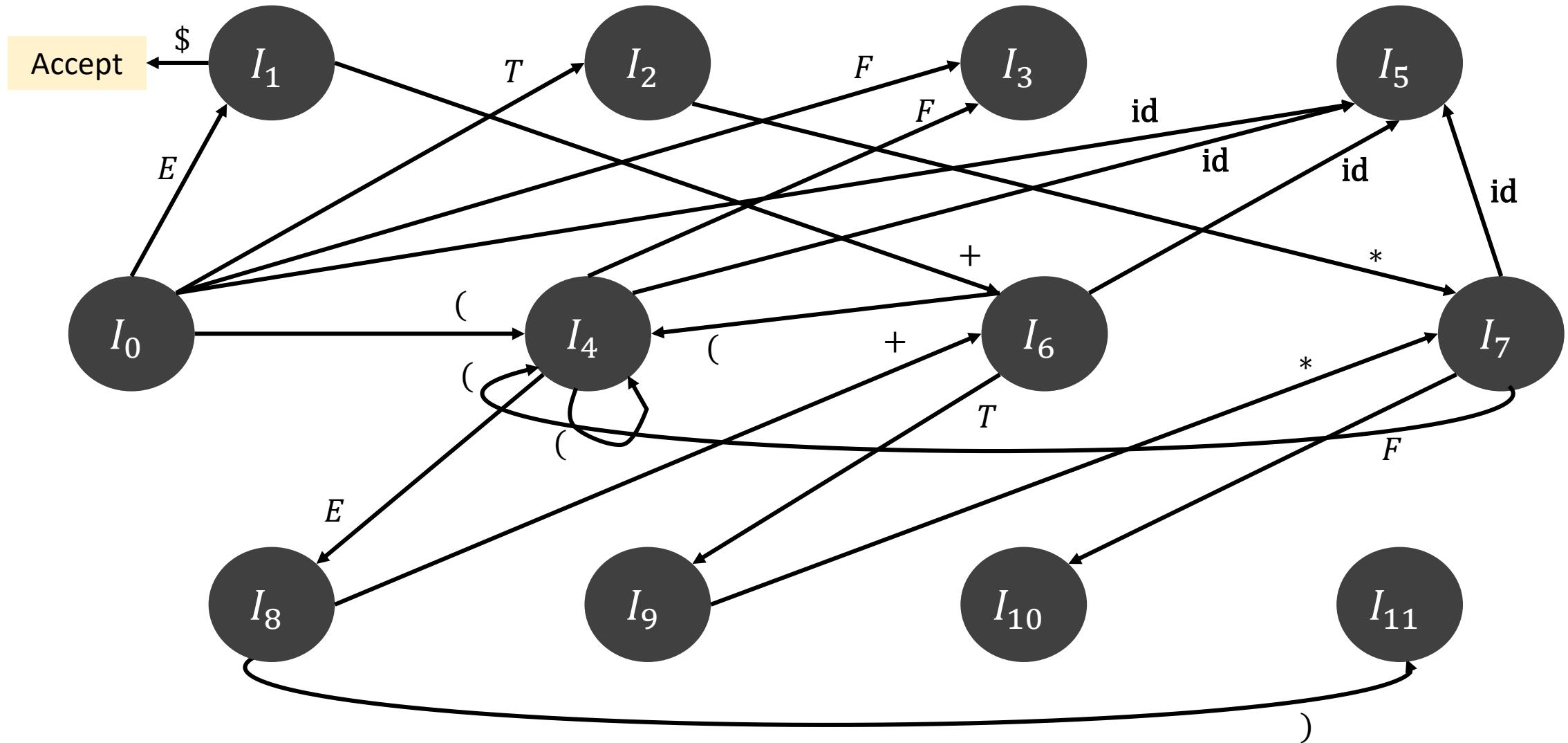
$$I_2 = \text{Goto}(I_4, T) \\ I_3 = \text{Goto}(I_4, F) \\ I_4 = \text{Goto}(I_4, "(") \\ I_5 = \text{Goto}(I_4, \text{id}) \\ I_3 = \text{Goto}(I_6, F) \\ I_4 = \text{Goto}(I_6, "(") \\ I_5 = \text{Goto}(I_6, \text{id}) \\ I_4 = \text{Goto}(I_7, "(") \\ I_5 = \text{Goto}(I_7, \text{id}) \\ I_6 = \text{Goto}(I_8,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

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 $\text{Closure}(\{S' \rightarrow \bullet S\})$, where S' is the start symbol of the augmented grammar
 - State j refers to the state corresponding to the set of items I_j

