

# CSE 309 (Compilers)

## Syntax Analysis

Dr. Muhammad Masroor Ali

Professor

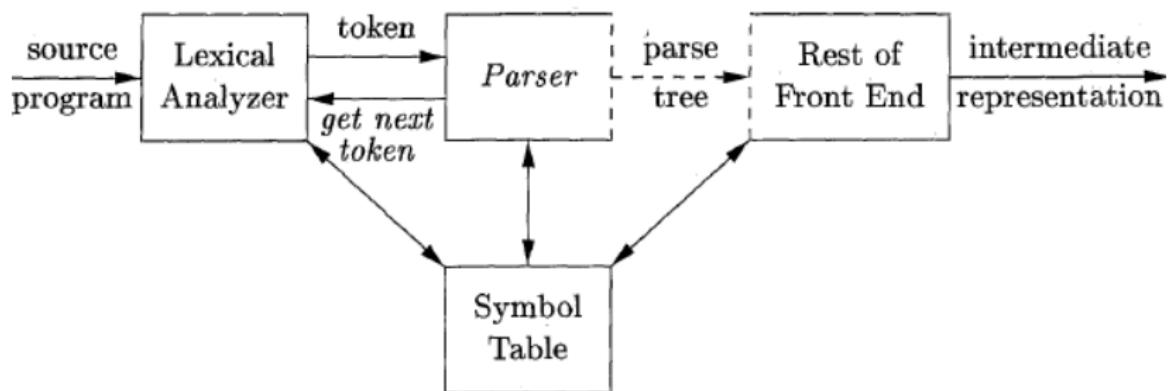
Department of Computer Science and Engineering  
Bangladesh University of Engineering and Technology  
Dhaka-1205, Bangladesh

January 2025

*Version: 1.1, Last modified: April 28, 2025*

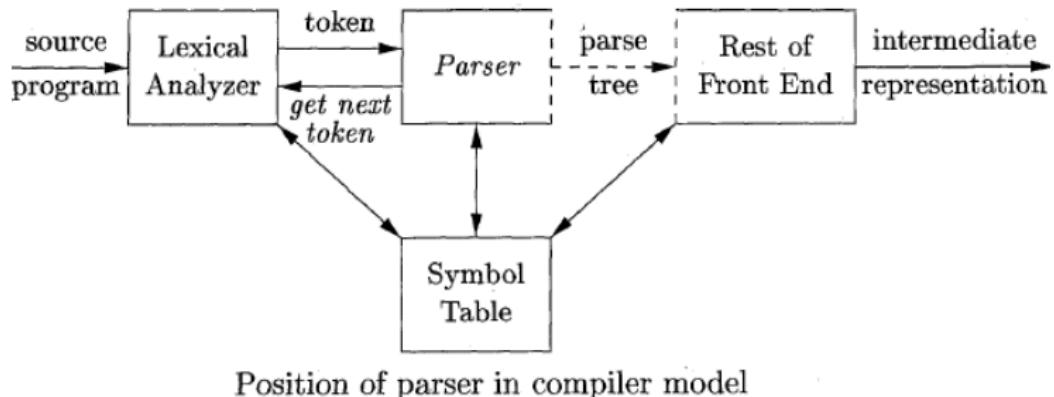
# The Role of the Parser

- In our compiler model, the parser obtains a string of tokens from the lexical analyzer.
- It then verifies that the string of token names can be generated by the grammar for the source language.



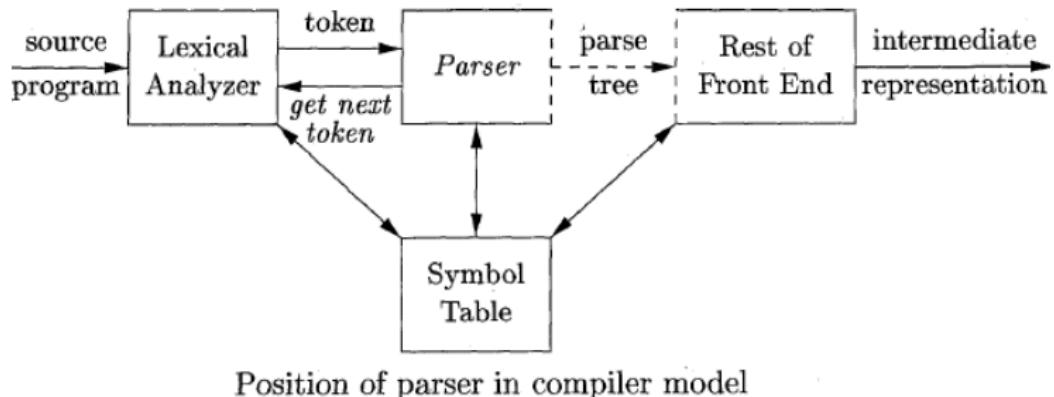
Position of parser in compiler model

# The Role of the Parser — *continued*



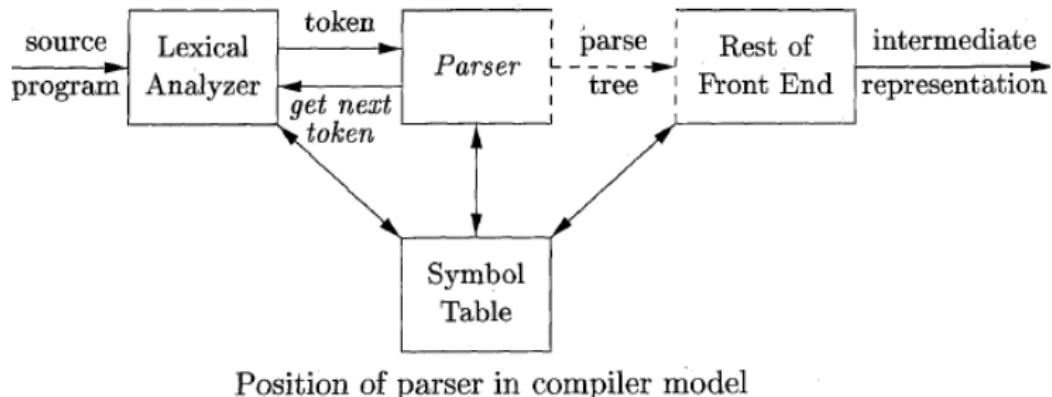
- We expect the parser
  - to report any syntax errors in an intelligible fashion and
  - to recover from commonly occurring errors to continue processing the remainder of the program.

## The Role of the Parser — *continued*



- Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.
- In fact, the parse tree need not be constructed explicitly.
- Since checking and translation actions can be interspersed with parsing.

## The Role of the Parser — *continued*



- Thus, the parser and the rest of the front end could well be implemented by a single module.

## The Role of the Parser — *continued*

- There are three general types of parsers for grammars:
  - universal,
  - top-down, and
  - bottom-up.
- Universal parsing methods such as the Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar.
- These general methods are, however, too inefficient to use in production compilers.

## The Role of the Parser — *continued*

- The methods commonly used in compilers can be classified as being either top-down or bottom-up.
- As implied by their names, top-down methods build parse trees from the top (root) to the bottom (leaves).
- Bottom-up methods start from the leaves and work their way up to the root.
- In either case, the input to the parser is scanned from left to right, one symbol at a time.

## The Role of the Parser — *continued*

- The most efficient top-down and bottom-up methods work only for subclasses of grammars.
- But several of these classes, particularly, LL and LR grammars, are expressive enough to describe most of the syntactic constructs in modern programming languages.
- Parsers implemented by hand often use LL grammars.
- The predictive-parsing approach works for LL grammars.
- Parsers for the larger class of LR grammars are usually constructed using automated tools.

# Representative Grammars

- Some of the grammars that will be examined are presented here for ease of reference.
- Constructs that begin with keywords like **while** or **int**, are relatively easy to parse.
- The keyword guides the choice of the grammar production that must be applied to match the input.
- We therefore concentrate on expressions, which present more of challenge, because of the associativity and precedence of operators.

# Representative Grammars

- Some of the grammars that will be examined are presented here for ease of reference.
- Constructs that begin with keywords like **while** or **int**, are relatively easy to parse.
- The keyword guides the choice of the grammar production that must be applied to match the input.
- We therefore concentrate on expressions, which present more of challenge, because of the associativity and precedence of operators.

## Representative Grammars — *continued*

- Associativity and precedence are captured in the following grammar.
- $E$  represents expressions consisting of terms separated by + signs.
- $T$  represents terms consisting of factors separated by \* signs.
- $F$  represents factors that can be either parenthesized expressions or identifiers:

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

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

## Representative Grammars — *continued*

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

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

- The above grammar belongs to the class of *LR* grammars that are suitable for bottom-up parsing.
- This grammar can be adapted to handle additional operators and additional levels of precedence.
- However, it cannot be used for top-down parsing because it is left recursive.

# Representative Grammars — *continued*

- The following non-left-recursive variant of the expression grammar will be used for top-down parsing:

$$\begin{array}{l} E \rightarrow TE' \\ E' \rightarrow +TE' \mid \epsilon \\ T \rightarrow FT' \\ T' \rightarrow *FT' \mid \epsilon \\ F \rightarrow ( E ) \mid \mathbf{id} \end{array}$$

## Representative Grammars — *continued*

- The following grammar treats  $+$  and  $*$  alike.

$$E \rightarrow E + E \quad | \quad E * E \quad | \quad ( E ) \quad | \quad \mathbf{id}$$

- So it is useful for illustrating techniques for handling ambiguities during parsing.
- Here,  $E$  represents expressions of all types.
- This grammar permits more than one parse tree for expressions like  $a + b * c$ .

# Syntax Error Handling

- If a compiler had to process only correct programs, its design and implementation would be simplified greatly.
- However, a compiler is expected to assist the programmer in locating and tracking down errors that inevitably creep into programs, despite the programmer's best efforts.
- Strikingly, few languages have been designed with error handling in mind, even though errors are so commonplace.

## Syntax Error Handling — *continued*

- Our civilization would be radically different if spoken languages had the same requirements for syntactic accuracy as computer languages.
- Most programming language specifications do not describe how a compiler should respond to errors.
- Error handling is left to the compiler designer.
- Planning the error handling right from the start can both simplify the structure of a compiler and improve its handling of errors.

## Syntax Error Handling — *continued*

Common programming errors can occur at many different levels.

Lexical errors include misspellings of identifiers, keywords, or operators — e.g., the use of an identifier `elipsesize` instead of `ellipsesize` — and missing quotes around text intended as a string.

# Syntax Error Handling — *continued*

Common programming errors can occur at many different levels.

Syntactic errors include misplaced semicolons or extra or missing braces, that is, “{” or “}”.

As another example, in C or Java, the appearance of a case statement without an enclosing switch is a syntactic error.

However, this situation is usually allowed by the parser and caught later in the processing, as the compiler attempts to generate code.

## Syntax Error Handling — *continued*

Common programming errors can occur at many different levels.

Semantic errors include type mismatches between operators and operands.

An example is a `return` statement in a Java method with result type `void`.

## Syntax Error Handling — *continued*

Common programming errors can occur at many different levels.

**Logical errors** can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator `=` instead of the comparison operator `==`.

The program containing `=` may be well formed; however, it may not reflect the programmer's intent.

## Syntax Error Handling — *continued*

- The precision of parsing methods allows syntactic errors to be detected very efficiently.
- Several parsing methods, such as the LL and LR methods, detect an error as soon as possible.
- That is, when the stream of tokens from the lexical analyzer cannot be parsed further according to the grammar for the language.
- More precisely, they have the viable-prefix property, meaning that they detect that an error has occurred as soon as they see a prefix of the input that cannot be completed to form a string in the language.

## Syntax Error Handling — *continued*

- Another reason for emphasizing error recovery during parsing is that many errors appear syntactic, whatever their cause, and are exposed when parsing cannot continue.
- A few semantic errors, such as type mismatches, can also be detected efficiently.
- However, accurate detection of semantic and logical errors at compile time is in general a difficult task.

## Syntax Error Handling — *continued*

- The error handler in a parser has goals that are simple to state but challenging to realize:
  - Report the presence of errors clearly and accurately.
  - Recover from each error quickly enough to detect subsequent errors.
  - Add minimal overhead to the processing of correct programs.

## Syntax Error Handling — *continued*

- Fortunately, common errors are simple ones.
- A relatively straightforward error-handling mechanism often suffices.
- How should an error handler report the presence of an error?
- At the very least, it must report the place in the source program where an error is detected.
- There is a good chance that the actual error occurred within the previous few tokens.
- A common strategy is to print the offending line with a pointer to the position at which an error is detected.

# Writing a Grammar

- Grammars are capable of describing most, but not all, of the syntax of programming languages.
- For instance, the requirement that identifiers be declared before they are used, cannot be described by a context-free grammar.
- Therefore, the sequences of tokens accepted by a parser form a superset of the programming language.
- Subsequent phases of the compiler must analyze the output of the parser to ensure compliance with rules that are not checked by the parser.

# Writing a Grammar

- Grammars are capable of describing most, but not all, of the syntax of programming languages.
- For instance, the requirement that identifiers be declared before they are used, cannot be described by a context-free grammar.
- Therefore, the sequences of tokens accepted by a parser form a superset of the programming language.
- Subsequent phases of the compiler must analyze the output of the parser to ensure compliance with rules that are not checked by the parser.

- A grammar is left recursive if it has a nonterminal  $A$  such that there is a derivation  $A \xrightarrow{+} Aa$  for some string  $a$ .
- Top-down parsing methods cannot handle left-recursive grammars, so a transformation that eliminates left recursion is needed.
- In simple left recursion there was one production of the form  $A \rightarrow A\alpha$ .
- Here we study the general case.

## Elimination of Left Recursion — *continued*

- Left-recursive pair of productions  $A \rightarrow A\alpha \mid \beta$  can be replaced by the non-left-recursive productions

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \epsilon$$

without changing the set of strings derivable from  $A$ .

- This rule by itself suffices in many grammars.

# Example

$$A \rightarrow A\alpha \mid \beta$$

to be replaced by

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \epsilon$$

- Grammar for arithmetic expressions,

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

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

- Eliminating the immediate left recursions we obtain,

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

## Elimination of Left Recursion — *continued*

- No matter how many  $A$ -productions there are, we can eliminate immediate left recursion from them.
- First, we group the  $A$ -productions as,

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \dots \mid \beta_n$$

where no  $\beta_i$ , begins with an  $A$ .

- Then, we replace the  $A$ -productions by,

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \dots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \dots \mid \alpha_m A' \mid \epsilon$$

- It does not eliminate left recursion involving derivations of two or more steps.

## Elimination of Left Recursion — *continued*

- It does not eliminate left recursion involving derivations of two or more steps.
- Consider the grammar,

$$\begin{array}{l} S \rightarrow Aa \mid b \\ A \rightarrow Ac \mid Sd \mid \epsilon \end{array}$$

- The nonterminal  $S$  is left-recursive because  $S \Rightarrow Aa \Rightarrow Sda$ , but it is not immediately left recursive.

# Algorithm

Eliminating left recursion.

**INPUT:** Grammar  $G$  with no cycles or  $\epsilon$ -productions.

**OUTPUT:** An equivalent grammar with no left recursion.

**METHOD:** Apply the algorithm to  $G$ . Note that the resulting non-left-recursive grammar may have  $\epsilon$ -productions.

- 1) arrange the nonterminals in some order  $A_1, A_2, \dots, A_n$ .
- 2) **for** ( each  $i$  from 1 to  $n$  ) {
- 3)   **for** (each  $j$  from 1 to  $i - 1$  ) {
- 4)     replace each production of the form  $A_i \rightarrow A_j \gamma$  by  
          the productions  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$   
          where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all the  
          current  $A_j$ -productions
- 5)    }
- 6)    eliminate the immediate left recursion among  
        the  $A_i$ -productions;
- 7) }

Grammar with cycles: Grammar where derivations of the form  
 $A \xrightarrow{+} A$  occurs.

- 1) arrange the nonterminals in some order  $A_1, A_2, \dots, A_n$ .
- 2) **for** ( each  $i$  from 1 to  $n$  ) {
- 3)   **for** (each  $j$  from 1 to  $i - 1$  ) {
- 4)     replace each production of the form  $A_i \rightarrow A_j \gamma$  by  
          the productions  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$   
          where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all the  
          current  $A_j$ -productions
- 5)     }
- 6)    eliminate the immediate left recursion among  
        the  $A_i$ -productions;
- 7) }

- In the first iteration for  $i = 1$ , the outer **for**-loop of lines (2) through (7) eliminates any immediate left recursion among  $A_1$ -productions.

- 1) arrange the nonterminals in some order  $A_1, A_2, \dots, A_n$ .
- 2) **for** ( each  $i$  from 1 to  $n$  ) {
- 3)   **for** (each  $j$  from 1 to  $i - 1$  ) {
- 4)     replace each production of the form  $A_i \rightarrow A_j \gamma$  by  
          the productions  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$   
          where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all the  
          current  $A_j$ -productions
- 5)     }
- 6)    eliminate the immediate left recursion among  
        the  $A_i$ -productions;
- 7) }

- Any remaining  $A_1$  productions of the form  $A_1 \rightarrow A_l \alpha$  must therefore have  $l > 1$ .

- 1) arrange the nonterminals in some order  $A_1, A_2, \dots, A_n$ .
- 2) **for** ( each  $i$  from 1 to  $n$  ) {
- 3)   **for** (each  $j$  from 1 to  $i - 1$  ) {
- 4)     replace each production of the form  $A_i \rightarrow A_j \gamma$  by  
          the productions  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$   
          where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all the  
          current  $A_j$ -productions
- 5)     }
- 6)    eliminate the immediate left recursion among  
        the  $A_i$ -productions;
- 7) }

- After the  $i - 1$ st iteration of the outer **for**- loop, all nonterminals  $A_k$ , where  $k < i$  , are “cleaned”.
- That is, any production  $A_k \rightarrow A_l \alpha$ , must have  $l > k$ .

- 1) arrange the nonterminals in some order  $A_1, A_2, \dots, A_n$ .
- 2) **for** ( each  $i$  from 1 to  $n$  ) {
- 3)   **for** (each  $j$  from 1 to  $i - 1$  ) {
- 4)     replace each production of the form  $A_i \rightarrow A_j \gamma$  by  
          the productions  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$   
          where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all the  
          current  $A_j$ -productions
- 5)     }
- 6)    eliminate the immediate left recursion among  
        the  $A_i$ -productions;
- 7) }

- As a result, on the  $i$ th iteration, the inner loop of lines (3) through (5) progressively raises the lower limit in any production  $A_i \rightarrow A_m \alpha$ , until we have  $m \geq i$ .

- 1) arrange the nonterminals in some order  $A_1, A_2, \dots, A_n$ .
- 2) **for** ( each  $i$  from 1 to  $n$  ) {
- 3)   **for** (each  $j$  from 1 to  $i - 1$  ) {
- 4)     replace each production of the form  $A_i \rightarrow A_j \gamma$  by  
          the productions  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$   
          where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all the  
          current  $A_j$ -productions
- 5)     }
- 6)    eliminate the immediate left recursion among  
        the  $A_i$ -productions;
- 7) }

- Then, eliminating immediate left recursion for the  $A_i$  productions at line (6) forces  $m$  to be greater than  $i$ .

# Example

Input Grammar  $G$  with no cycles or  $\epsilon$ -productions.