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, $\text{ACTION}[S_i, a_i]$ 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 $\text{Goto}(I_i, a) = I_j$, then set $\text{ACTION}[i, a] = \text{"Shift } j"$
 - b) If $[A \rightarrow \alpha \bullet]$ is in I_i , then set Action $[i, a] = \text{"Reduce } A \rightarrow \alpha"$ for all a
 - c) If $[S' \rightarrow S \bullet]$ is in I_i , then set Action $[i, \$] = \text{"Accept"}$
- 3) If $\text{Goto}(I_i, A) = I_j$, then $\text{GOTO}[i, A] = j$
- 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	\$	$\text{id} * \text{id\$}$	Shift
0 5	\$id	$* \text{id\$}$	Reduce by $F \rightarrow \text{id}$
0 3	\$F	$* \text{id\$}$	Reduce by $T \rightarrow F$
0 2	\$T	$* \text{id\$}$	Shift
0 2 7	\$T *	$\text{id\$}$	Shift
0 2 7 5	\$T * id	\$	Reduce by $F \rightarrow \text{id}$
0 2 7 10	\$T * F	\$	Reduce by $T \rightarrow T * F$
0 2	\$T	\$	Reduce by $E \rightarrow T$
0 1	\$E	\$	Accept

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$	\$

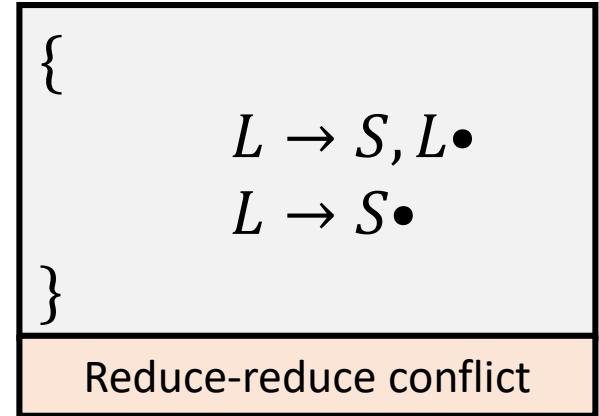
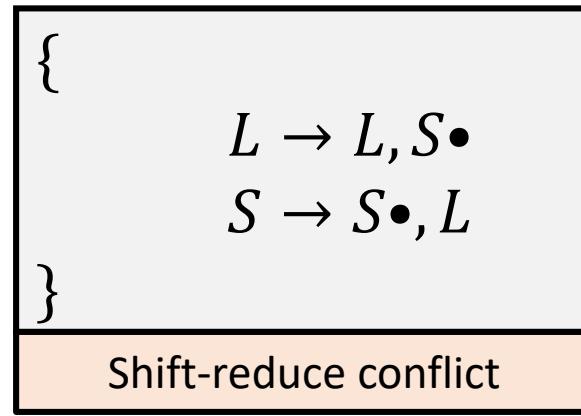
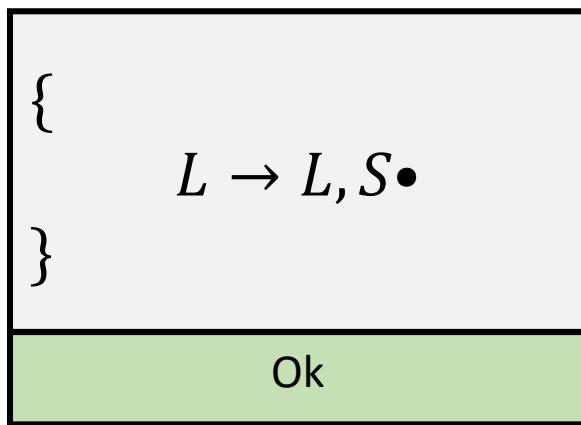
$X_1X_2X_3$ can never appear on the stack

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



- 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

$$\begin{aligned} S &\rightarrow S + E \mid E \\ E &\rightarrow \text{num} \end{aligned}$$

Left associative

$$\begin{aligned} S &\rightarrow E + S \mid E \\ E &\rightarrow \text{num} \end{aligned}$$

Right associative

Not
LR(0)

$$\begin{aligned} S &\rightarrow E^\bullet + S \\ S &\rightarrow E^\bullet \end{aligned}$$