---

- We apply the procedure to grammar,

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Ac \mid Sd \mid \epsilon$$

- Technically, the algorithm is not guaranteed to work, because of the  $\epsilon$ -production.
- But in this case the production  $A \rightarrow \epsilon$  turns out to be harmless.

## Example — *continued*

- 1) arrange the nonterminals in some order  $A_1, A_2, \dots, A_n$ .

### Left-Recursive Grammar

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Ac \mid Sd \mid \epsilon$$

- We order the nonterminals  $S, A$ .
- $A_1 = S, A_2 = A$

## Example — *continued*

- 1) arrange the nonterminals in some order  $A_1, A_2, \dots, A_n$ .

### Left-Recursive Grammar

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Ac \mid Sd \mid \epsilon$$

- We order the nonterminals  $S, A$ .
- $A_1 = S, A_2 = A$

## Example — *continued*

- 1) arrange the nonterminals in some order  $A_1, A_2, \dots, A_n$ .
- 2) **for** ( each  $i$  from 1 to  $n$  ) {
- 3)   **for** (each  $j$  from 1 to  $i - 1$  ) {
- 4)       replace each production of the form  $A_i \rightarrow A_j \gamma$  by  
          the productions  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$   
          where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all the  
          current  $A_j$ -productions
- 5)     }
- 6)     eliminate the immediate left recursion among  
          the  $A_i$ -productions;
- 7) }

- $i = 1, A_1 = S$
- $j = 1$  to  $j = 1 - 1 = 0$ , the loop is *not* entered

## Example — *continued*

- 1) arrange the nonterminals in some order  $A_1, A_2, \dots, A_n$ .
- 2) **for** ( each  $i$  from 1 to  $n$  ) {
- 3)   **for** (each  $j$  from 1 to  $i - 1$  ) {
- 4)       replace each production of the form  $A_i \rightarrow A_j \gamma$  by  
          the productions  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$   
          where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all the  
          current  $A_j$ -productions
- 5)       }
- 6)       eliminate the immediate left recursion among  
          the  $A_i$ -productions;
- 7) }

- $i = 1, A_1 = S$
- $j = 1$  to  $j = 1 - 1 = 0$ , the loop is *not* entered

## Example — *continued*

- 6) eliminate the immediate left recursion among the  $A_i$ -productions;

### Left-Recursive Grammar

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Ac \mid Sd \mid \epsilon$$

- There is no immediate left recursion among the  $S$ -productions, so nothing happens for the case  $i = 1$ . ( $A_1 = S$ )

## Example — *continued*

- 1) arrange the nonterminals in some order  $A_1, A_2, \dots, A_n$ .
- 2) **for** ( each  $i$  from 1 to  $n$  ) {
- 3)   **for** (each  $j$  from 1 to  $i - 1$  ) {
- 4)       replace each production of the form  $A_i \rightarrow A_j \gamma$  by  
          the productions  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$   
          where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all the  
          current  $A_j$ -productions
- 5)     }
- 6)     eliminate the immediate left recursion among  
          the  $A_i$ -productions;
- 7) }

- $i = 2, A_2 = A$
- $j = 1$  to  $j = 2 - 1 = 1$ , the loop is entered

## Example — *continued*

- 1) arrange the nonterminals in some order  $A_1, A_2, \dots, A_n$ .
- 2) **for** ( each  $i$  from 1 to  $n$  ) {
- 3)   **for** (each  $j$  from 1 to  $i - 1$  ) {
- 4)       replace each production of the form  $A_i \rightarrow A_j \gamma$  by  
          the productions  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$   
          where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all the  
          current  $A_j$ -productions
- 5)     }
- 6)     eliminate the immediate left recursion among  
          the  $A_i$ -productions;
- 7) }

- $i = 2, A_2 = A$
- $j = 1$  to  $j = 2 - 1 = 1$ , the loop is entered

## Example — *continued*

- 4) replace each production of the form  $A_i \rightarrow A_j\gamma$  by the productions  $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \dots \mid \delta_k\gamma$  where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \delta_k$  are all the current  $A_j$ -productions

### Left-Recursive Grammar

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

- $i = 2, A_2 = A, j = 1, A_1 = S$
- We need to
  - put productions of the form  $S \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$
  - in productions of the form  $A \rightarrow S\gamma$
- Production(s) with  $S$  at the left-hand-side,  $S \rightarrow Aa \mid b$
- Productions(s) with  $A$  at the left side and right side beginning with  $S$  is (are),  $A \rightarrow Sd$

## Example — *continued*

- 4) replace each production of the form  $A_i \rightarrow A_j\gamma$  by the productions  $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \dots \mid \delta_k\gamma$  where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \delta_k$  are all the current  $A_j$ -productions

### Left-Recursive Grammar

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

- $i = 2, A_2 = A, j = 1, A_1 = S$
- We need to
  - put productions of the form  $S \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$
  - in productions of the form  $A \rightarrow S\gamma$
- Production(s) with  $S$  at the left-hand-side,  $S \rightarrow Aa \mid b$
- Productions(s) with  $A$  at the left side and right side beginning with  $S$  is (are),  $A \rightarrow Sd$

## Example — *continued*

- 4) replace each production of the form  $A_i \rightarrow A_j\gamma$  by the productions  $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \dots \mid \delta_k\gamma$  where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all the current  $A_j$ -productions

### Left-Recursive Grammar

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

- $i = 2, A_2 = A, j = 1, A_1 = S$
- We need to
  - put productions of the form  $S \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$
  - in productions of the form  $A \rightarrow S\gamma$
- Production(s) with  $S$  at the left-hand-side,  $S \rightarrow Aa \mid b$
- Productions(s) with  $A$  at the left side and right side beginning with  $S$  is (are),  $A \rightarrow Sd$

## Example — *continued*

- 4) replace each production of the form  $A_i \rightarrow A_j\gamma$  by the productions  $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \dots \mid \delta_k\gamma$  where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all the current  $A_j$ -productions

### Left-Recursive Grammar

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

- $i = 2, A_2 = A, j = 1, A_1 = S$
- We need to
  - put productions of the form  $S \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$
  - in productions of the form  $A \rightarrow S\gamma$
- Production(s) with  $S$  at the left-hand-side,  $S \rightarrow Aa \mid b$
- Productions(s) with  $A$  at the left side and right side beginning with  $S$  is (are),  $A \rightarrow Sd$

## Example — *continued*

- 4) replace each production of the form  $A_i \rightarrow A_j\gamma$  by the productions  $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \dots \mid \delta_k\gamma$  where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all the current  $A_j$ -productions

### Left-Recursive Grammar

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

- $i = 2, A_2 = A, j = 1, A_1 = S$
- We need to
  - put productions of the form  $S \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$
  - in productions of the form  $A \rightarrow S\gamma$
- Production(s) with  $S$  at the left-hand-side,  $S \rightarrow Aa \mid b$
- Productions(s) with  $A$  at the left side and right side beginning with  $S$  is (are),  $A \rightarrow Sd$

## Example — *continued*

- 4) replace each production of the form  $A_i \rightarrow A_j\gamma$  by the productions  $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \dots \mid \delta_k\gamma$  where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all the current  $A_j$ -productions

### Left-Recursive Grammar

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

- $i = 2, A_2 = A, j = 1, A_1 = S$
- We need to
  - put productions of the form  $S \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$
  - in productions of the form  $A \rightarrow S\gamma$
- Production(s) with  $S$  at the left-hand-side,  $S \rightarrow Aa \mid b$
- Productions(s) with  $A$  at the left side and right side beginning with  $S$  is (are),  $A \rightarrow Sd$

## Example — *continued*

- 4) replace each production of the form  $A_i \rightarrow A_i\gamma$  by the productions  $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \dots \mid \delta_k\gamma$  where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \delta_k$  are all the current  $A_j$ -productions

### Left-Recursive Grammar

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

- $S \rightarrow Aa \mid b$  to be put in  $A$ ,  $A \rightarrow Sd$
- We substitute  $S \rightarrow Aa \mid b$  in  $A \rightarrow Sd$  to get the following  $A$ -productions,

$$A \rightarrow Aad \mid bd$$

## Example — *continued*

- 4) replace each production of the form  $A_i \rightarrow A_i\gamma$  by the productions  $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \dots \mid \delta_k\gamma$  where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \delta_k$  are all the current  $A_j$ -productions

### Left-Recursive Grammar

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

- $S \rightarrow Aa \mid b$  to be put in  $A$ ,  $A \rightarrow Sd$
- We substitute  $S \rightarrow Aa \mid b$  in  $A \rightarrow Sd$  to get the following  $A$ -productions,

$$A \rightarrow Aad \mid bd$$

## Example — *continued*

- 1) arrange the nonterminals in some order  $A_1, A_2, \dots, A_n$ .
- 2) **for** ( each  $i$  from 1 to  $n$  ) {
- 3)   **for** (each  $j$  from 1 to  $i - 1$  ) {
- 4)       replace each production of the form  $A_i \rightarrow A_j \gamma$  by  
          the productions  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$   
          where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all the  
          current  $A_j$ -productions
- 5)       }
- 6)   $\Rightarrow$  eliminate the immediate left recursion among  
          the  $A_i$ -productions;
- 7) }

## Example — *continued*

- 6) eliminate the immediate left recursion among the  $A_i$ -productions;

- All  $A_i = A_2 = A$ -productions together,

$$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$$

- Eliminating the immediate left recursion among the  $A$ -productions yields the following,

$$A \rightarrow bdA' \mid A'$$

$$A' \rightarrow cA' \mid adA' \mid \epsilon$$

## Example — *continued*

- 6) eliminate the immediate left recursion among the  $A_i$ -productions;

- All  $A_i = A_2 = A$ -productions together,

$$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$$

- Eliminating the immediate left recursion among the  $A$ -productions yields the following,

$$A \rightarrow bdA' \mid A'$$

$$A' \rightarrow cA' \mid adA' \mid \epsilon$$

## Example — *continued*

- 1) arrange the nonterminals in some order  $A_1, A_2, \dots, A_n$ .
- 2) **for** ( each  $i$  from 1 to  $n$  ) {
- 3)   **for** (each  $j$  from 1 to  $i - 1$  ) {
- 4)     replace each production of the form  $A_i \rightarrow A_j \gamma$  by  
        the productions  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$   
        where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all the  
        current  $A_j$ -productions
- 5)     }
- 6)     eliminate the immediate left recursion among  
        the  $A_i$ -productions;
- 7) }

$i$  has attained the value of  $n = 2$  and the loops are no more entered.

## Example — *continued*

### Left-Recursive Grammar

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Ac \mid Sd \mid \epsilon$$

- Put together we get the following non-left-recursive grammar,

$$S \rightarrow Aa \mid b$$

$$A \rightarrow bdA' \mid A'$$

$$A' \rightarrow cA' \mid adaA' \mid \epsilon$$

- 1) arrange the nonterminals in some order  $A_1, A_2, \dots, A_n$ .
- 2) **for** ( each  $i$  from 1 to  $n$  ) {
- 3)   **for** (each  $j$  from 1 to  $i - 1$  ) {
- 4)     replace each production of the form  $A_i \rightarrow A_j \gamma$  by  
          the productions  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$   
          where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all the  
          current  $A_j$ -productions
- 5)    }
- 6)    eliminate the immediate left recursion among  
        the  $A_i$ -productions;
- 7) }

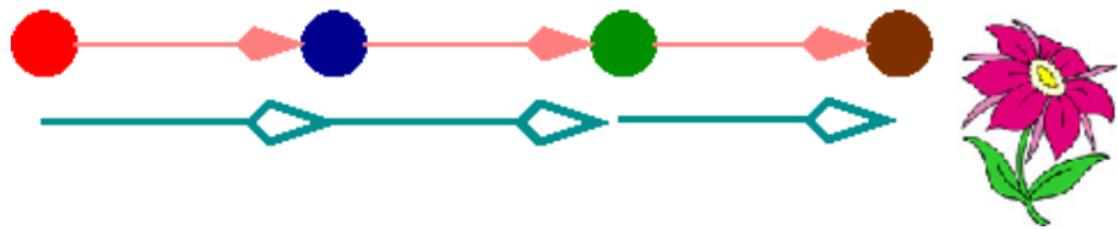
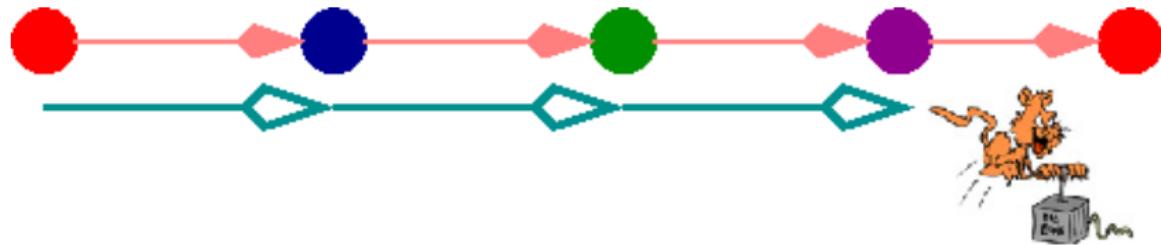
## Conceptual Technique Summary (AGAIN)

- Put some order in the nonterminals.
- Start by making first nonterminal productions left-recursion-free.
- Put the first nonterminal left-recursion-free productions into those of the second one.
- Now make the productions of second nonterminal left-recursion-free.
- Thus keep on growing the set of left-recursion-free productions.

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.
- The basic idea is that sometimes it is not clear which of two alternative productions to use to expand a nonterminal  $A$ .
- We may be able to rewrite the  $A$ -productions to defer the decision until we have seen enough of the input to make the right choice.

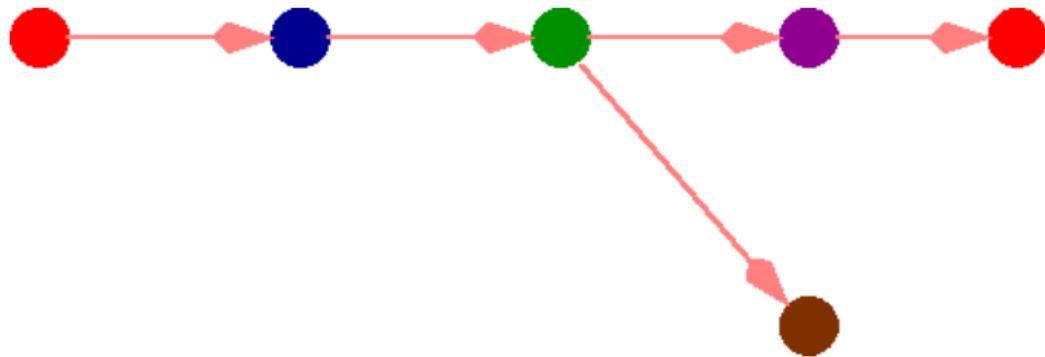
## Left Factoring — *continued*

Road Direction: *darkred* → *bluepigment* → *Green* → *Brown*



## Left Factoring — *continued*

Defer the decision until we have seen enough of the input to make the right choice.



## Left Factoring — *continued*

- We have the two productions,

$$\begin{aligned}stmt &\rightarrow \text{if } expr \text{ then } stmt \text{ else } stmt \\&\quad | \quad \text{if } expr \text{ then } stmt\end{aligned}$$

- On seeing the input token **if**, we cannot immediately tell which production to choose to expand *stmt*.

## Left Factoring — *continued*

- $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$  are two  $A$ -productions.
- The input begins with a nonempty string derived from  $\alpha$ .
- We do not know whether to expand  $A$  to  $\alpha\beta_1$  or  $\alpha\beta_2$ .
- However, we may defer the decision by expanding  $A$  to  $\alpha A'$ .
- Then, after seeing the input derived from  $\alpha$  we expand  $A'$  to  $\beta_1$  or  $\beta_2$ .
- Left-factored, the original productions become,

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

# Left Factoring Algorithm

INPUT. Grammar  $G$ .

OUTPUT An equivalent left-factored grammar.

# Left Factoring Algorithm — *continued*

## Method.

- For each nonterminal  $A$  find the longest prefix  $\alpha$  common to two or more of its alternatives.
- If  $\alpha \neq \epsilon$  (there is a nontrivial common prefix), replace all the  $A$  productions  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$  where  $\gamma$  represents all alternatives that do not begin with  $\alpha$  by

$$A \rightarrow \alpha A' \mid \gamma$$

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

where  $A'$  is a new nonterminal.

- Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

# Example

- The following grammar abstracts the dangling-else problem:

$$S \rightarrow iEtS \mid iEtSeS \mid a$$

$$E \rightarrow b$$

- Here  $i$ ,  $t$ , and  $e$  stand for **if**, **then** and **else**,  $E$  and  $S$  for “expression” and “statement.”
- Left-factored, this grammar becomes:

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

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

$$E \rightarrow b$$

## Example — *continued*

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

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

$$E \rightarrow b$$

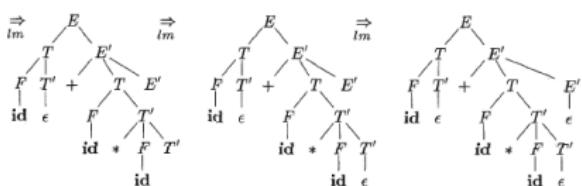
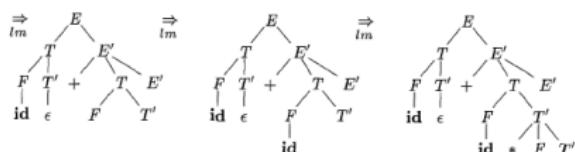
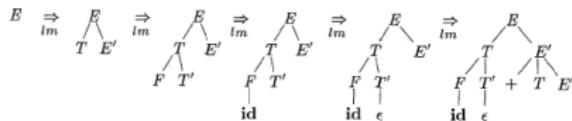
- Thus, we may expand  $S$  to  $iEtSS'$  on input  $i$ , and wait until  $iEtS$  has been seen to decide whether to expand  $S'$  to  $eS$  or to  $\epsilon$ .
- Of course, both of the grammars are ambiguous.
- On input  $e$ , it will not be clear which alternative for  $S$  should be chosen.