Shift-reduce conflict

Canonical Collection of Sets of LR(0) Items

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

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

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

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

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

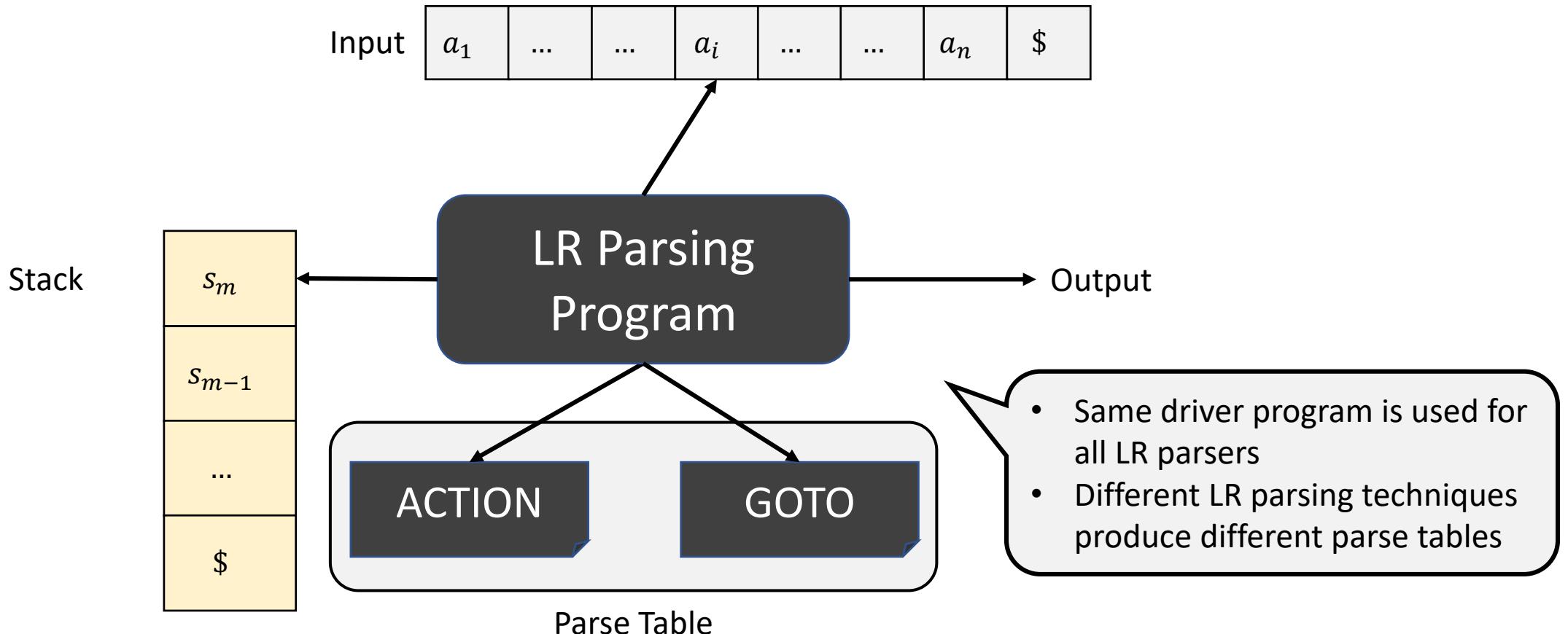
$I_2 = \text{Goto}(I_0, E) = \{$
 $S \rightarrow E \bullet + S,$
 $S \rightarrow E \bullet$
}

Not
LR(0)

Simple LR Parsing

SLR(1)

Block Diagram of LR Parser



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** $c \in \text{FOLLOW}(A)$ or $c = \epsilon$ and $S \xrightarrow{*} \gamma A$

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 $\text{Goto}(I_i, a) = I_j$, then set $\text{ACTION}[i, a] = \text{"Shift } j"$
 - b) If $[A \rightarrow \alpha \bullet]$ is in I_i , then set $\text{ACTION}[i, a] = \text{"Reduce } A \rightarrow \alpha"$ **for all a in $\text{FOLLOW}(A)$**
 - c) If $[S' \rightarrow S \bullet]$ is in I_i , then set $\text{Action}[i, \$] = \text{"Accept"}$
- 3) If $\text{Goto}(I_i, A) = I_j$, then $\text{GOTO}[i, A] = j$
- 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	$F \rightarrow \text{id}$

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

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

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

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

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

Canonical Collection of Sets of LR(0) Items

$$I_0 = \text{Closure}(\{E' \rightarrow \bullet E\}) = \{ \begin{array}{l} E' \rightarrow \bullet E, \\ E \rightarrow \bullet E + T, \\ E \rightarrow \bullet T, \\ T \rightarrow \bullet T * F, \\ T \rightarrow \bullet F, \\ F \rightarrow \bullet(E), \\ F \rightarrow \bullet \text{id}, \end{array} \}$$

$$I_2 = \text{Goto}(I_0, T) = \{ \begin{array}{l} E \rightarrow T \bullet, \\ T \rightarrow T \bullet * F \end{array} \}$$
$$I_3 = \text{Goto}(I_0, F) = \{ \begin{array}{l} T \rightarrow F \bullet \end{array} \}$$

$$I_5 = \text{Goto}(I_0, \text{id}) = \{ \begin{array}{l} F \rightarrow \text{id} \bullet \end{array} \}$$

$$I_4 = \text{Goto}(I_0, "(") = \{ \begin{array}{l} F \rightarrow (\bullet E), \\ E \rightarrow \bullet E + T, \\ E \rightarrow \bullet T, \\ T \rightarrow \bullet T * F, \\ T \rightarrow \bullet F, \\ F \rightarrow \bullet(E), \\ F \rightarrow \bullet \text{id}, \end{array} \}$$

$$I_7 = \text{Goto}(I_2, *) = \{ \begin{array}{l} T \rightarrow T * \bullet F, \\ F \rightarrow \bullet(E), \\ F \rightarrow \bullet \text{id} \end{array} \}$$

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

Canonical Collection of Sets of LR(0) Items

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

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

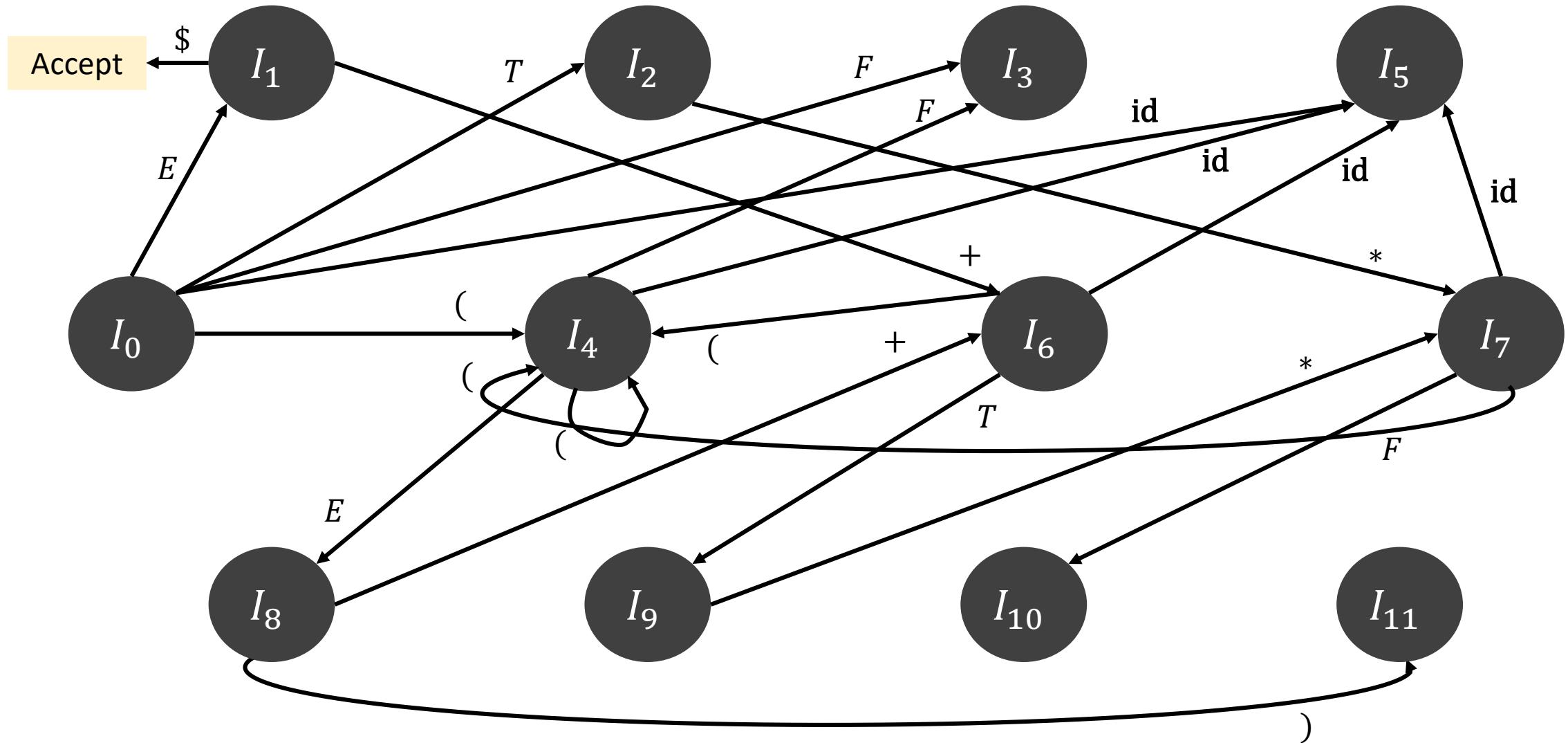
$$I_9 = \text{Goto}(I_6, T) = \{ \\ E \rightarrow E + T \bullet, \\ T \rightarrow T \bullet * F \\ \}$$

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

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

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

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 $\langle S_0, S_1, \dots, S_m, a_i a_{i+1} \dots a_n \$ \rangle$
 - Left half is stack content, and right half is the remaining input
- Configuration represents the right sentential form $X_1 X_2 \dots X_m a_i a_{i+1} \dots a_n$