- Top-down parsing can be viewed as the problem of
  - constructing a parse tree for the input string,
  - starting from the root and
  - creating the nodes of the parse tree in preorder (depth-first).
- Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

# Example

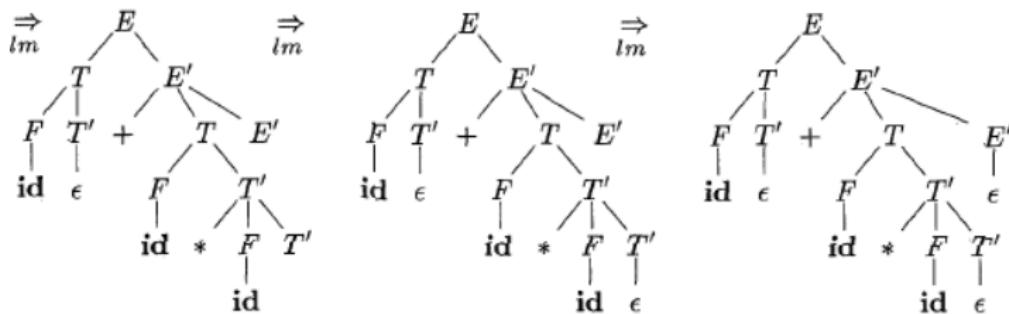
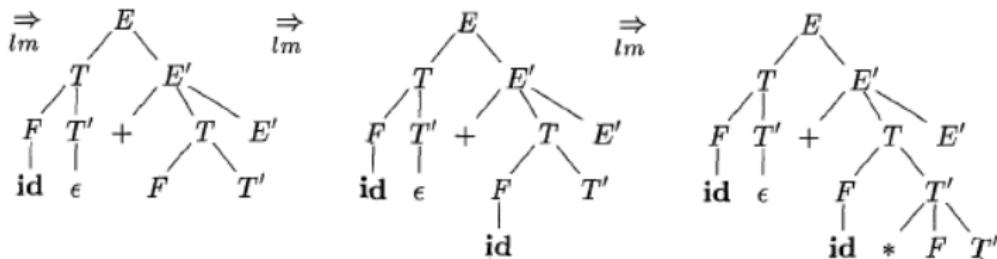
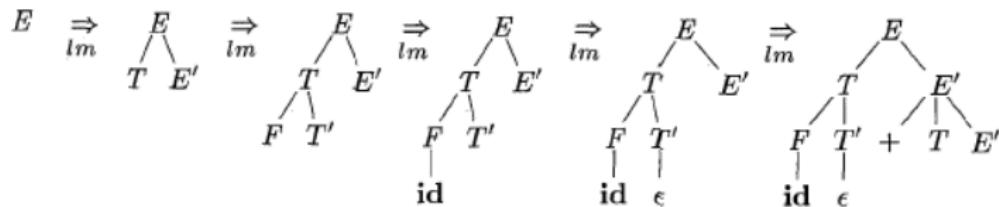
- The sequence of parse trees for the input **id + id \* id** is a top-down parse according to grammar.

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



Top-down parse for **id + id \* id**

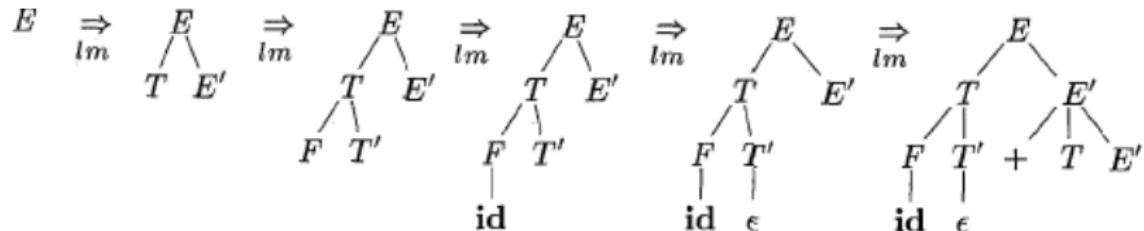
- This sequence of trees corresponds to a leftmost derivation of the input.



Top-down parse for **id + id \* id**

- At each step of a top-down parse, the key problem is that of determining the production to be applied for a nonterminal, say  $A$ .
- Once an  $A$ -production is chosen, the rest of the parsing process consists of “matching” the terminal symbols in the production body with the input string.

## Top-Down Parsing — *continued*



- Consider the top-down parse in figure.
- This constructs a tree with two nodes labeled  $E'$ .
- At the first  $E'$  node (in preorder), the production  $E' \rightarrow +TE'$  is chosen.
- At the second  $E'$  node, the production  $E' \rightarrow \epsilon$  is chosen.
- A predictive parser can choose between  $E'$ -productions by looking at the next input symbol.

- The class of grammars for which we can construct predictive parsers looking  $k$  symbols ahead in the input is sometimes called the  $\text{LL}(k)$  class.

# FIRST and FOLLOW

- The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar  $G$ .
- During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol.
- During panic-mode error recovery, sets of tokens produced by FOLLOW can be used as synchronizing tokens.

# FIRST and FOLLOW

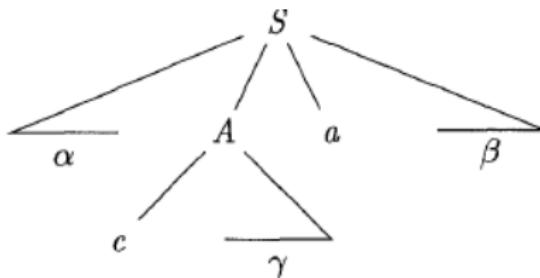
- The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar  $G$ .
- During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol.
- During panic-mode error recovery, sets of tokens produced by FOLLOW can be used as synchronizing tokens.

# FIRST and FOLLOW

- The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar  $G$ .
- During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol.
- During panic-mode error recovery, sets of tokens produced by FOLLOW can be used as synchronizing tokens.

## FIRST and FOLLOW — *continued*

- Define  $\text{FIRST}(\alpha)$ , where  $\alpha$  is any string of grammar symbols, to be the set of terminals that begin strings derived from  $\alpha$ .
- If  $\alpha \Rightarrow \epsilon$ , then  $\epsilon$  is also in  $\text{FIRST}(\alpha)$ .
- For example, in figure  $A \Rightarrow c\gamma$ , so  $c$  is in  $\text{FIRST}(A)$ .



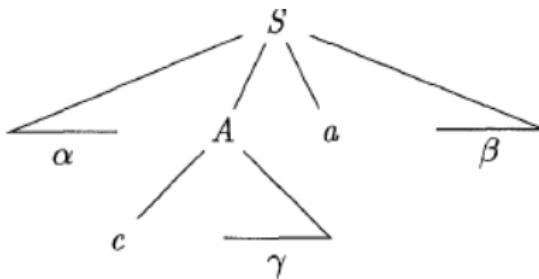
Terminal  $c$  is in  $\text{FIRST}(A)$  and  $a$  is in  $\text{FOLLOW}(A)$

## FIRST and FOLLOW — *continued*

- Let us see how FIRST can be used during predictive parsing.
- Consider two  $A$ -productions  $A \rightarrow \alpha \mid \beta$ , where  $\text{FIRST}(\alpha)$  and  $\text{FIRST}(\beta)$  are disjoint sets.
- We can then choose between these  $A$ -productions by looking at the next input symbol  $a$ , since  $a$  can be in at most one of  $\text{FIRST}(\alpha)$  and  $\text{FIRST}(\beta)$ , not both.
- For instance, if  $a$  is in  $\text{FIRST}(\beta)$  choose the production  $A \rightarrow \beta$ .

# FIRST and FOLLOW — *continued*

- Define  $\text{FOLLOW}(A)$ , nonterminal  $A$ , to be the set of terminals  $a$  that can appear immediately to the right of  $A$  in some sentential form.
- That is, the set of terminals  $a$  such that there exists a derivation of the form  $S \Rightarrow \alpha A a \beta$ , for some  $\alpha$  and  $\beta$ .
- Note that there may have been symbols between  $A$  and  $a$ , at some time during the derivation, but if so, they derived  $\epsilon$  and disappeared.



Terminal  $c$  is in  $\text{FIRST}(A)$  and  $a$  is in  $\text{FOLLOW}(A)$

## FIRST and FOLLOW — *continued*

- In addition, if  $A$  can be the rightmost symbol in some sentential form, then  $\$$  is in  $\text{FOLLOW}(A)$ .
- Recall that  $\$$  is a special “endmarker” symbol that is assumed not to be a symbol of any grammar.

## FIRST and FOLLOW — *continued*

To compute  $\text{FIRST}(X)$  for all grammar symbols  $X$ , apply the following rules until no more terminals or  $\epsilon$  can be added to any  $\text{FIRST}$  set.

1. If  $X$  is terminal, then  $\text{FIRST}(X)$  is  $\{X\}$ .

## FIRST and FOLLOW — *continued*

2.
    - If  $X$  is nonterminal and  $X \rightarrow Y_1 Y_2 \dots Y_k$  is a production for some  $k \geq 1$ , then place  $a$  in  $\text{FIRST}(X)$  if for some  $i$ ,  $a$  is in  $\text{FIRST}(Y_i)$ , and  $\epsilon$  is in all of  $\text{FIRST}(Y_1), \dots, \text{FIRST}(Y_{i-1})$ ; that is,  $Y_1, \dots, Y_{i-1} \xrightarrow{*} \epsilon$ .
    - If  $\epsilon$  is in  $\text{FIRST}(Y_j)$  for all  $j = 1, 2, \dots, k$  then add  $\epsilon$  to  $\text{FIRST}(X)$ .
  3. If  $X \rightarrow \epsilon$  is a production, then add  $\epsilon$  to  $\text{FIRST}(X)$ .
- 

- For example, everything in  $\text{FIRST}(Y_1)$  is surely in  $\text{FIRST}(X)$ .
- If  $Y_1$ , does not derive  $\epsilon$ , then we add nothing more to  $\text{FIRST}(X)$ , but if  $Y_1 \xrightarrow{*} \epsilon$ , then we add  $\text{FIRST}(Y_2)$  and so on.

## FIRST and FOLLOW — *continued*

Now, we can compute FIRST for any string  $X_1X_2\dots X_n$  as follows.

- Add to  $\text{FIRST}(X_1X_2\dots X_n)$  all the non- $\epsilon$  symbols of  $\text{FIRST}(X_1)$ .
- Also add the non- $\epsilon$  symbols of  $\text{FIRST}(X_2)$  if  $\epsilon$  is in  $\text{FIRST}(X_1)$ , the non- $\epsilon$  symbols of  $\text{FIRST}(X_3)$  if  $\epsilon$  is in both  $\text{FIRST}(X_1)$  and  $\text{FIRST}(X_2)$  and so on.
- Finally, add  $\epsilon$  to  $\text{FIRST}(X_1X_2\dots X_n)$  if, for all  $i$ ,  $\epsilon$  is in  $\text{FIRST}(X_i)$ .

## FIRST and FOLLOW — *continued*

To compute  $\text{FOLLOW}(A)$  for all nonterminals  $A$ , apply the following rules until nothing can be added to any FOLLOW set.

1. Place  $\$$  in  $\text{FOLLOW}(S)$ , where  $S$  is the start symbol and  $\$$  is the input right endmarker.
2. If there is a production  $A \rightarrow \alpha B \beta$ , then everything in  $\text{FIRST}(\beta)$  except  $\epsilon$  is in  $\text{FOLLOW}(B)$ .
3. If there is a production  $A \rightarrow \alpha B$ , or a production  $A \rightarrow \alpha B \beta$  where  $\text{FIRST}(\beta)$  contains  $\epsilon$ , then everything in  $\text{FOLLOW}(A)$  is in  $\text{FOLLOW}(B)$ .

# Example

1. If  $X$  is terminal, then  $\text{FIRST}(X)$  is  $\{X\}$ .
2. If  $X \rightarrow \epsilon$  is a production, then add  $\epsilon$  to  $\text{FIRST}(X)$ .
3. If  $X$  is nonterminal and  $X \rightarrow Y_1 Y_2 \dots Y_k$  is a production for some  $k \geq 1$ , then place  $a$  in  $\text{FIRST}(X)$  if for some  $i$ ,  $a$  is in  $\text{FIRST}(Y_i)$ , and  $\epsilon$  is in all of  $\text{FIRST}(Y_1), \dots, \text{FIRST}(Y_{i-1})$ ; that is,  $Y_1, \dots, Y_{i-1} \xrightarrow{*} \epsilon$ . If  $\epsilon$  is in  $\text{FIRST}(Y_j)$  for all  $j = 1, 2, \dots, k$  then add  $\epsilon$  to  $\text{FIRST}(X)$ .

---

Grammar,

$$\begin{array}{l} E \rightarrow TE' \\ E' \rightarrow +TE' \mid \epsilon \\ T \rightarrow FT' \\ T' \rightarrow *FT' \mid \epsilon \\ F \rightarrow (E) \mid \mathbf{id} \end{array}$$

Then,

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

# Example

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

- To see why, note that the two productions for  $F$  have bodies that start with these two terminal symbols,  $\text{id}$  and the left parenthesis.
- $T$  has only one production, and its body starts with  $F$ .
- Since  $F$  does not derive  $\epsilon$ ,  $\text{FIRST}(T)$  must be the same as  $\text{FIRST}(F)$ .
- The same argument covers  $\text{FIRST}(E)$ .

---

Grammar,

Then,

$$E \rightarrow TE'$$

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

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

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

$$T \rightarrow FT'$$

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

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

$$\text{FIRST}(T') = \{*, \epsilon\}$$

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

# Example

2.  $\text{FIRST}(E') = \{+, \epsilon\}$ .

- The reason is that one of the two productions for  $E'$  has a body that begins with terminal  $+$ , and the other's body is  $\epsilon$ .
- Whenever a nonterminal derives  $\epsilon$ , we place  $\epsilon$  in  $\text{FIRST}$  for that nonterminal.

---

Grammar,

$$\begin{array}{l} E \rightarrow TE' \\ E' \rightarrow +TE' \mid \epsilon \\ T \rightarrow FT' \\ T' \rightarrow *FT' \mid \epsilon \\ F \rightarrow (E) \mid \mathbf{id} \end{array}$$

Then,

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

# Example

3.  $\text{FIRST}(T') = \{*, \epsilon\}$ .

■ The reasoning is analogous to that for  $\text{FIRST}(E')$ .

---

Grammar,

$$\begin{array}{l} E \rightarrow TE' \\ E' \rightarrow +TE' \mid \epsilon \\ T \rightarrow FT' \\ T' \rightarrow *FT' \mid \epsilon \\ F \rightarrow (E) \mid \mathbf{id} \end{array}$$

Then,

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

# Example — *continued*

## ■ Grammar:

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

## ■ Computation of FOLLOW:

FOLLOW( $E$ )	FOLLOW( $E'$ )	FOLLOW( $T$ )	FOLLOW( $T'$ )	FOLLOW( $F$ )

Initially all sets are empty

--	--	--	--	--

Put  $\$$  in FOLLOW( $E$ ) by rule (1) (Place  $\$$  in FOLLOW( $S$ ), where  $S$  is the start symbol and  $\$$  is the input right endmarker)

\$				
----	--	--	--	--

## Example — *continued*

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

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

By rule (2) (If there is a production  $A \rightarrow \alpha B \beta$ , then everything in  $\text{FIRST}(\beta)$  except for  $\epsilon$  is placed in  $\text{FOLLOW}(B)$ ) applied to,

$E \rightarrow TE': \text{FIRST}(E') \text{ except } \epsilon \text{ i.e. } \{+\} \text{ are in } \text{FOLLOW}(T)$

$E' \rightarrow +TE': \text{FIRST}(E') \text{ except } \epsilon \text{ i.e. } \{+\} \text{ are in } \text{FOLLOW}(T)$

$T \rightarrow FT': \text{FIRST}(T') \text{ except } \epsilon \text{ i.e. } \{*\} \text{ are in } \text{FOLLOW}(F)$

$T \rightarrow *FT': \text{FIRST}(T') \text{ except } \epsilon \text{ i.e. } \{*\} \text{ are in } \text{FOLLOW}(F)$

$F \rightarrow (E): \text{FIRST}(\text{)} \text{ i.e. } \{\}\} \text{ are in } \text{FOLLOW}(E)$

$\text{FOLLOW}(E)$	$\text{FOLLOW}(E')$	$\text{FOLLOW}(T)$	$\text{FOLLOW}(T')$	$\text{FOLLOW}(F)$
$\text{, } \text{)}$		$+$		$*$

Rule (2) is not applicable any more since it depends only on  $\text{FIRST}$ , which are now stable sets.

# Example — *continued*

**Application of rule (3)** (If there is a production  $A \rightarrow \alpha B$ , or a production  $A \rightarrow \alpha B\beta$  where  $\text{FIRST}(\beta)$  contains  $\epsilon$  (i.e.,  $\beta \xrightarrow{*} \epsilon$ ), then everything in  $\text{FOLLOW}(A)$  is in  $\text{FOLLOW}(B)$ )

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

$\text{FOLLOW}(E)$	$\text{FOLLOW}(E')$	$\text{FOLLOW}(T)$	$\text{FOLLOW}(T')$	$\text{FOLLOW}(F)$
$\$, )$		$+$		$*$

$E \rightarrow TE'$ : Everything in  $\text{FOLLOW}(E)$  are in  $\text{FOLLOW}(E')$

$\$, )$	$\$, )$	$+$		$*$
---------	---------	-----	--	-----

$E' \rightarrow +TE'$  (also  $\epsilon \in \text{FIRST}(E')$ ): Everything in  $\text{FOLLOW}(E')$  are in  $\text{FOLLOW}(T)$

$\$, )$	$\$, )$	$+, \$, )$		$*$
---------	---------	------------	--	-----

$T \rightarrow FT'$ : Everything in  $\text{FOLLOW}(T)$  are in  $\text{FOLLOW}(T')$

$\$, )$	$\$, )$	$+, \$, )$	$+, \$, )$	$*$
---------	---------	------------	------------	-----

# Example — *continued*

**Application of rule (3)** (If there is a production  $A \rightarrow \alpha B$ , or a production  $A \rightarrow \alpha B\beta$  where  $\text{FIRST}(\beta)$  contains  $\epsilon$  (i.e.,  $\beta \xrightarrow{*} \epsilon$ ), then everything in  $\text{FOLLOW}(A)$  is in  $\text{FOLLOW}(B)$ )

$$\begin{array}{l} E \rightarrow TE' \\ E' \rightarrow +TE' | \epsilon \end{array} \quad \begin{array}{l} T \rightarrow FT' \\ T' \rightarrow *FT' | \epsilon \end{array} \quad \begin{array}{l} F \rightarrow (E) | \text{id} \end{array}$$

$\text{FOLLOW}(E)$	$\text{FOLLOW}(E')$	$\text{FOLLOW}(T)$	$\text{FOLLOW}(T')$	$\text{FOLLOW}(F)$
$\$, )$		$+$		$*$

$E \rightarrow TE'$ : Everything in  $\text{FOLLOW}(E)$  are in  $\text{FOLLOW}(E')$

$\$, )$	$\$, )$	$+$		$*$
---------	---------	-----	--	-----

$E' \rightarrow +TE'$  (also  $\epsilon \in \text{FIRST}(E')$ ): Everything in  $\text{FOLLOW}(E')$  are in  $\text{FOLLOW}(T)$

$\$, )$	$\$, )$	$+, \$, )$		$*$
---------	---------	------------	--	-----

$T \rightarrow FT'$ : Everything in  $\text{FOLLOW}(T)$  are in  $\text{FOLLOW}(T')$

$\$, )$	$\$, )$	$+, \$, )$	$+, \$, )$	$*$
---------	---------	------------	------------	-----

## Example — *continued*

**Application of rule (3)** (If there is a production  $A \rightarrow \alpha B$ , or a production  $A \rightarrow \alpha B\beta$  where  $\text{FIRST}(\beta)$  contains  $\epsilon$  (i.e.,  $\beta \xrightarrow{*} \epsilon$ ), then everything in  $\text{FOLLOW}(A)$  is in  $\text{FOLLOW}(B)$ )

$$\begin{array}{l} E \rightarrow TE' \\ E' \rightarrow +TE' | \epsilon \end{array} \quad \begin{array}{l} T \rightarrow FT' \\ T' \rightarrow *FT' | \epsilon \end{array} \quad \begin{array}{l} F \rightarrow (E) | \text{id} \end{array}$$

$\text{FOLLOW}(E)$	$\text{FOLLOW}(E')$	$\text{FOLLOW}(T)$	$\text{FOLLOW}(T')$	$\text{FOLLOW}(F)$
$\$, )$		$+$		$*$