LR Parsing Algorithm

- If $\text{ACTION}[s_m, a_i] = \text{shift } s$, new configuration is $\langle s_0, s_1, \dots, s_m s, a_{i+1} \dots a_n \$ \rangle$
- If $\text{ACTION}[s_m, a_i] = \text{reduce } A \rightarrow \beta$, new configuration is $\langle s_0, s_1, \dots, s_{m-r}, a_i a_{i+1} \dots a_n \$ \rangle$, where $r = |\beta|$ and $s = \text{GOTO}[s_{m-r}, A]$
- If $\text{ACTION}[s_m, a_i] = \text{accept}$, parsing is successful
- If $\text{ACTION}[s_m, a_i] = \text{error}$, 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  $A \rightarrow \beta$ 
        pop  $|\beta|$  symbols off the stack
        push GOTO[ $t, A$ ] onto the stack
        output production  $A \rightarrow \beta$ 
    else if ACTION[ $s, a$ ] == accept
        break
    else
        invoke error recovery
```

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

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

$$\begin{aligned} S &\rightarrow L = R \mid R \\ L &\rightarrow *R \mid \text{id} \\ R &\rightarrow L \end{aligned}$$

Example Derivation

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

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

$$\begin{aligned} \text{FOLLOW}(S) &= \text{FOLLOW}(L) = \text{FOLLOW}(R) \\ &= \{=, \$\} \end{aligned}$$

Canonical LR(0) Collection

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

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

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

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

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

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

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

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

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

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

SLR Parsing Table

State	ACTION				GOTO		
	=	*	id	\$	S	L	R
0		s_4	s_5		1	2	3
1				acc			
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

$$I_0 = \text{Closure}(S' \rightarrow .S) = \{$$
$$S' \rightarrow \bullet S,$$
$$S \rightarrow \bullet L = R,$$
$$S \rightarrow \bullet R$$

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

$$I_5 = \text{Goto}(I_0, \text{id}) = \{$$
$$L \rightarrow \bullet \text{id}$$
$$\}$$
$$L \rightarrow \text{Goto}(I_0, R) = \{$$

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

$$I_1 =$$
$$\{$$
$$S' \rightarrow S \bullet$$
$$\}$$
$$I_2 = \text{Goto}(I_0, L) = \{$$
$$\textcolor{red}{S \rightarrow L \bullet = R,}$$
$$\textcolor{red}{R \rightarrow L \bullet}$$
$$\}$$

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

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

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

Stack	Input	Action	Stack	Input	Action
0	$\text{id}=\text{id}\$$	Shift 5	0	$\text{id}=\text{id}\$$	Shift 5
$0 \text{id} 5$	$=\text{id}\$$	Reduce by $L \rightarrow \text{id}$	$0 \text{id} 5$	$=\text{id}\$$	Reduce by $L \rightarrow \text{id}$
$0 L 2$	$=\text{id}\$$	Reduce by $R \rightarrow L$	$0 L 2$	$=\text{id}\$$	Shift 6
$0 R 3$	$=\text{id}\$$	Error	$0 L 2 = 6$	$\text{id}\$$	Shift 5
 No right sentential form begins with $R = \dots$			$0 L 2 = 6 \text{id} 5$	\$	Reduce by $L \rightarrow \text{id}$
			$0 L 2 = 6 L 8$	\$	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 $\text{id}=\text{id}$

Stack	Input	Action	Stack	Input	Action
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 $A \rightarrow \alpha$					

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

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

Stack	Input	Action	Stack	Input	Action
0	$\text{id}=\text{id}\$$	Shift 5	0	$\text{id}=\text{id}\$$	Shift 5

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

Stack	Input	Action
$0 L 2 = 6 L 8$	\$	Reduce by $R \rightarrow L$
$0 L 2 = 6 R 9$	\$	Reduce by $S \rightarrow L = R$
$0 S 1$	\$	Accept

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 $\alpha \in T \cup \{\$\}$
- 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 $\text{FOLLOW}(A)$, but can be a proper subset

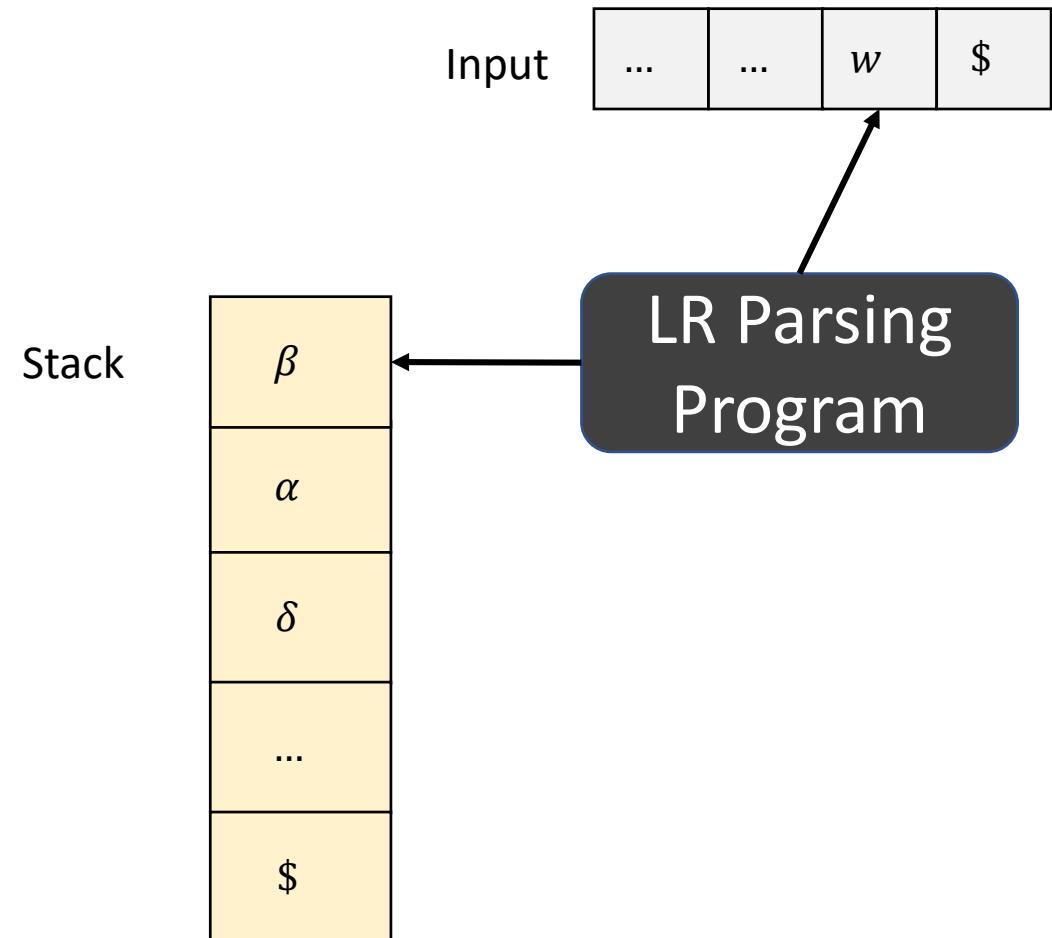
LR(1) Item

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

$$S \xrightarrow[rm]^* \delta A w \xrightarrow[rm]{} \delta \alpha \beta w$$

where

- $\gamma = \delta \alpha$, and
- a is the first symbol in w , or,
 $w = \epsilon$ and $a = \$$



Constructing LR(1) Sets of Items

Closure(I)

```
repeat
    for each item  $[A \rightarrow \alpha \bullet B\beta, a]$  in  $I$ 
        for each production  $B \rightarrow \gamma$  in  $G'$ 
            for each terminal  $b$  in 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)

```
initialize  $J$  to be the empty set
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

Items(G'):

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

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

generates the regular language c^*dc^*d

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

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

Example Construction of LR(1) Items

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

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

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

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

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

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

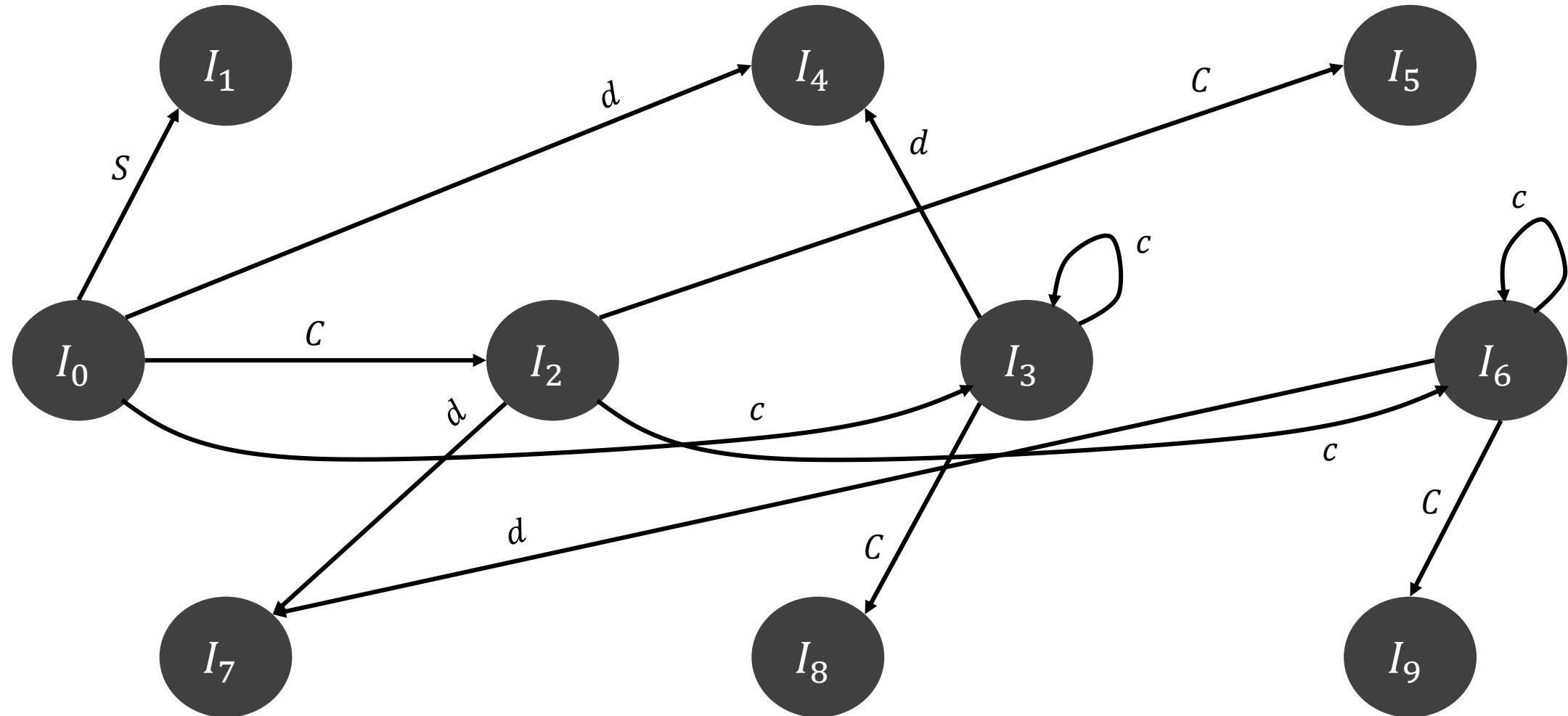
$$I_6 = \text{Goto}(I_2, c) = \{ C \rightarrow c \bullet C, \$, \\ C \rightarrow \bullet c C, \$, \\ C \rightarrow \bullet d, \$ \}$$

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

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

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

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

State	ACTION			GOTO	
	c	d	\$	S	C
0	s_3	s_4		1	2
1			acc		
2	s_6	s_7			5
3	s_3	s_4			8
4	r_3	r_3			
5			r_1		
6	s_6	s_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

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 $C \rightarrow d$
4	0 3 8	cC	cd\$	Reduce by $C \rightarrow cC$
5	0 2	C	cd\$	Shift
6	0 2 6	Cc	d\$	Shift
7	0 2 6 7	Ccd	\$	Reduce by $C \rightarrow d$
8	0 2 6 9	CcC	\$	Reduce by $C \rightarrow cC$
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

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

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

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

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

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

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

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

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

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

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

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 $J = I_1 \cup I_2 \cup \dots \cup I_k$, where the cores of I_1, I_2, \dots, I_k are same