$E \rightarrow TE'$ : Everything in  $\text{FOLLOW}(E)$  are in  $\text{FOLLOW}(E')$

$\$, )$	$\$, )$	$+$		$*$
---------	---------	-----	--	-----

$E' \rightarrow +TE'$  (also  $\epsilon \in \text{FIRST}(E')$ ): Everything in  $\text{FOLLOW}(E')$  are in  $\text{FOLLOW}(T)$

$\$, )$	$\$, )$	$+, \$, )$		$*$
---------	---------	------------	--	-----

$T \rightarrow FT'$ : Everything in  $\text{FOLLOW}(T)$  are in  $\text{FOLLOW}(T')$

$\$, )$	$\$, )$	$+, \$, )$	$+, \$, )$	$*$
---------	---------	------------	------------	-----

## Example — *continued*

**Application of rule (3)** (If there is a production  $A \rightarrow \alpha B$ , or a production  $A \rightarrow \alpha B\beta$  where  $\text{FIRST}(\beta)$  contains  $\epsilon$  (i.e.,  $\beta \xrightarrow{*} \epsilon$ ), then everything in  $\text{FOLLOW}(A)$  is in  $\text{FOLLOW}(B)$ )

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

$\text{FOLLOW}(E)$	$\text{FOLLOW}(E')$	$\text{FOLLOW}(T)$	$\text{FOLLOW}(T')$	$\text{FOLLOW}(F)$
$\$, )$		$+$		$*$

$E \rightarrow TE'$ : Everything in  $\text{FOLLOW}(E)$  are in  $\text{FOLLOW}(E')$

$\$, )$	$\$, )$	$+$		$*$
---------	---------	-----	--	-----

$E' \rightarrow +TE'$  (also  $\epsilon \in \text{FIRST}(E')$ ): Everything in  $\text{FOLLOW}(E')$  are in  $\text{FOLLOW}(T)$

$\$, )$	$\$, )$	$+, \$, )$		$*$
---------	---------	------------	--	-----

$T \rightarrow FT'$ : Everything in  $\text{FOLLOW}(T)$  are in  $\text{FOLLOW}(T')$

$\$, )$	$\$, )$	$+, \$, )$	$+, \$, )$	$*$
---------	---------	------------	------------	-----

## Example — *continued*

**Application of rule (3)** (If there is a production  $A \rightarrow \alpha B$ , or a production  $A \rightarrow \alpha B\beta$  where  $\text{FIRST}(\beta)$  contains  $\epsilon$  (i.e.,  $\beta \xrightarrow{*} \epsilon$ ), then everything in  $\text{FOLLOW}(A)$  is in  $\text{FOLLOW}(B)$ )

$$\begin{array}{l} E \rightarrow TE' \\ E' \rightarrow +TE' | \epsilon \end{array} \quad \begin{array}{l} T \rightarrow FT' \\ T' \rightarrow *FT' | \epsilon \end{array} \quad \begin{array}{l} F \rightarrow (E) | \text{id} \end{array}$$

$\text{FOLLOW}(E)$	$\text{FOLLOW}(E')$	$\text{FOLLOW}(T)$	$\text{FOLLOW}(T')$	$\text{FOLLOW}(F)$
$\$, )$		$+$		$*$

$E \rightarrow TE'$ : Everything in  $\text{FOLLOW}(E)$  are in  $\text{FOLLOW}(E')$

$\$, )$	$\$, )$	$+$		$*$
---------	---------	-----	--	-----

$E' \rightarrow +TE'$  (also  $\epsilon \in \text{FIRST}(E')$ ): Everything in  $\text{FOLLOW}(E')$  are in  $\text{FOLLOW}(T)$

$\$, )$	$\$, )$	$+, \$, )$		$*$
---------	---------	------------	--	-----

$T \rightarrow FT'$ : Everything in  $\text{FOLLOW}(T)$  are in  $\text{FOLLOW}(T')$

$\$, )$	$\$, )$	$+, \$, )$	$+, \$, )$	$*$
---------	---------	------------	------------	-----

## Example — *continued*

**Application of rule (3)** (If there is a production  $A \rightarrow \alpha B$ , or a production  $A \rightarrow \alpha B\beta$  where  $\text{FIRST}(\beta)$  contains  $\epsilon$  (i.e.,  $\beta \xrightarrow{*} \epsilon$ ), then everything in  $\text{FOLLOW}(A)$  is in  $\text{FOLLOW}(B)$ )

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

$\text{FOLLOW}(E)$	$\text{FOLLOW}(E')$	$\text{FOLLOW}(T)$	$\text{FOLLOW}(T')$	$\text{FOLLOW}(F)$
$\$, )$		$+$		$*$

$E \rightarrow TE'$ : Everything in  $\text{FOLLOW}(E)$  are in  $\text{FOLLOW}(E')$

$\$, )$	$\$, )$	$+$		$*$
---------	---------	-----	--	-----

$E' \rightarrow +TE'$  (also  $\epsilon \in \text{FIRST}(E')$ ): Everything in  $\text{FOLLOW}(E')$  are in  $\text{FOLLOW}(T)$

$\$, )$	$\$, )$	$+, \$, )$		$*$
---------	---------	------------	--	-----

$T \rightarrow FT'$ : Everything in  $\text{FOLLOW}(T)$  are in  $\text{FOLLOW}(T')$

$\$, )$	$\$, )$	$+, \$, )$	$+, \$, )$	$*$
---------	---------	------------	------------	-----

## Example — *continued*

**Application of rule (3) — *continued*** (If there is a production  $A \rightarrow \alpha B$ , or a production  $A \rightarrow \alpha B\beta$  where  $\text{FIRST}(\beta)$  contains  $\epsilon$  (i.e.,  $\beta \xrightarrow{*} \epsilon$ ), then everything in  $\text{FOLLOW}(A)$  is in  $\text{FOLLOW}(B)$ )

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

$\text{FOLLOW}(E)$	$\text{FOLLOW}(E')$	$\text{FOLLOW}(T)$	$\text{FOLLOW}(T')$	$\text{FOLLOW}(F)$
$\$, )$	$\$, )$	$+, \$, )$	$+, \$, )$	$*$

$T' \rightarrow *FT'$  (also  $\epsilon \in \text{FIRST}(T')$ ): Everything in  $\text{FOLLOW}(T')$  are in  $\text{FOLLOW}(F)$

$\$, )$	$\$, )$	$+, \$, )$	$+, \$, )$	$*, +, \$, )$
---------	---------	------------	------------	---------------

We can try applying Rule (3) again, but will find that the sets have stabilized (nothing can be added to any  $\text{FOLLOW}$  set).

## Example — *continued*

**Application of rule (3) — *continued*** (If there is a production  $A \rightarrow \alpha B$ , or a production  $A \rightarrow \alpha B\beta$  where  $\text{FIRST}(\beta)$  contains  $\epsilon$  (i.e.,  $\beta \xrightarrow{*} \epsilon$ ), then everything in  $\text{FOLLOW}(A)$  is in  $\text{FOLLOW}(B)$ )

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

$\text{FOLLOW}(E)$	$\text{FOLLOW}(E')$	$\text{FOLLOW}(T)$	$\text{FOLLOW}(T')$	$\text{FOLLOW}(F)$
$\$, )$	$\$, )$	$+, \$, )$	$+, \$, )$	$*$

$T' \rightarrow *FT'$  (also  $\epsilon \in \text{FIRST}(T')$ ): Everything in  $\text{FOLLOW}(T')$  are in  $\text{FOLLOW}(F)$

$\$, )$	$\$, )$	$+, \$, )$	$+, \$, )$	$*, +, \$, )$
---------	---------	------------	------------	---------------

We can try applying Rule (3) again, but will find that the sets have stabilized (nothing can be added to any  $\text{FOLLOW}$  set).

## Example — *continued*

**Application of rule (3) — *continued*** (If there is a production  $A \rightarrow \alpha B$ , or a production  $A \rightarrow \alpha B\beta$  where  $\text{FIRST}(\beta)$  contains  $\epsilon$  (i.e.,  $\beta \xrightarrow{*} \epsilon$ ), then everything in  $\text{FOLLOW}(A)$  is in  $\text{FOLLOW}(B)$ )

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

$\text{FOLLOW}(E)$	$\text{FOLLOW}(E')$	$\text{FOLLOW}(T)$	$\text{FOLLOW}(T')$	$\text{FOLLOW}(F)$
$\$, )$	$\$, )$	$+, \$, )$	$+, \$, )$	$*$

$T' \rightarrow *FT'$  (also  $\epsilon \in \text{FIRST}(T')$ ): Everything in  $\text{FOLLOW}(T')$  are in  $\text{FOLLOW}(F)$

$\$, )$	$\$, )$	$+, \$, )$	$+, \$, )$	$*, +, \$, )$
---------	---------	------------	------------	---------------

We can try applying Rule (3) again, but will find that the sets have stabilized (nothing can be added to any FOLLOW set).

## Example — *continued*

**Application of rule (3) — *continued*** (If there is a production  $A \rightarrow \alpha B$ , or a production  $A \rightarrow \alpha B\beta$  where  $\text{FIRST}(\beta)$  contains  $\epsilon$  (i.e.,  $\beta \xrightarrow{*} \epsilon$ ), then everything in  $\text{FOLLOW}(A)$  is in  $\text{FOLLOW}(B)$ )

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

$\text{FOLLOW}(E)$	$\text{FOLLOW}(E')$	$\text{FOLLOW}(T)$	$\text{FOLLOW}(T')$	$\text{FOLLOW}(F)$
$\$, )$	$\$, )$	$+, \$, )$	$+, \$, )$	$*$

$T' \rightarrow *FT'$  (also  $\epsilon \in \text{FIRST}(T')$ ): Everything in  $\text{FOLLOW}(T')$  are in  $\text{FOLLOW}(F)$

$\$, )$	$\$, )$	$+, \$, )$	$+, \$, )$	$*, +, \$, )$
---------	---------	------------	------------	---------------

We can try applying Rule (3) again, but will find that the sets have stabilized (nothing can be added to any FOLLOW set).

- Predictive parsers, that is, recursive-descent parsers needing no backtracking, can be constructed for a class of grammars called LL(1).
- The first “L” in LL(1) stands for scanning the input from left to right.
- The second “L” for producing a leftmost derivation.
- And the “1” for using one input symbol of lookahead at each step to make parsing action decisions.

- The class of LL(1) grammars is rich enough to cover most programming constructs.
- Although care is needed in writing a suitable grammar for the source language.
- For example, no left-recursive or ambiguous grammar can be LL(1).

A grammar  $G$  is LL(1) if and only if whenever  $A \rightarrow \alpha \mid \beta$  are two distinct productions of  $G$  the following conditions hold:

1. For no terminal  $a$  do both  $\alpha$  and  $\beta$  derive strings beginning with  $a$ .
2. At most one of  $\alpha$  and  $\beta$  can derive the empty string.
3. If  $\beta \xrightarrow{*} \epsilon$ , then  $\alpha$  does not derive any string beginning with a terminal in  $\text{FOLLOW}(A)$ .

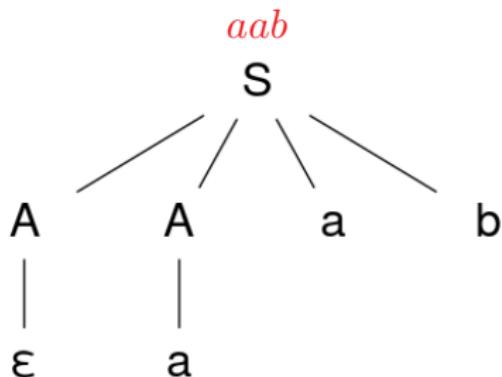
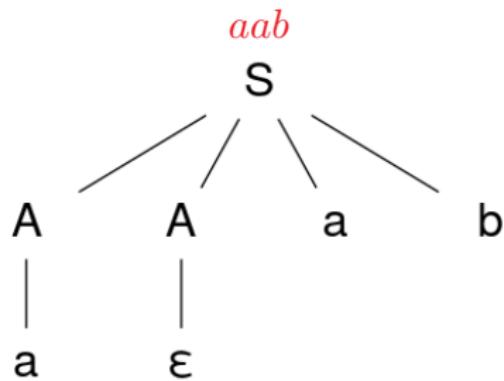
Likewise,  $\alpha \xrightarrow{*} \epsilon$ , then  $\beta$  does not derive any string beginning with a terminal in  $\text{FOLLOW}(A)$ .

1. For no terminal  $a$  do both  $\alpha$  and  $\beta$  derive strings beginning with  $a$ .
2. At most one of  $\alpha$  and  $\beta$  can derive the empty string.
3. If  $\beta \xrightarrow{*} \epsilon$ , then  $\alpha$  does not derive any string beginning with a terminal in  $\text{FOLLOW}(A)$ .

Likewise,  $\alpha \xrightarrow{*} \epsilon$ , then  $\beta$  does not derive any string beginning with a terminal in  $\text{FOLLOW}(A)$ .

- 
- The first two conditions are equivalent to the statement that  $\text{FIRST}(\alpha)$  and  $\text{FIRST}(\beta)$  are disjoint sets.
  - The third condition is equivalent to stating that if  $\epsilon$  is in  $\text{FIRST}(\beta)$ , then  $\text{FIRST}(\alpha)$  and  $\text{FOLLOW}(A)$  are disjoint sets, and likewise if  $\epsilon$  is in  $\text{FIRST}(\alpha)$ .

# A Case of a non-LL(1) Grammar

$$S \rightarrow AAab \mid BBba$$
$$A \rightarrow a \mid \epsilon$$
$$B \rightarrow b \mid \epsilon$$


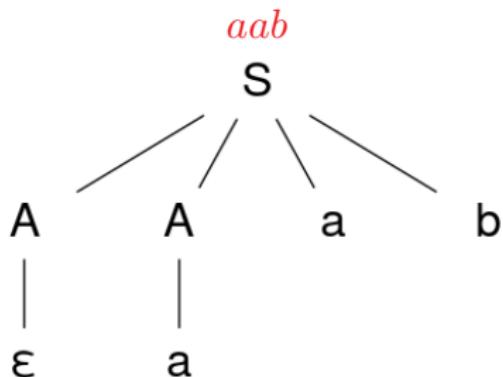
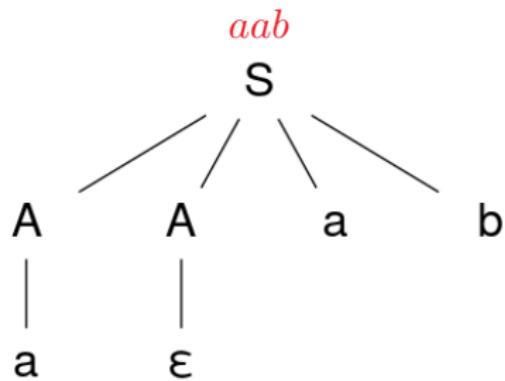
- Now, when expanding the first  $A$ , we can not decide which of the two production rules to be used.
- Both give us the promise of an  $a$ .

# A Case of a non-LL(1) Grammar

$$S \rightarrow AAab \mid BBba$$

$$A \rightarrow a \mid \epsilon$$

$$B \rightarrow b \mid \epsilon$$



- Two derivations for the same string.

- Predictive parsers can be constructed for LL(1) grammars since the proper production to apply for a nonterminal can be selected by looking only at the current input symbol.
- Flow-of-control constructs, with their distinguishing key-words, generally satisfy the LL(1) constraints.

- For instance, if we have the productions,

$$\begin{aligned}stmt &\rightarrow \mathbf{if} \ ( \ expr \ ) \ stmt \ \mathbf{else} \ stmt \\&\quad | \quad \mathbf{while} \ ( \ expr \ ) \ stmt \\&\quad | \quad \{ \ stmt\_list \ }\end{aligned}$$

then the keywords **if**, **while**, and the symbol { tell us which alternative is the one that could possibly succeed if we are to find a statement.

- The next algorithm collects the information from FIRST and FOLLOW sets into a predictive parsing table  $M[A, a]$ , a two dimensional array, where  $A$  is a nonterminal, and  $a$  is a terminal or the symbol  $\$$ , the input endmarker.
- The idea behind the algorithm is the following.
- Suppose  $A \rightarrow \alpha$  is a production with  $a$  in  $\text{FIRST}(\alpha)$ .
- Then, the parser will expand  $A$  by  $\alpha$  when the current input symbol is  $a$ .
- The only complication occurs when  $\alpha = \epsilon$  or  $\alpha \stackrel{*}{\Rightarrow} \epsilon$ .
- In this case, we should again expand  $A$  by  $\alpha$  if the current input symbol is in  $\text{FOLLOW}(A)$ , or if the  $\$$  on the input has been reached and  $\$$  is in  $\text{FOLLOW}(A)$ .

# Algorithm for Construction of a Predictive Parsing Table

INPUT: Grammar  $G$ .

OUTPUT: Parsing table  $M$ .

METHOD: For each production  $A \rightarrow \alpha$  of the grammar, do the following:

1. For each terminal  $a$  in  $\text{FIRST}(\alpha)$ , add  $A \rightarrow \alpha$  to  $M[A, a]$ .
2. If  $\epsilon$  is in  $\text{FIRST}(\alpha)$ , then for each terminal  $b$  in  $\text{FOLLOW}(A)$ , add  $A \rightarrow \alpha$  to  $M[A, b]$ .  
If  $\epsilon$  is in  $\text{FIRST}(\alpha)$  and  $\$$  is in  $\text{FOLLOW}(A)$ , add  $A \rightarrow \alpha$  to  $M[A, \$]$  as well.

If, after performing the above, there is no production at all in  $M[A, a]$ , then set  $M[A, a]$  to **error** (which we normally represent by an empty entry in the table).

# Example

- For the expression grammar below,

$$\begin{array}{lcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \mid \epsilon \\ T & \rightarrow & FT' \\ T' & \rightarrow & *FT' \mid \epsilon \end{array}$$

the algorithm produces the parsing table in figure.

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	\$
$E$	$E \rightarrow TE'$			$E \rightarrow TE'$		
$E'$		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
$T$	$T \rightarrow FT'$			$T \rightarrow FT'$		
$T'$		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
$F$	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

- Blanks are error entries.
- Nonblanks indicate a production with which to expand a nonterminal.

## Example — *continued*

For each production  $A \rightarrow \alpha$  of the grammar, do the following:

1. For each terminal  $a$  in  $\text{FIRST}(\alpha)$ , add  $A \rightarrow \alpha$  to  $M[A, a]$ .
2. If  $\epsilon$  is in  $\text{FIRST}(\alpha)$ , then for each terminal  $b$  in  $\text{FOLLOW}(A)$ , add  $A \rightarrow \alpha$  to  $M[A, b]$ .  
If  $\epsilon$  is in  $\text{FIRST}(\alpha)$  and  $\$$  is in  $\text{FOLLOW}(A)$ , add  $A \rightarrow \alpha$  to  $M[A, \$]$  as well.

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	\$
$E$	$E \rightarrow TE'$			$E \rightarrow TE'$		
$E'$		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
$T$	$T \rightarrow FT'$			$T \rightarrow FT'$		
$T'$		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
$F$	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

- Consider production  $E \rightarrow TE'$ .
- Since,

$$\text{FIRST}(TE') = \text{FIRST}(T) = \{(), \text{id}\}$$

this production is added to  $M[E, ()]$  and  $M[E, \text{id}]$ .

## Example — *continued*

For each production  $A \rightarrow \alpha$  of the grammar, do the following:

1. For each terminal  $a$  in  $\text{FIRST}(\alpha)$ , add  $A \rightarrow \alpha$  to  $M[A, a]$ .
  2. If  $\epsilon$  is in  $\text{FIRST}(\alpha)$ , then for each terminal  $b$  in  $\text{FOLLOW}(A)$ , add  $A \rightarrow \alpha$  to  $M[A, b]$ .
- If  $\epsilon$  is in  $\text{FIRST}(\alpha)$  and  $\$$  is in  $\text{FOLLOW}(A)$ , add  $A \rightarrow \alpha$  to  $M[A, \$]$  as well.

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	\$
$E$	$E \rightarrow TE'$			$E \rightarrow TE'$		
$E'$		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
$T$	$T \rightarrow FT'$			$T \rightarrow FT'$		
$T'$		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
$F$	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

- Production  $E' \rightarrow +TE'$  is added to  $M[E', +]$  since  $\text{FIRST}(+TE') = \{+\}$ .

## Example — *continued*

For each production  $A \rightarrow \alpha$  of the grammar, do the following:

1. For each terminal  $a$  in  $\text{FIRST}(\alpha)$ , add  $A \rightarrow \alpha$  to  $M[A, a]$ .
2. If  $\epsilon$  is in  $\text{FIRST}(\alpha)$ , then for each terminal  $b$  in  $\text{FOLLOW}(A)$ , add  $A \rightarrow \alpha$  to  $M[A, b]$ .  
If  $\epsilon$  is in  $\text{FIRST}(\alpha)$  and  $\$$  is in  $\text{FOLLOW}(A)$ , add  $A \rightarrow \alpha$  to  $M[A, \$]$  as well.

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	\$
$E$	$E \rightarrow TE'$			$E \rightarrow TE'$		
$E'$		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
$T$	$T \rightarrow FT'$			$T \rightarrow FT'$		
$T'$		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
$F$	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

- Since  $\text{FOLLOW}(E') = \{ \), \$ \}$ , production  $E' \rightarrow \epsilon$  is added to  $M[E', \text{)}]$  and  $M[E', \$]$ .

- The aforementioned algorithm can be applied to any grammar  $G$  to produce a parsing table  $M$ .
- For every LL(1) grammar, each parsing-table entry uniquely identifies a production or signals an error.

## Algorithm . . . Predictive Parsing Table — *continued*

- For some grammars, however,  $M$  may have some entries that are multiply defined.
- For example, if  $G$  is left-recursive or ambiguous, then  $M$  will have at least one multiply defined entry.
- Although left-recursion elimination and left factoring are easy to do, there are some grammars for which no amount of alteration will produce an LL(1) grammar.
- The language in the following example has no LL(1) grammar at all.

## Example

- The following grammar, which abstracts the dangling-else problem, is repeated here:
- The grammar,

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

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

$$E \rightarrow b$$

- The grammar is ambiguous.
- On input  $e$ , it will not be clear which alternative for  $S'$  should be chosen.

## Example — *continued*

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

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

$$E \rightarrow b$$

NON - TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

## Example — *continued*

NON - TERMINAL	INPUT SYMBOL					
	<i>a</i>	<i>b</i>	<i>e</i>	<i>i</i>	<i>t</i>	$\$$
<i>S</i>	$S \rightarrow a$			$S \rightarrow iEtSS'$		
<i>S'</i>			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
<i>E</i>		$E \rightarrow b$				

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

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

$$E \rightarrow b$$

- The entry for  $M[S', e]$  contains both  $S' \rightarrow eS$  and  $S' \rightarrow \epsilon$ , since  $\text{FOLLOW}(S') = \{e, \$\}$

## Example — *continued*

NON - TERMINAL	INPUT SYMBOL					
	<i>a</i>	<i>b</i>	<i>e</i>	<i>i</i>	<i>t</i>	$\$$
<i>S</i>	$S \rightarrow a$			$S \rightarrow iEtSS'$		
<i>S'</i>			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
<i>E</i>		$E \rightarrow b$				

- The grammar is ambiguous and the ambiguity is manifested by a choice in what production to use when an *e* (**else**) is seen.
- We can resolve the ambiguity if we choose  $S' \rightarrow eS$ .
- This choice corresponds to associating **else**'s with the closest previous **then**'s.

## Example — *continued*

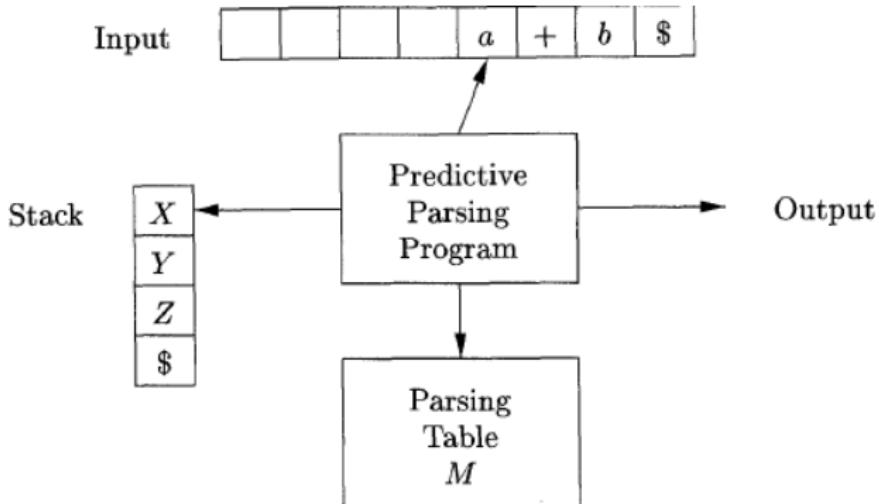
NON - TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	\$
$S$	$S \rightarrow a$			$S \rightarrow iEtSS'$		
$S'$			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
$E$		$E \rightarrow b$				

- Note that the choice  $S' \rightarrow \epsilon$  would prevent  $e$  from ever being put on the stack or removed from the input, and is therefore surely wrong.

# Nonrecursive Predictive Parsing

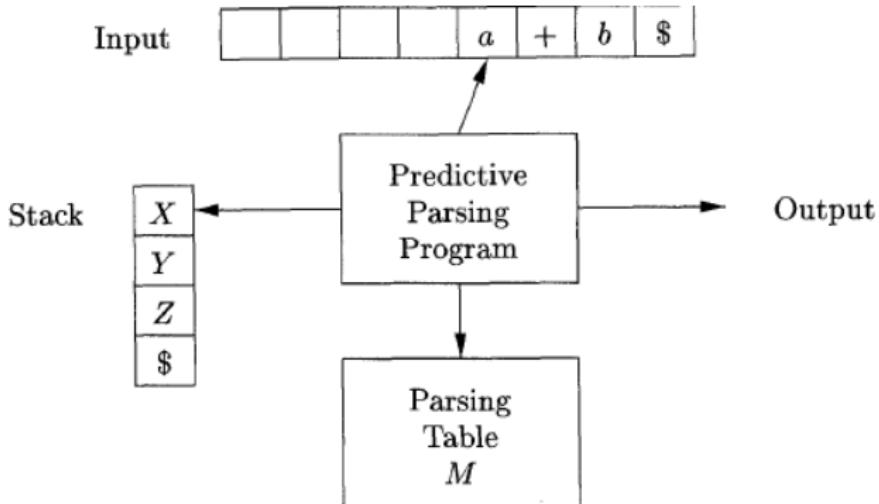
- A nonrecursive predictive parser can be built by maintaining a stack explicitly, rather than implicitly via recursive calls.
- The parser mimics a leftmost derivation.
- If  $w$  is the input that has been matched so far, then the stack holds a sequence of grammar symbols  $\alpha$  such that

$$S \xrightarrow[lm]{} w\alpha$$



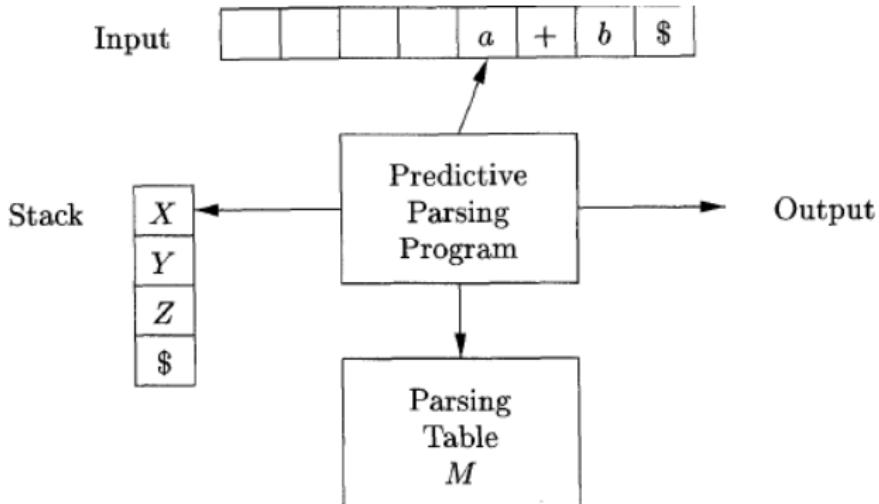
## Model of a table-driven predictive parser

- The table-driven parser in figure has an input buffer, a stack containing a sequence of grammar symbols, a parsing table constructed by algorithm, and an output stream.



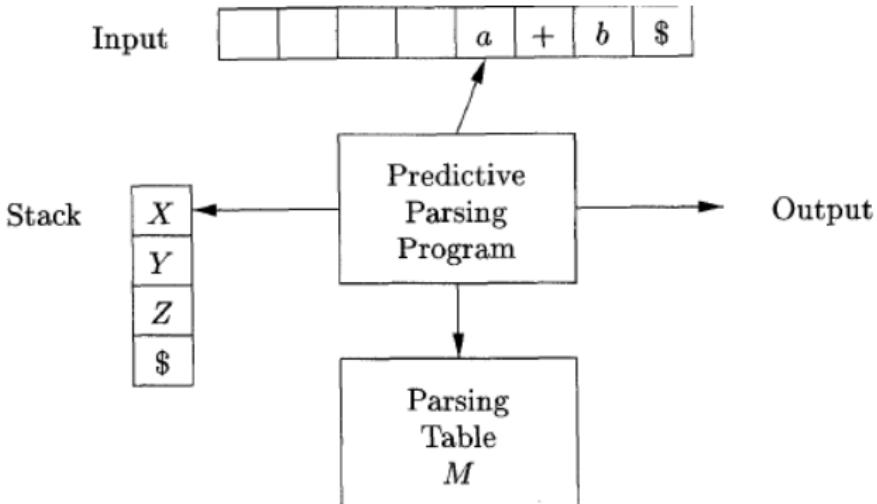
Model of a table-driven predictive parser

- The input buffer contains the string to be parsed, followed by the endmarker  $\$$ .
- We reuse the symbol  $\$$  to mark the bottom of the stack, which initially contains the start symbol of the grammar on top of  $\$$ .



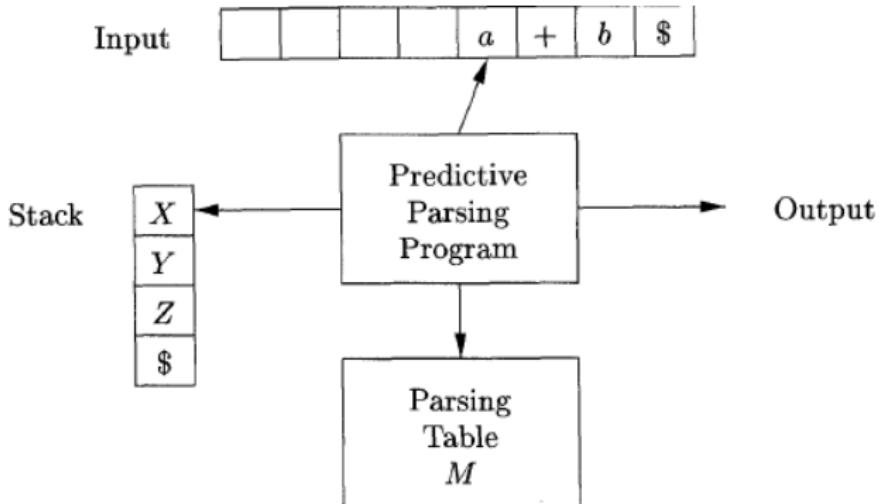
Model of a table-driven predictive parser

- The parser is controlled by a program that considers  $X$ , the symbol on top of the stack, and  $a$ , the current input symbol.



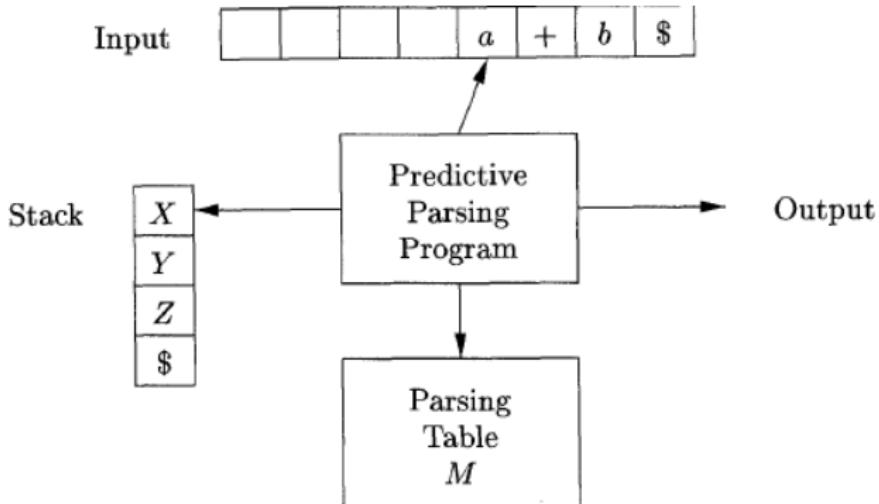
Model of a table-driven predictive parser

- If  $X$  is a nonterminal, the parser chooses an  $X$ -production by consulting entry  $M[X, a]$  of the parsing table  $M$ .
- Additional code could be executed here, for example, code to construct a node in a parse tree.



Model of a table-driven predictive parser

- Otherwise, it checks for a match between the terminal  $X$  and current input symbol  $a$ .



## Model of a table-driven predictive parser

- The behavior of the parser can be described in terms of its configurations, which give the stack contents and the remaining input.
  - The next algorithm describes how configurations are manipulated.

# Nonrecursive Predictive Parsing — *continued*

Nonrecursive predictive parsing.

**INPUT.** A string  $w$  and a parsing table  $M$  for grammar  $G$ .

**OUTPUT.** ■ If  $w$  is in  $L(G)$ , a leftmost derivation of  $w$ ;  
■ otherwise, an error indication.

**METHOD.** ■ Initially, the parser is in a configuration in which it has  $\$S$  on the stack with  $S$ , the start symbol of  $G$  on top, and  $w$  in the input buffer.  
■ The program that utilizes the predictive parsing table  $M$  to produce a parse for the input is shown here.

## Nonrecursive Predictive Parsing — *continued*

```
set ip to point to the first symbol of w;  
set X to the top stack symbol;  
while ( X ≠ $ ) { /* stack is not empty */  
    if ( X is a ) pop the stack and advance ip;  
    else if ( X is a terminal ) error();  
    else if ( M[X, a] is an error entry ) error();  
    else if ( M[X, a] = X → Y1Y2 … Yk ) {  
        output the production X → Y1Y2 … Yk;  
        pop the stack;  
        push Yk, Yk-1, …, Y1 onto the stack, with Y1 on top;  
    }  
    set X to the top stack symbol;  
}
```

Predictive parsing algorithm

# Example

- We consider grammar:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

- We have already seen its parsing table.

NON-TERMINAL	INPUT SYMBOL					
	<b>id</b>	+	*	(	)	\$
$E$	$E \rightarrow TE'$			$E \rightarrow TE'$		
$E'$		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
$T$	$T \rightarrow FT'$			$T \rightarrow FT'$		
$T'$		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
$F$	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

- On input **id + id \* id**, the nonrecursive predictive parser algorithm makes the sequence of moves,

MATCHED	STACK	INPUT	ACTION
	$E\$$	$id + id * id\$$	
	$TE'\$$	$id + id * id\$$	output $E \rightarrow TE'$
	$FT'E'\$$	$id + id * id\$$	output $T \rightarrow FT'$
	$id T'E'\$$	$id + id * id\$$	output $F \rightarrow id$
<b>id</b>	$T'E'\$$	$+ id * id\$$	match <b>id</b>
<b>id</b>	$E'\$$	$+ id * id\$$	output $T' \rightarrow \epsilon$
<b>id</b>	$+ TE'\$$	$+ id * id\$$	output $E' \rightarrow + TE'$
<b>id +</b>	$TE'\$$	$id * id\$$	match <b>+</b>
<b>id +</b>	$FT'E'\$$	$id * id\$$	output $T \rightarrow FT'$
<b>id +</b>	$id T'E'\$$	$id * id\$$	output $F \rightarrow id$
<b>id + id</b>	$T'E'\$$	$* id\$$	match <b>id</b>
<b>id + id</b>	$* FT'E'\$$	$* id\$$	output $T' \rightarrow * FT'$
<b>id + id *</b>	$FT'E'\$$	$id\$$	match <b>*</b>
<b>id + id *</b>	$id T'E'\$$	$id\$$	output $F \rightarrow id$
<b>id + id * id</b>	$T'E'\$$	$\$$	match <b>id</b>
<b>id + id * id</b>	$E'\$$	$\$$	output $T' \rightarrow \epsilon$
<b>id + id * id</b>	$\$$	$\$$	output $E' \rightarrow \epsilon$

Moves made by a predictive parser on input **id + id \* id**

- These moves correspond to a leftmost derivation,

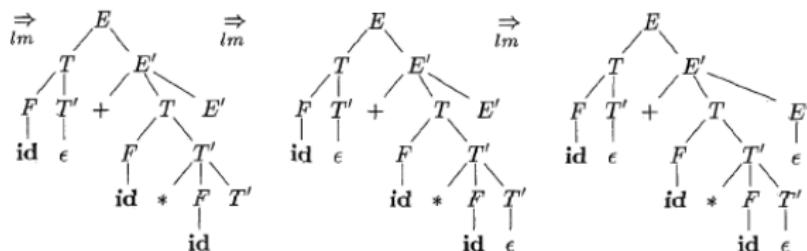
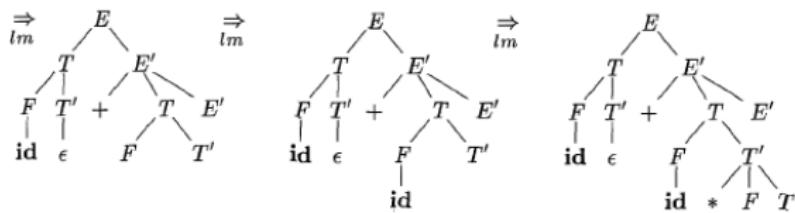
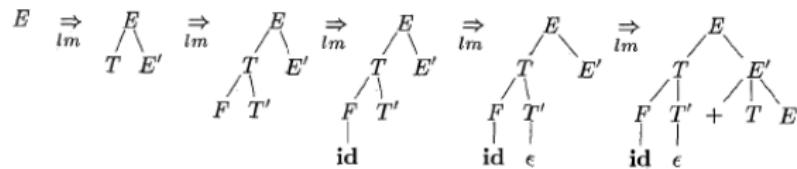
$$E \xrightarrow{lm} TE' \xrightarrow{lm} FT'E' \xrightarrow{lm} \mathbf{id}T'E' \xrightarrow{lm} \mathbf{id}E' \xrightarrow{lm} \mathbf{id} + TE' \xrightarrow{lm} \dots$$

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \rightarrow TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \rightarrow FT'$
	$\mathbf{id} T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $T' \rightarrow \epsilon$
$\mathbf{id}$	$+ TE'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $E' \rightarrow + TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id} +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \rightarrow FT'$
$\mathbf{id} +$	$\mathbf{id} T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$* FT'E'\$$	$* \mathbf{id}\$$	output $T' \rightarrow * FT'$
$\mathbf{id} + \mathbf{id} *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id} T'E'\$$	$\mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \rightarrow \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \rightarrow \epsilon$

Moves made by a predictive parser on input  $\mathbf{id} + \mathbf{id} * \mathbf{id}$

■ These moves correspond to a leftmost derivation,

$$E \xrightarrow{lm} TE' \xrightarrow{lm} FT'E' \xrightarrow{lm} \mathbf{id}T'E' \xrightarrow{lm} \mathbf{id}E' \xrightarrow{lm} \mathbf{id} + TE' \xrightarrow{lm} \dots$$



Top-down parse for  $\mathbf{id} + \mathbf{id} * \mathbf{id}$

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$}$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \rightarrow TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \rightarrow FT'$
$\mathbf{id}$	$T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $T' \rightarrow \epsilon$
$\mathbf{id}$	$+ TE'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $E' \rightarrow + TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id} +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \rightarrow FT'$
$\mathbf{id} +$	$\mathbf{id} T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$* FT'E'\$$	$* \mathbf{id}\$$	output $T' \rightarrow * FT'$
$\mathbf{id} + \mathbf{id} *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id} T'E'\$$	$\mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E\$$	$\$$	output $T' \rightarrow \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \rightarrow \epsilon$

Moves made by a predictive parser on input  $\mathbf{id} + \mathbf{id} * \mathbf{id}$

- Note that the sentential forms in this derivation correspond to the input that has already been matched (in column MATCHED) followed by the stack contents.