 - Cores of $\text{Goto}(I_1, X), \text{Goto}(I_2, X), \dots, \text{Goto}(I_k, X)$ will also be the same
 - Let $K = \text{Goto}(I_1, X) \cup \text{Goto}(I_2, X) \cup \dots \cup \text{Goto}(I_k, X)$, then $\text{Goto}(J, X) = K$

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	$C \rightarrow d$

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

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

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

LALR Parsing Table

State	ACTION			GOTO	
	<i>c</i>	<i>d</i>	\$	<i>S</i>	<i>C</i>
0	<i>s</i> 36	<i>s</i> 47		1	2
1			<i>acc</i>		
2	<i>s</i> 36	<i>s</i> 47			5
36	<i>s</i> 36	<i>s</i> 47			89
47	<i>r</i> 3	<i>r</i> 3	<i>r</i> 3		
5			<i>r</i> 1		
89	<i>r</i> 2	<i>r</i> 2	<i>r</i> 2		

Moves of a LALR Parser on **cdcd**

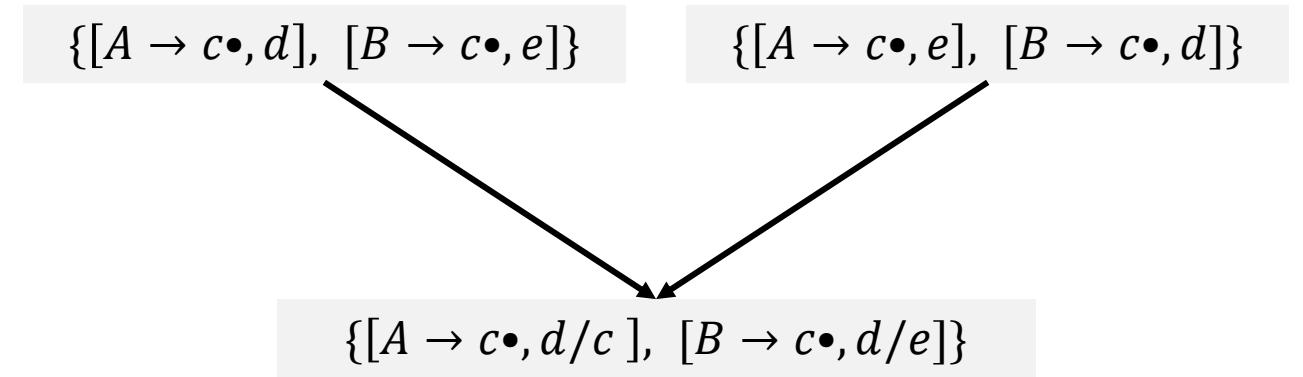
	Stack	Symbols	Input	Action
1	0		cdcd\$	Shift
2	0 36	c	dcd\$	Shift
3	0 36 47	cd	cd\$	Reduce by $C \rightarrow d$
4	0 36 89	cC	cd\$	Reduce by $C \rightarrow cC$
5	0 2	C	cd\$	Shift
6	0 2 36	Cc	d\$	Shift
7	0 2 36 47	Ccd	\$	Reduce by $C \rightarrow d$
8	0 2 36 89	CcC	\$	Reduce by $C \rightarrow cC$
9	0 2 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



Dealing with Errors with LALR Parsing

- Consider an erroneous input **ccd**

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

CLR Parsing Table					
State	Action			Goto	
	<i>c</i>	<i>d</i>	\$	<i>S</i>	<i>C</i>
0	<i>s3</i>	<i>s4</i>		1	2
1			<i>acc</i>		
2	<i>s6</i>	<i>s7</i>			5
3	<i>s3</i>	<i>s4</i>			8
4	<i>r3</i>	<i>r3</i>			
5			<i>r1</i>		
6	<i>s6</i>	<i>s7</i>			9
7			<i>r3</i>		
8	<i>r2</i>	<i>r2</i>			
9			<i>r2</i>		

LALR Parsing Table					
State	Action			Goto	
	<i>c</i>	<i>d</i>	\$	<i>S</i>	<i>C</i>
0	<i>s36</i>	<i>s47</i>		1	2
1			<i>acc</i>		
2	<i>s36</i>	<i>s47</i>			5
36	<i>s36</i>	<i>s47</i>			89
47	<i>r3</i>	<i>r3</i>	<i>r3</i>		
5			<i>r1</i>		
89	<i>r2</i>	<i>r2</i>	<i>r2</i>		

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 $C \rightarrow d$
0 36 36 89	ccC	\$	Reduce by $C \rightarrow cC$
0 36 89	cC	\$	Reduce by $C \rightarrow cC$
0 2	C	\$	Error

Comparing Moves of CLR and LALR Parsers

- Consider an erroneous input ccd

CLR Parsing Table	LALR Parsing Table												