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$}$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \rightarrow TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \rightarrow FT'$
	$\mathbf{id} T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $T' \rightarrow \epsilon$
$\mathbf{id}$	$+ TE'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $E' \rightarrow + TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id} +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \rightarrow FT'$
$\mathbf{id} +$	$\mathbf{id} T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$* FT'E'\$$	$* \mathbf{id}\$$	output $T' \rightarrow * FT'$
$\mathbf{id} + \mathbf{id} *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id} T'E'\$$	$\mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \rightarrow \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \rightarrow \epsilon$

Moves made by a predictive parser on input  $\mathbf{id} + \mathbf{id} * \mathbf{id}$

- The matched input is shown only to highlight the correspondence.

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$}$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \rightarrow TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \rightarrow FT'$
	$\mathbf{id} T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $T' \rightarrow \epsilon$
$\mathbf{id}$	$+ TE'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $E' \rightarrow + TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id} +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \rightarrow FT'$
$\mathbf{id} +$	$\mathbf{id} T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$* FT'E'\$$	$* \mathbf{id}\$$	output $T' \rightarrow * FT'$
$\mathbf{id} + \mathbf{id} *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id} T'E'\$$	$\mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \rightarrow \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \rightarrow \epsilon$

Moves made by a predictive parser on input  $\mathbf{id} + \mathbf{id} * \mathbf{id}$

- For the same reason, the top of the stack is to the left.
- When we consider bottom-up parsing, it will be more natural to show the top of the stack to the right.

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$}$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \rightarrow TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \rightarrow FT'$
	$\mathbf{id} T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $T' \rightarrow \epsilon$
$\mathbf{id}$	$+ TE'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $E' \rightarrow + TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id} +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \rightarrow FT'$
$\mathbf{id} +$	$\mathbf{id} T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$* FT'E'\$$	$* \mathbf{id}\$$	output $T' \rightarrow * FT'$
$\mathbf{id} + \mathbf{id} *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id} T'E'\$$	$\mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \rightarrow \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \rightarrow \epsilon$

Moves made by a predictive parser on input  $\mathbf{id} + \mathbf{id} * \mathbf{id}$

- The input pointer points to the leftmost symbol of the string in the INPUT column.

## Example — *continued*

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	\$
$E$	$E \rightarrow TE'$				$E \rightarrow TE'$	
$E'$		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
$T$	$T \rightarrow FT'$			$T \rightarrow FT'$		
$T'$		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
$F$	$F \rightarrow id$			$F \rightarrow (E)$		

STACK	INPUT	ACTION
$E\$$	$id + id * id \$$	
$TE' \$$	$id + id * id \$$	output $E \rightarrow TE'$

## Example — *continued*

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	\$
$E$	$E \rightarrow TE'$				$E \rightarrow TE'$	
$E'$		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
$T$	$T \rightarrow FT'$				$T \rightarrow FT'$	
$T'$		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
$F$	$F \rightarrow id$			$F \rightarrow (E)$		

STACK	INPUT	ACTION
$TE' \$$ ↑	$id + id * id \$$ ↑	
$FT'E' \$$	$id + id * id \$$	output $T \rightarrow FT'$

## Example — *continued*

NON - TERMINAL	INPUT SYMBOL					
	<b>id</b>	+	*	(	)	\$
$E$	$E \rightarrow TE'$			$E \rightarrow TE'$		
$E'$		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
$T$	$T \rightarrow FT'$			$T \rightarrow FT'$		
$T'$		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
$F$	$F \rightarrow id$			$F \rightarrow (E)$		

STACK	INPUT	ACTION
$FT'E'\$$	$id + id * id \$$	
$idT'E'\$$	$id + id * id \$$	output $F \rightarrow id$

## Example — *continued*

STACK	INPUT	ACTION
$\mathbf{id}T'E'\$$ ↑	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$ ↑	match <b>id</b>

*Both are terminals and match. So, popped from the stack and input pointer advanced*

$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$ ↑
----------	--------------------------------------

## Example — *continued*

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	\$
$E$	$E \rightarrow TE'$			$E \rightarrow TE'$		
$E'$		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
$T$	$T \rightarrow FT'$			$T \rightarrow FT'$		
$T'$		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
$F$	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

STACK	INPUT	ACTION
$T'E'\$$	$+ \text{id} * \text{id} \$$	
$E'\$$	$+ \text{id} * \text{id} \$$	output $T' \rightarrow \epsilon$

## Example — *continued*

• • •

• • •

• • •

## Example — *continued*

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	\$
$E$	$E \rightarrow TE'$			$E \rightarrow TE'$		
$E'$		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
$T$	$T \rightarrow FT'$			$T \rightarrow FT'$		
$T'$		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
$F$	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

STACK	INPUT	ACTION
$E' \$$ ↑ \$	\$	\$ output $E' \rightarrow \epsilon$

## Example — *continued*

STACK	INPUT	ACTION
\$ ↑	\$ ↑	

*Both are \$, the parser halts and announces successful completion of parsing.*

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$}$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \rightarrow TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \rightarrow FT'$
	$\mathbf{id} T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $T' \rightarrow \epsilon$
$\mathbf{id}$	$+ TE'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $E' \rightarrow + TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id} +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \rightarrow FT'$
$\mathbf{id} +$	$\mathbf{id} T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$* FT'E'\$$	$* \mathbf{id}\$$	output $T' \rightarrow * FT'$
$\mathbf{id} + \mathbf{id} *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id} T'E'\$$	$\mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \rightarrow \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \rightarrow \epsilon$

Moves made by a predictive parser on input  $\mathbf{id} + \mathbf{id} * \mathbf{id}$

For a leftmost derivation the production rules in the ACTION column (outputs only) are to be used from top to bottom.

## Example — *continued*

$$\begin{array}{lcl} E & \xrightarrow[E \rightarrow TE']{} & TE' \\ & \xrightarrow[T \rightarrow FT']{} & FT'E' \\ & \xrightarrow[F \rightarrow \mathbf{id}]{} & \mathbf{id}T'E' \\ & \xrightarrow[T' \rightarrow \epsilon]{} & \mathbf{id}E' \\ & \xrightarrow[E' \rightarrow +TE']{} & \mathbf{id} + TE' \\ & \xrightarrow[T \rightarrow FT']{} & \mathbf{id} + FT'E' \\ & \xrightarrow[F \rightarrow \mathbf{id}]{} & \mathbf{id} + \mathbf{id}T'E' \end{array}$$

JT	ACTION
* id\$	
* id\$	output $E \rightarrow TE'$
* id\$	output $T \rightarrow FT'$
* id\$	output $F \rightarrow \mathbf{id}$
* id\$	match <b>id</b>
* id\$	output $T' \rightarrow \epsilon$
* id\$	output $E' \rightarrow + TE'$
* id\$	match $+$
* id\$	output $T \rightarrow FT'$
* id\$	output $F \rightarrow \mathbf{id}$
* id\$	match <b>id</b>
* id\$	output $T' \rightarrow * FT'$
id\$	match $*$
id\$	output $F \rightarrow \mathbf{id}$
\$	match <b>id</b>
\$	output $T' \rightarrow \epsilon$
\$	output $E' \rightarrow \epsilon$

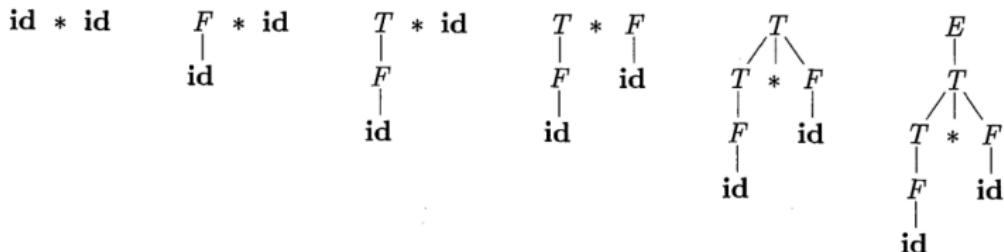
## Example — *continued*

$E \xrightarrow[E \rightarrow TE']{} \dots$   
 $E \xrightarrow[F \rightarrow \mathbf{id}]{}$   $\mathbf{id} + \mathbf{id}T'E'$   
 $E \xrightarrow[T' \rightarrow *FT']{} \mathbf{id} + \mathbf{id} * FT'E'$   
 $E \xrightarrow[F \rightarrow \mathbf{id}]{}$   $\mathbf{id} + \mathbf{id} * \mathbf{id}T'E'$   
 $E \xrightarrow[T' \rightarrow \epsilon]{}$   $\mathbf{id} + \mathbf{id} * \mathbf{id}E'$   
 $E \xrightarrow[E' \rightarrow \epsilon]{}$   $\mathbf{id} + \mathbf{id} * \mathbf{id}$

JT	ACTION
* id\$	
* id\$	output $E \rightarrow TE'$
* id\$	output $T \rightarrow FT'$
* id\$	output $F \rightarrow \mathbf{id}$
* id\$	match <b>id</b>
* id\$	output $T' \rightarrow \epsilon$
* id\$	output $E' \rightarrow + TE'$
* id\$	match <b>+</b>
* id\$	output $T \rightarrow FT'$
* id\$	output $F \rightarrow \mathbf{id}$
* id\$	match <b>id</b>
* id\$	output $T' \rightarrow * FT'$
id\$	match <b>*</b>
id\$	output $F \rightarrow \mathbf{id}$
\$	match <b>id</b>
\$	output $T' \rightarrow \epsilon$
\$	output $E' \rightarrow \epsilon$

- A bottom-up parse corresponds to the construction of a parse tree for an input string,
  - beginning at the leaves (the bottom) and
  - working up towards the root (the top).
- It is convenient to describe parsing as the process of building parse trees.
- Although a front end may in fact carry out a translation directly without building an explicit tree.

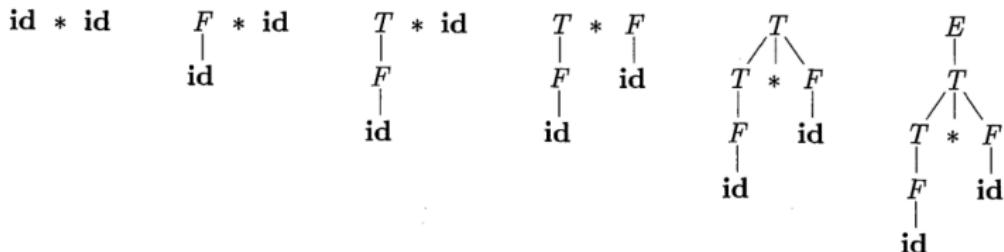
# Bottom-Up Parsing — *continued*



A bottom-up parse for **id \* id**

- The sequence of tree snapshots illustrates a bottom-up parse of the token stream **id \* id**, with respect to the expression grammar.

# Bottom-Up Parsing — *continued*



A bottom-up parse for **id \* id**

**id \* id**

$$E \rightarrow E + T \mid T$$

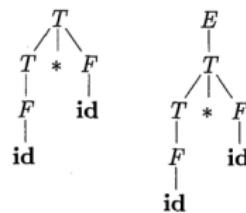
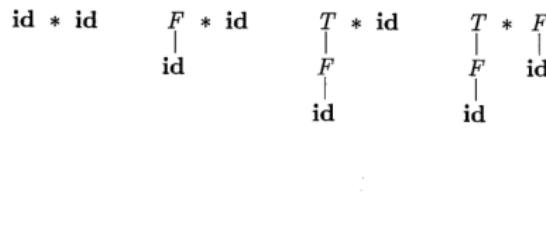
$$T \rightarrow T * F \mid F$$

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

- We can think of bottom-up parsing as the process of “reducing” a string  $w$  to the start symbol of the grammar.
- At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.
- The key decisions during bottom-up parsing are about,
  - when to reduce and
  - about what production to apply,as the parse proceeds.

- We can think of bottom-up parsing as the process of “reducing” a string  $w$  to the start symbol of the grammar.
- At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.
- The key decisions during bottom-up parsing are about,
  - when to reduce and
  - about what production to apply,as the parse proceeds.

# Example



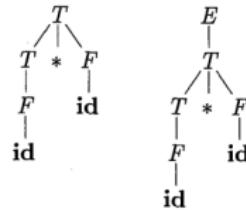
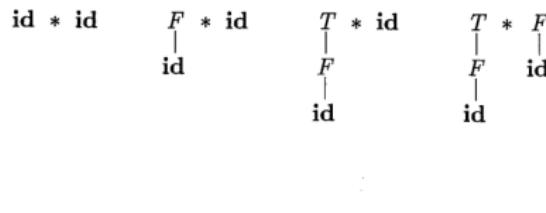
$E \rightarrow E + T \mid T$   
 $T \rightarrow T * F \mid F$   
 $F \rightarrow (E) \mid \mathbf{id}$

A bottom-up parse for **id \* id**

$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$

- Illustrates a sequence of reductions.
- The grammar is the expression grammar.

# Example



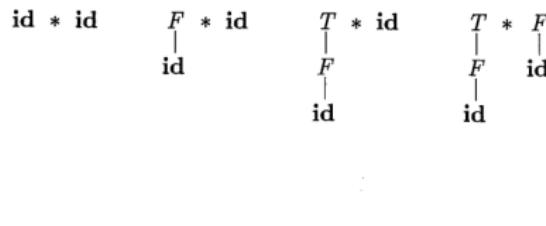
$E \rightarrow E + T \mid T$   
 $T \rightarrow T * F \mid F$   
 $F \rightarrow (E) \mid \mathbf{id}$

Bottom-up parser: A bottom-up parse for **id \* id**

$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$

- The reductions will be discussed in terms of the sequence of strings,
  - **id \* id**, **F \* id**, **T \* id**, **T \* F**, **T**, **E**.

# Example



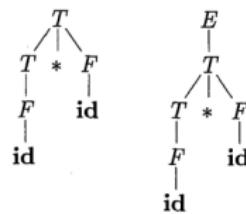
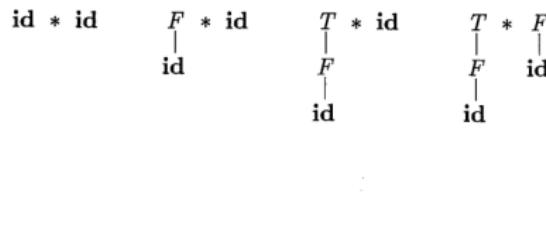
$E \rightarrow E + T \mid T$   
 $T \rightarrow T * F \mid F$   
 $F \rightarrow (E) \mid \mathbf{id}$

↓  
A bottom-up parse for **id \* id**

$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$

- The strings in this sequence are formed from the roots of all the subtrees in the snapshots.
- The sequence starts with the input string **id \* id**.

# Example



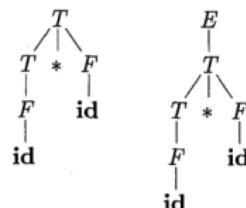
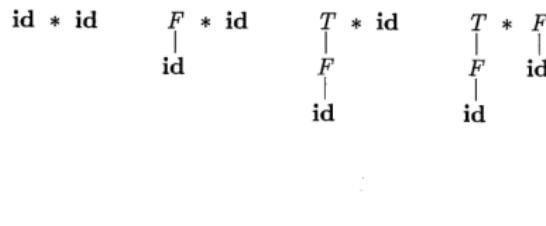
$E \rightarrow E + T \mid T$   
 $T \rightarrow T * F \mid F$   
 $F \rightarrow (E) \mid \mathbf{id}$

→ A bottom-up parse for **id \* id**

$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$

- The first reduction produces  **$F * \mathbf{id}$**  by reducing the leftmost **id** to **F**, using the production  $F \rightarrow \mathbf{id}$ .

# Example



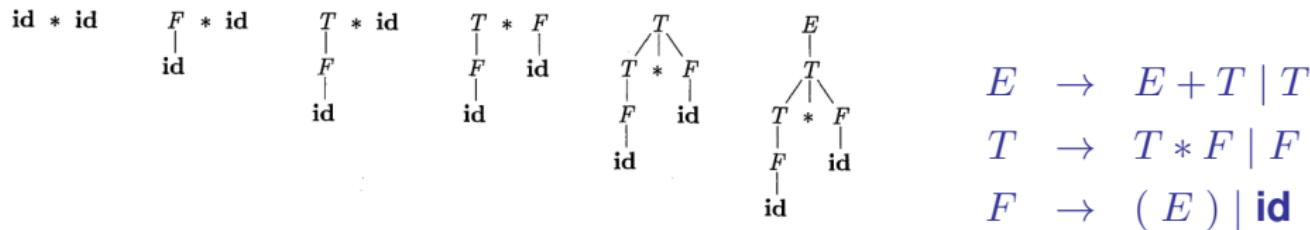
$E \rightarrow E + T \mid T$   
 $T \rightarrow T * F \mid F$   
 $F \rightarrow (E) \mid \mathbf{id}$

Bottom-up parser: A bottom-up parse for **id \* id**

$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$

- The second reduction produces  $T * \mathbf{id}$  by reducing  $F$  to  $T$ .

# Example



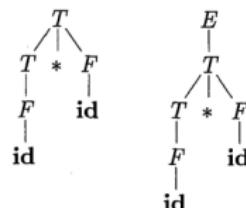
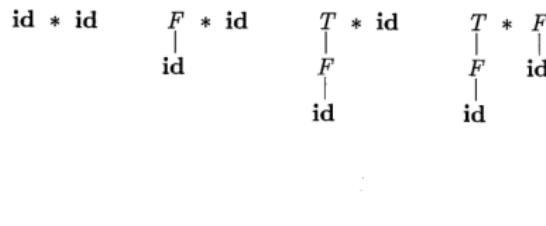
A bottom-up parse for **id \* id**

$E \Rightarrow T \Rightarrow T * F \Rightarrow T * id \Rightarrow F * id \Rightarrow id * id$

■ Now, we have a choice between,

- reducing the string  $T$ , which is the body of  $E \rightarrow T$ , and
- the string consisting of the second **id**, which is the body of  $F \rightarrow id$ .

# Example



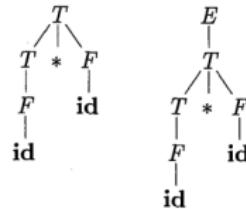
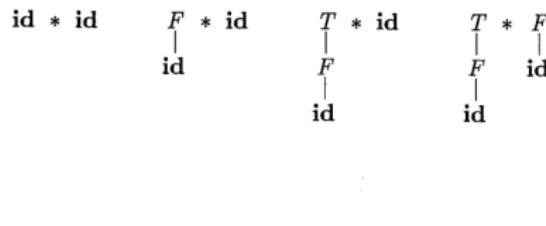
$E \rightarrow E + T \mid T$   
 $T \rightarrow T * F \mid F$   
 $F \rightarrow (E) \mid \mathbf{id}$

Bottom-up parser: A bottom-up parse for **id \* id**

$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$

- Rather than reduce  $T$  to  $E$ , the second **id** is reduced to  $F$ , resulting in the string  $T * F$ .

# Example



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

Bottom-up parser: A bottom-up parse for **id \* id**

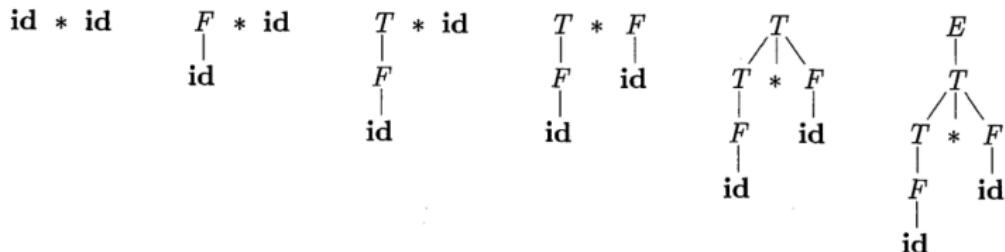
$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$

- This string then reduces to  $T$ .
- The parse completes with the reduction of  $T$  to the start symbol  $E$ .

## Reductions — *continued*

- By definition, a reduction is the reverse of a step in a derivation.
- Recall that in a derivation, a nonterminal in a sentential form is replaced by the body of one of its productions.
- The goal of bottom-up parsing is therefore to construct a derivation in reverse.

# Reductions — *continued*



A bottom-up parse for **id \* id**

- The following derivation corresponds to the parse,

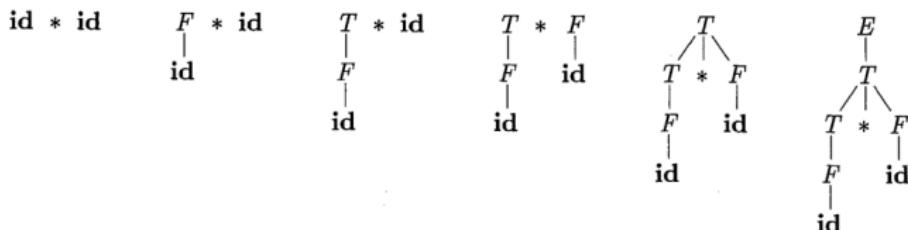
$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}.$$

- This derivation is in fact a *rightmost* derivation.

# Handle Pruning

- Bottom-up parsing during a left-to-right scan of the input constructs a rightmost derivation in reverse.
- Informally, a “handle” is a substring that matches the body of a production.
- And whose reduction represents one step along the reverse of a rightmost derivation.

# Handle Pruning — *continued*



RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\text{id}_1 * \text{id}_2$	$\text{id}_1$	$F \rightarrow \text{id}$
$F * \text{id}_2$	$F$	$T \rightarrow F$
$T * \text{id}_2$	$\text{id}_2$	$F \rightarrow \text{id}$
$T * F$	$T * F$	<del><math>E \rightarrow T * F</math></del>

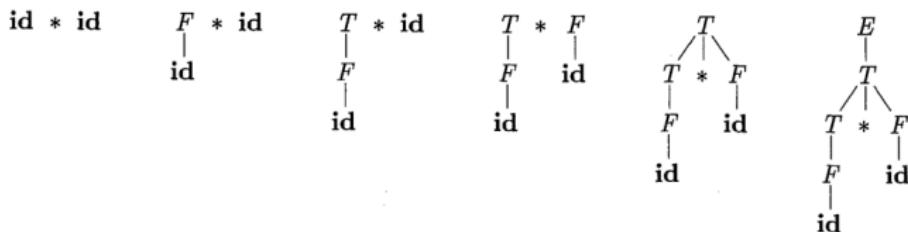
Handles during a parse of  $\text{id}_1 * \text{id}_2$

- Adding subscripts to the tokens **id** for clarity.

- The handles during the parse of  $\text{id}_1 * \text{id}_2$  according to the expression grammar.

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

# Handle Pruning — *continued*



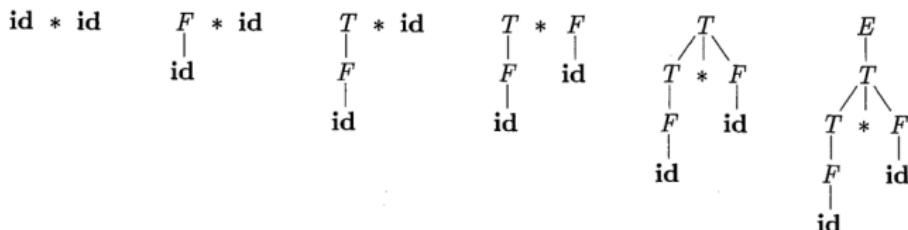
RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\text{id}_1 * \text{id}_2$	$\text{id}_1$	$F \rightarrow \text{id}$
$F * \text{id}_2$	$F$	$T \rightarrow F$
$T * \text{id}_2$	$\text{id}_2$	$F \rightarrow \text{id}$
$T * F$	$T * F$	<del><math>E \rightarrow T * F</math></del>

Handles during a parse of  $\text{id}_1 * \text{id}_2$

- Although  $T$  is the body of the production  $E \rightarrow T$ , the symbol  $T$  is not a handle in the sentential form  $T * \text{id}_2$ .

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

# Handle Pruning — *continued*



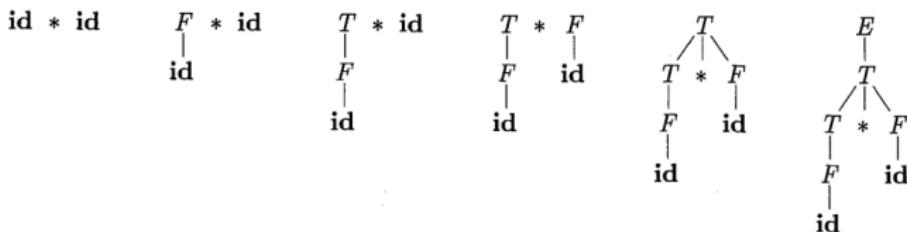
RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id_1 * id_2}$	$\mathbf{id_1}$	$F \rightarrow \mathbf{id}$
$F * \mathbf{id_2}$	$F$	$T \rightarrow F$
$T * \mathbf{id_2}$	$\mathbf{id_2}$	$F \rightarrow \mathbf{id}$
$T * F$	$T * F$	<del><math>E \rightarrow T * F</math></del>

Handles during a parse of  $\mathbf{id_1 * id_2}$

- If  $T$  were indeed replaced by  $E$ , we would get the string  $E * \mathbf{id_2}$ , which cannot be derived from the start symbol  $E$ .

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

# Handle Pruning — *continued*



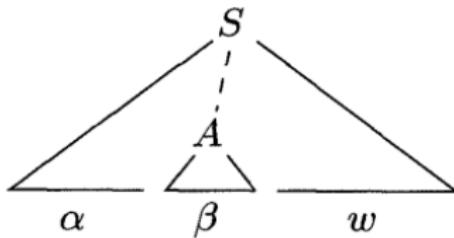
RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\text{id}_1 * \text{id}_2$	$\text{id}_1$	$F \rightarrow \text{id}$
$F * \text{id}_2$	$F$	$T \rightarrow F$
$T * \text{id}_2$	$\text{id}_2$	$F \rightarrow \text{id}$
$T * F$	$T * F$	<del><math>E \rightarrow T * F</math></del>

Handles during a parse of  $\text{id}_1 * \text{id}_2$

- Thus, the leftmost substring that matches the body of some production *need not* be a handle.

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

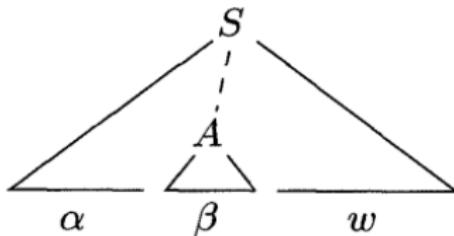
## Handle Pruning — *continued*



Wiggin 2.7.10 A handle  $A \rightarrow \beta$  in the parse tree for  $\alpha\beta w$

- Formally, if  $S \xrightarrow[\text{rm}]{*} \alpha A w \xrightarrow[\text{rm}]{*} \alpha \beta w$ , then production  $A \rightarrow \beta$ , in the position following  $\alpha$  is a handle of  $\alpha\beta w$ .

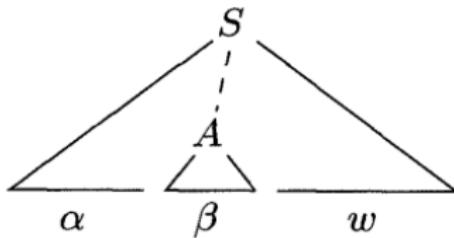
## Handle Pruning — *continued*



Wiggin 2.7.10 A handle  $A \rightarrow \beta$  in the parse tree for  $\alpha\beta w$

- Alternatively, a handle of a right-sentential form  $\gamma$  is a production  $A \rightarrow \beta$  and a position of  $\gamma$  where the string  $\beta$  may be found.
- Replacing  $\beta$  at that position by  $A$  produces the previous right-sentential form in a rightmost derivation of  $\gamma$ .

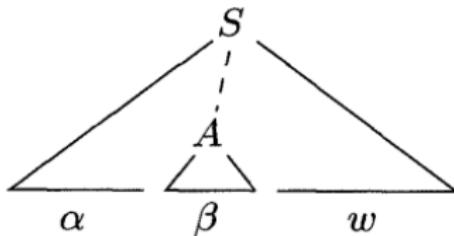
## Handle Pruning — *continued*



Wiggin 2.7.1 A handle  $A \rightarrow \beta$  in the parse tree for  $\alpha\beta w$

- Notice that the string  $w$  to the right of the handle must contain only terminal symbols.
- For convenience, we refer to the body  $\beta$  rather than  $A \rightarrow \beta$  as a handle.

## Handle Pruning — *continued*



*What is a handle?* A handle  $A \rightarrow \beta$  in the parse tree for  $\alpha\beta w$

- Note we say “a handle” rather than “the handle,” because the grammar could be ambiguous, with more than one rightmost derivation of  $\alpha\beta w$ .
- If a grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.

## Handle Pruning — *continued*

- A rightmost derivation in reverse can be obtained by “handle pruning.”
- That is, we start with a string of terminals  $w$  to be parsed.
- If  $w$  is a sentence of the grammar at hand, then let  $w = \gamma_n$ , where  $\gamma$  is the  $n$ th right-sentential form of some as yet unknown rightmost derivation,

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

## Handle Pruning — *continued*

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

- To reconstruct this derivation in reverse order, we locate the handle  $\beta_n$  in  $\gamma_n$ .
- And replace  $\beta_n$  by the head of the relevant production  $A_n \rightarrow \beta_n$  to obtain the previous right-sentential form  $\gamma_{n-1}$ .

## Handle Pruning — *continued*

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

- We then repeat this process.
- That is, we locate the handle  $\beta_{n-1}$  in  $\gamma_{n-1}$  and reduce this handle to obtain the right-sentential form  $\gamma_{n-2}$ .
- By continuing this process we produce a right-sentential form consisting only of the start symbol  $S$ .
- Then we halt and announce successful completion of parsing.
- The reverse of the sequence of productions used in the reductions is a rightmost derivation for the input string.

- Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols.
- And an input buffer holds the rest of the string to be parsed.
- The handle always appears at the top of the stack just before it is identified as the handle.

## Shift-Reduce Parsing — *continued*

- We use  $\$$  to mark the bottom of the stack and also the right end of the input.
- Conventionally, when discussing bottom-up parsing, we show the top of the stack on the right, rather than on the left as we did for top-down parsing.
- Initially, the stack is empty, and the string  $w$  is on the input, as follows:

STACK

$\$$

INPUT

$w\$$

## Shift-Reduce Parsing — *continued*

STACK	INPUT
$\$$	$w\$$

- During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string  $\beta$  of grammar symbols on top of the stack.
- It then reduces  $\beta$  to the head of the appropriate production.
- The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty.

STACK	INPUT
$\$S$	$\$$

## Shift-Reduce Parsing — *continued*

STACK	INPUT
$\$S$	$\$$

- Upon entering this configuration, the parser halts and announces successful completion of parsing.

## Shift-Reduce Parsing — *continued*

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

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

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

STACK	INPUT	ACTION
\$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$ \mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \rightarrow \mathbf{id}$
$\$ F$	$* \mathbf{id}_2 \$$	reduce by $T \rightarrow F$
$\$ T$	$* \mathbf{id}_2 \$$	shift
$\$ T *$	$\mathbf{id}_2 \$$	shift
$\$ T * \mathbf{id}_2$	\$	reduce by $F \rightarrow \mathbf{id}$
$\$ T * F$	\$	reduce by $T \rightarrow T * F$
$\$ T$	\$	reduce by $E \rightarrow T$
$\$ E$	\$	accept

Configurations of a shift-reduce parser on input  $\mathbf{id}_1 * \mathbf{id}_2$

- While the primary operations are shift and reduce, there are actually four possible actions a shift-reduce parser can make:

*Shift.* Shift the next input symbol onto the top of the stack.

*Reduce.* The right end of the string to be reduced must be at the top of the stack.

- Locate the left end of the string within the stack.
- And decide with what nonterminal to replace the string.

*Accept.* Announce successful completion of parsing.

*Error.* Discover a syntax error and call an error recovery routine.

- While the primary operations are shift and reduce, there are actually four possible actions a shift-reduce parser can make:

*Shift.* Shift the next input symbol onto the top of the stack.

*Reduce.* The right end of the string to be reduced must be at the top of the stack.

- Locate the left end of the string within the stack.
- And decide with what nonterminal to replace the string.

*Accept.* Announce successful completion of parsing.

*Error.* Discover a syntax error and call an error recovery routine.

# Shift-Reduce Parsing — *continued*

- While the primary operations are shift and reduce, there are actually four possible actions a shift-reduce parser can make:

*Shift.* Shift the next input symbol onto the top of the stack.

*Reduce.* The right end of the string to be reduced must be at the top of the stack.

- Locate the left end of the string within the stack.
- And decide with what nonterminal to replace the string.

*Accept.* Announce successful completion of parsing.

*Error.* Discover a syntax error and call an error recovery routine.

# Shift-Reduce Parsing — *continued*

- While the primary operations are shift and reduce, there are actually four possible actions a shift-reduce parser can make:

*Shift.* Shift the next input symbol onto the top of the stack.

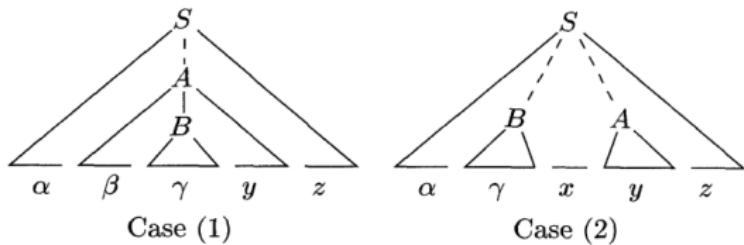
*Reduce.* The right end of the string to be reduced must be at the top of the stack.

- Locate the left end of the string within the stack.
- And decide with what nonterminal to replace the string.

*Accept.* Announce successful completion of parsing.

*Error.* Discover a syntax error and call an error recovery routine.

# Shift-Reduce Parsing — *continued*



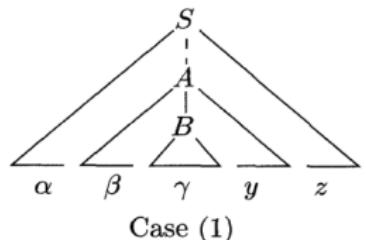
Cases for two successive steps of a rightmost derivation

$$\begin{array}{lcl} S & \rightarrow & \alpha Az \mid \alpha BxAz \\ A & \rightarrow & \beta By \mid y \\ B & \rightarrow & \gamma \end{array}$$

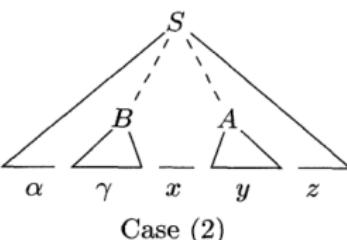
*(the productions are indicated in a bit loose manner)*

- The use of a stack in shift-reduce parsing is justified by an important fact.
- The handle will always eventually appear on top of the stack, never inside.

# Shift-Reduce Parsing — *continued*



Case (1)



Case (2)

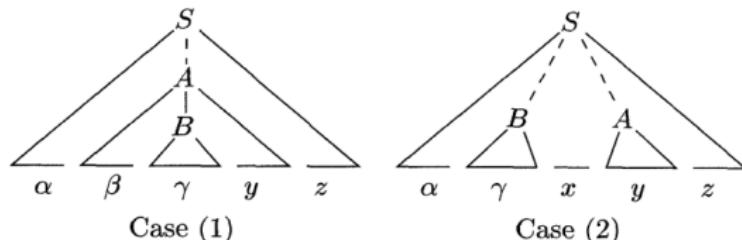
$$\begin{array}{lcl} S & \rightarrow & \alpha Az \mid \alpha BxAz \\ A & \rightarrow & \beta By \mid y \\ B & \rightarrow & \gamma \end{array}$$

(the productions are indicated in a bit loose manner)

Cases for two successive steps of a rightmost derivation

- This fact can be shown by considering the possible forms of two successive steps in any rightmost derivation.