<ul style="list-style-type: none">CLR parser will not even reduce before reporting an errorSLR and LALR parsers may reduce several times before reporting an error, but will never shift an erroneous input symbol onto the stack	<table border="1"><thead><tr><th>0 . 5 0 5 0 8 9</th><th>ccC</th><th>\$</th><th>Reduce by $C \rightarrow cc$</th></tr></thead><tbody><tr><td>0 3 6 8 9</td><td>cC</td><td>\$</td><td>Reduce by $C \rightarrow cC$</td></tr><tr><td>0 2</td><td>C</td><td>\$</td><td>Error</td></tr></tbody></table>	0 . 5 0 5 0 8 9	ccC	\$	Reduce by $C \rightarrow cc$	0 3 6 8 9	cC	\$	Reduce by $C \rightarrow cC$	0 2	C	\$	Error
0 . 5 0 5 0 8 9	ccC	\$	Reduce by $C \rightarrow cc$										
0 3 6 8 9	cC	\$	Reduce by $C \rightarrow cC$										
0 2	C	\$	Error										

Using Ambiguous Grammars

Dealing with Ambiguous Grammars

$$\begin{aligned} E' &\rightarrow E \\ E &\rightarrow E + E \mid E * E \mid (E) \mid \text{id} \end{aligned}$$

$$I_0 = \text{Closure}(\{E' \rightarrow \bullet E\}) = \{ \begin{aligned} E' &\rightarrow \bullet E, \\ E &\rightarrow \bullet E + E, \\ E &\rightarrow \bullet E * E, \\ E &\rightarrow \bullet(E), \\ E &\rightarrow \bullet \text{id} \end{aligned} \}$$

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

Does not specify the associativity and precedence of the two operators

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

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

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

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

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

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

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

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

SLR(1) Parsing Table

State	id	ACTION						GOTO
		+	*	()	\$	E	
0	s_3				s_2			1
1		s_4	s_5				acc	
2	s_3				s_2			6
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	

Moves of an SLR Parser on $\text{id} + \text{id} * \text{id}$

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

What can the parser do to resolve the ambiguity?

SLR(1) Parsing Table

State	Action							Goto
	id	+	*	()	\$	E	
0	s_3				s_2			1
1		s_4	s_5				acc	
2	s_3				s_2			6
3		r_4	r_4			r_4	r_4	
6		s_4	s_5		s_2			7
7		s_4, r_1	s_5, r_1		s_2			8
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?

Summary

Comparison across LR Parsing Techniques

- $\text{SLR}(1) = \text{LR}(0) \text{ items} + \text{FOLLOW}$
 - 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
- $\text{SLR}(1) \leq \text{LALR}(1) \leq \text{LR}(1)$
- $\text{SLR}(k) \leq \text{LALR}(k) \leq \text{LR}(k)$
- $\text{LL}(k) \leq \text{LR}(k)$
- 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.