# Shift-Reduce Parsing — *continued*



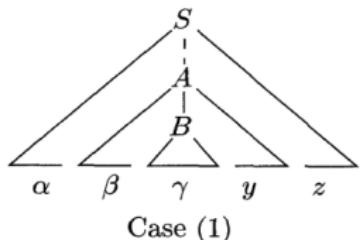
Cases for two successive steps of a rightmost derivation

$$\begin{array}{lcl} S & \rightarrow & \alpha Az \mid \alpha BxAz \\ A & \rightarrow & \beta By \mid y \\ B & \rightarrow & \gamma \end{array}$$

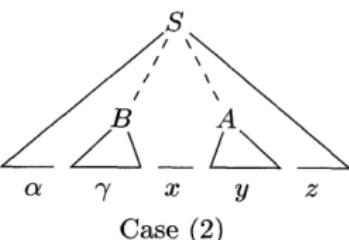
(the productions are indicated in a bit loose manner)

- In case (1),  $A$  is replaced by  $\beta By$ .
- Then the rightmost nonterminal  $B$  in the body  $\beta By$  is replaced by  $\gamma$ .

# Shift-Reduce Parsing — *continued*



Case (1)



Case (2)

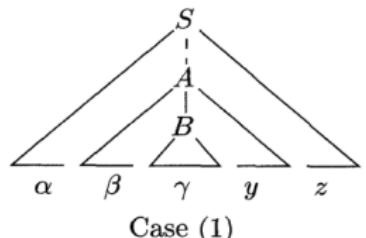
$$\begin{array}{lcl} S & \rightarrow & \alpha Az \mid \alpha BxAz \\ A & \rightarrow & \beta By \mid y \\ B & \rightarrow & \gamma \end{array}$$

(the productions are indicated in a bit loose manner)

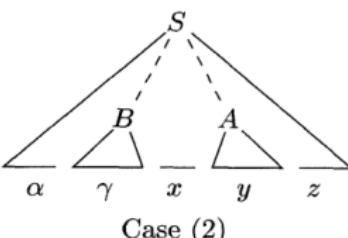
Cases for two successive steps of a rightmost derivation

- In case (2),  $A$  is again expanded first, but this time the body is a string  $y$  of terminals only.
- The next rightmost nonterminal  $B$  will be somewhere to the left of  $y$ .

# Shift-Reduce Parsing — *continued*



Case (1)



Case (2)

$$\begin{array}{lcl} S & \rightarrow & \alpha Az \mid \alpha BxAz \\ A & \rightarrow & \beta By \mid y \\ B & \rightarrow & \gamma \end{array}$$

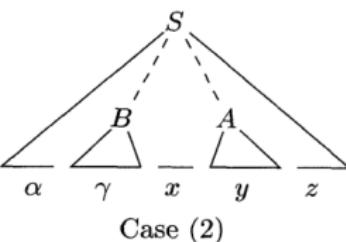
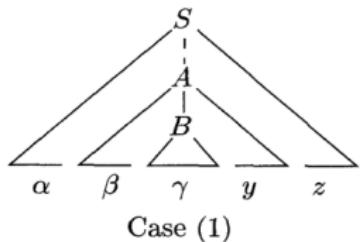
(the productions are indicated in a bit loose manner)

Cases for two successive steps of a rightmost derivation

## ■ In other words:

$$(1) \quad S \xrightarrow[\text{rm}]{*} \alpha Az \xrightarrow[\text{rm}]{*} \alpha \beta Byz \xrightarrow[\text{rm}]{*} \alpha \beta \gamma yz$$

$$(2) \quad S \xrightarrow[\text{rm}]{*} \alpha BxAz \xrightarrow[\text{rm}]{*} \alpha Bxyz \xrightarrow[\text{rm}]{*} \alpha \gamma xyz$$



$$\begin{aligned}
 S &\rightarrow \alpha Az \mid \alpha BxAz \\
 A &\rightarrow \beta By \mid y \\
 B &\rightarrow \gamma
 \end{aligned}$$

(the productions are indicated in a bit loose manner)

$$(1) \ S \xrightarrow[\text{rm}]{*} \alpha Az \xrightarrow[\text{rm}]{*} \alpha \beta Byz \xrightarrow[\text{rm}]{*} \alpha \beta \gamma yz$$

$$(2) \ S \xrightarrow[\text{rm}]{*} \alpha BxAz \xrightarrow[\text{rm}]{*} \alpha Bxyz \xrightarrow[\text{rm}]{*} \alpha \gamma xyz$$

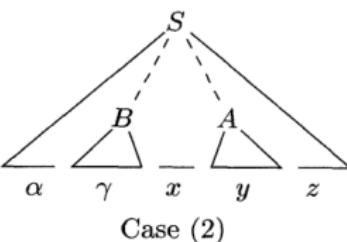
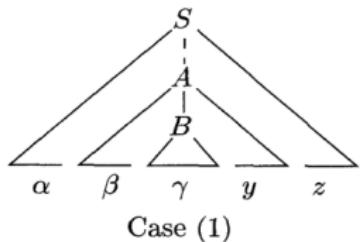
- Consider case (1) in reverse, where a shift-reduce parser has just reached the configuration,

STACK

$\$ \alpha \beta \gamma$

INPUT

$yz\$$



$$\begin{aligned}
 S &\rightarrow \alpha Az \mid \alpha BxAz \\
 A &\rightarrow \beta By \mid y \\
 B &\rightarrow \gamma
 \end{aligned}$$

(the productions are indicated in a bit loose manner)

$$(1) \ S \xrightarrow[\text{rm}]{*} \alpha Az \xrightarrow[\text{rm}]{*} \alpha \beta Byz \xrightarrow[\text{rm}]{*} \alpha \beta \gamma yz$$

$$(2) \ S \xrightarrow[\text{rm}]{*} \alpha BxAz \xrightarrow[\text{rm}]{*} \alpha Bxyz \xrightarrow[\text{rm}]{*} \alpha \gamma xyz$$

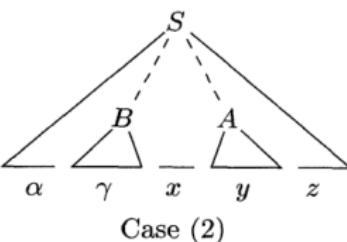
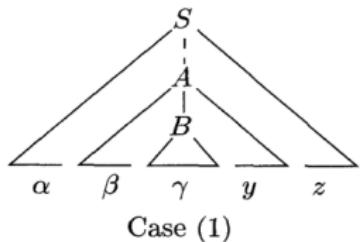
STACK  
\$\alpha \beta \gamma

INPUT  
yz\$

- The parser reduces the handle  $\gamma$  to  $B$  to reach the configuration,

STACK  
\$\alpha \beta B

INPUT  
yz\$



$$\begin{aligned}
 S &\rightarrow \alpha Az \mid \alpha BxAz \\
 A &\rightarrow \beta By \mid y \\
 B &\rightarrow \gamma
 \end{aligned}$$

(the productions are indicated in a bit loose manner)

$$(1) \ S \xrightarrow[\text{rm}]{*} \alpha Az \xrightarrow[\text{rm}]{*} \alpha \beta Byz \xrightarrow[\text{rm}]{*} \alpha \beta \gamma yz$$

$$(2) \ S \xrightarrow[\text{rm}]{*} \alpha BxAz \xrightarrow[\text{rm}]{*} \alpha Bxyz \xrightarrow[\text{rm}]{*} \alpha \gamma xyz$$

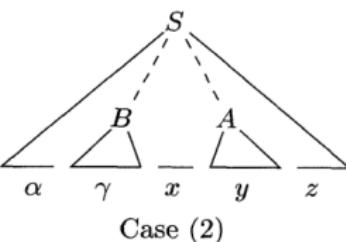
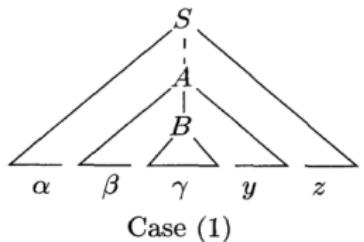
STACK  
\$\alpha \beta **B**

INPUT  
yz\$

- The parser can now shift the string  $y$  onto the stack by a sequence of zero or more shift moves to reach the configuration,

STACK  
\$\alpha **βBy**

INPUT  
z\$



$$\begin{aligned}
 S &\rightarrow \alpha Az \mid \alpha BxAz \\
 A &\rightarrow \beta By \mid y \\
 B &\rightarrow \gamma
 \end{aligned}$$

(the productions are indicated in a bit loose manner)

$$(1) \ S \xrightarrow[\text{rm}]{*} \alpha Az \xrightarrow[\text{rm}]{*} \alpha \beta Byz \xrightarrow[\text{rm}]{*} \alpha \beta \gamma yz$$

$$(2) \ S \xrightarrow[\text{rm}]{*} \alpha BxAz \xrightarrow[\text{rm}]{*} \alpha Bxyz \xrightarrow[\text{rm}]{*} \alpha \gamma xyz$$

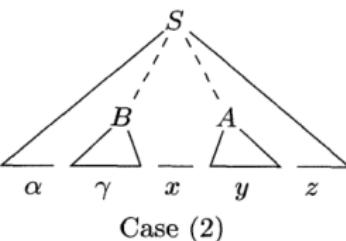
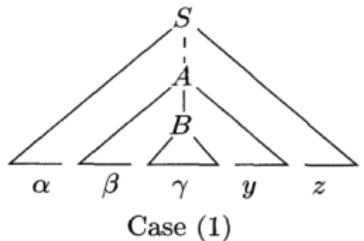
STACK  
 $\$ \alpha \beta By$

INPUT  
 $z \$$

- With the handle  $\beta By$  on top of the stack, and it gets reduced to  $A$ .

STACK  
 $\$ \alpha A$

INPUT  
 $z \$$



$$\begin{aligned}
 S &\rightarrow \alpha Az \mid \alpha BxAz \\
 A &\rightarrow \beta By \mid y \\
 B &\rightarrow \gamma
 \end{aligned}$$

(the productions are indicated in a bit loose manner)

$$(1) \ S \xrightarrow[\text{rm}]{*} \alpha Az \xrightarrow[\text{rm}]{*} \alpha \beta Byz \xrightarrow[\text{rm}]{*} \alpha \beta \gamma yz$$

$$(2) \ S \xrightarrow[\text{rm}]{*} \alpha BxAz \xrightarrow[\text{rm}]{*} \alpha Bxyz \xrightarrow[\text{rm}]{*} \alpha \gamma xyz$$

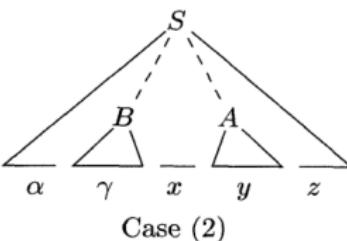
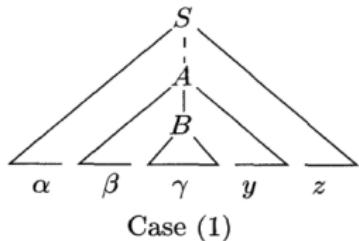
- Now consider case (2).
- In configuration the handle  $\gamma$  is on top of the stack.

STACK

$\$ \alpha \gamma$

INPUT

$xyz \$$



$$\begin{aligned}
 S &\rightarrow \alpha Az \mid \alpha BxAz \\
 A &\rightarrow \beta By \mid y \\
 B &\rightarrow \gamma
 \end{aligned}$$

(the productions are indicated in a bit loose manner)

$$(1) \ S \xrightarrow[\text{rm}]{*} \alpha Az \xrightarrow[\text{rm}]{*} \alpha \beta Byz \xrightarrow[\text{rm}]{*} \alpha \beta \gamma yz$$

$$(2) \ S \xrightarrow[\text{rm}]{*} \alpha BxAz \xrightarrow[\text{rm}]{*} \alpha Bxyz \xrightarrow[\text{rm}]{*} \alpha \gamma xyz$$

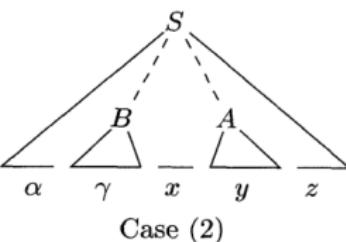
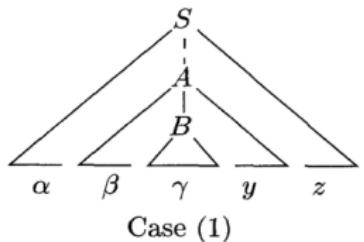
STACK  
\$\alpha\gamma\$

INPUT  
xyz\$

■ After reducing the handle  $\gamma$  to  $B$ , the configuration is:

STACK  
\$\alpha B\$

INPUT  
xyz\$



$$\begin{aligned}
 S &\rightarrow \alpha Az \mid \alpha BxAz \\
 A &\rightarrow \beta By \mid y \\
 B &\rightarrow \gamma
 \end{aligned}$$

(the productions are indicated in a bit loose manner)

$$(1) \ S \xrightarrow[\text{rm}]{*} \alpha Az \xrightarrow[\text{rm}]{*} \alpha \beta Byz \xrightarrow[\text{rm}]{*} \alpha \beta \gamma yz$$

$$(2) \ S \xrightarrow[\text{rm}]{*} \alpha BxAz \xrightarrow[\text{rm}]{*} \alpha Bxyz \xrightarrow[\text{rm}]{*} \alpha \gamma xyz$$

STACK

$\$ \alpha B$

INPUT

$xyz\$$

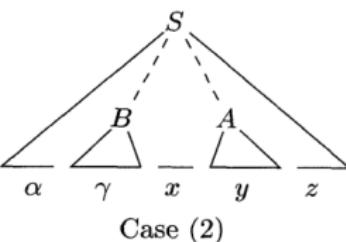
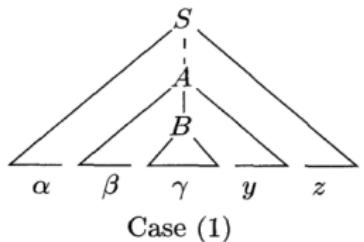
- The parser can shift the string  $xy$  to get the next handle  $y$  on top of the stack, ready to be reduced to  $A$ :

STACK

$\$ \alpha Bx y$

INPUT

$z\$$



$$\begin{aligned}
 S &\rightarrow \alpha Az \mid \alpha BxAz \\
 A &\rightarrow \beta By \mid y \\
 B &\rightarrow \gamma
 \end{aligned}$$

(the productions are indicated in a bit loose manner)

$$(1) \ S \xrightarrow[\text{rm}]{*} \alpha Az \xrightarrow[\text{rm}]{*} \alpha \beta Byz \xrightarrow[\text{rm}]{*} \alpha \beta \gamma yz$$

$$(2) \ S \xrightarrow[\text{rm}]{*} \alpha BxAz \xrightarrow[\text{rm}]{*} \alpha Bxyz \xrightarrow[\text{rm}]{*} \alpha \gamma xyz$$

- In both cases, after making a reduction the parser had to shift zero or more symbols to get the next handle onto the stack.
- It *never* had to go into the stack to find the handle.

# Conflicts During Shift-Reduce Parsing

- There are context-free grammars for which shift-reduce parsing *cannot* be used.
- Every shift-reduce parser for such a grammar can reach a configuration.

- Here, the parser, knowing the entire stack contents and the next input symbol,
  - cannot decide whether to shift or to reduce (a *shift/reduce* conflict),
  - or cannot decide which of several reductions to make (a *reduce/reduce* conflict).

- The  $k$  in  $\text{LR}(k)$  refers to the number of symbols of lookahead on the input.
- Grammars used in compiling usually fall in the  $\text{LR}(1)$  class, with one symbol of lookahead at most.

# Example

- An ambiguous grammar can never be LR.
- Consider the dangling-else grammar.

```
stmt → if expr then stmt
      | if expr then stmt else stmt
      | other
```

## Example — *continued*

```
stmt → if expr then stmt
      | if expr then stmt else stmt
      | other
```

- We have a shift-reduce parser in configuration,

STACK	INPUT
... if expr then stmt	else ... \$

- We cannot tell whether “**if expr then stmt**” is the handle, no matter what appears below it on the stack.

## Example — *continued*

*stmt* → **if** *expr then stmt*  
| **if** *expr then stmt* **else** *stmt*  
| **other**

STACK  
... **if** *expr then stmt*

INPUT  
**else** ... \$

- Here there is a shift/reduce conflict.

## Example — *continued*

*stmt* → **if** *expr then stmt*  
| **if** *expr then stmt* **else** *stmt*  
| **other**

STACK	INPUT
... <b>if</b> <i>expr then stmt</i>	<b>else</b> ... \$

- Depending on what follows the **else** on the input,
  - It might be correct to reduce **if** *expr then stmt* to *stmt*.
  - Or it might be correct to shift **else** and then to look for another *stmt* to complete the alternative, **if** *expr then stmt* **else** *stmt*.

## Example — *continued*

```
stmt → if expr then stmt
      | if expr then stmt else stmt
      | other
```

STACK	INPUT
... if expr then stmt	else ... \$

- Note that shift-reduce parsing can be adapted to parse certain ambiguous grammars, such as the if-then-else grammar above.
- If we resolve the shift/reduce conflict on **else** in favor of shifting, the parser will behave as we expect, associating each **else** with the previous unmatched **then**.

## Conflicts During Shift-Reduce Parsing — *continued*

- Another common setting for conflicts occurs when we know we have a handle.
- But the stack contents and the next input symbol are insufficient to determine which production should be used in a reduction.

# Example

- Suppose we have a lexical analyzer that returns the token name **id** for all names, regardless of their type.
- Suppose also that our language,
  - invokes procedures by giving their names, with parameters surrounded by parentheses,
  - and that arrays are referenced by the same syntax.

## Example — *continued*

- The translation of indices in array references and parameters in procedure calls are different.
- We want to use different productions to generate lists of actual parameters and indices.

## Example — *continued*

- Our grammar might therefore have (among others) productions such as those shown.

- (1)  $stmt \rightarrow id ( parameter\_list )$
- (2)  $stmt \rightarrow expr := expr$
- (3)  $parameter\_list \rightarrow parameter\_list, parameter$
- (4)  $parameter\_list \rightarrow parameter$
- (5)  $parameter \rightarrow id$
- (6)  $expr \rightarrow id ( expr\_list )$
- (7)  $expr \rightarrow id$
- (8)  $expr\_list \rightarrow expr\_list, expr$
- (9)  $expr\_list \rightarrow expr$

Productions involving procedure calls and array references

## Example — *continued*

**p(i, j)**

(1)	<i>stmt</i>	→	<b>id</b> ( <i>parameter_list</i> )
(2)	<i>stmt</i>	→	<i>expr</i> := <i>expr</i>
(3)	<i>parameter_list</i>	→	<i>parameter_list</i> , <i>parameter</i>
(4)	<i>parameter_list</i>	→	<i>parameter</i>
(5)	<i>parameter</i>	→	<b>id</b>
(6)	<i>expr</i>	→	<b>id</b> ( <i>expr_list</i> )
(7)	<i>expr</i>	→	<b>id</b>
(8)	<i>expr_list</i>	→	<i>expr_list</i> , <i>expr</i>
(9)	<i>expr_list</i>	→	<i>expr</i>

Productions involving procedure calls and array references

- A statement beginning with **p(i, j)** would appear as the token stream **id(id, id)** to the parser.

## Example — *continued*

**p(i, j)**

(1)	<i>stmt</i>	→	<b>id</b> ( <i>parameter_list</i> )
(2)	<i>stmt</i>	→	<i>expr</i> := <i>expr</i>
(3)	<i>parameter_list</i>	→	<i>parameter_list</i> , <i>parameter</i>
(4)	<i>parameter_list</i>	→	<i>parameter</i>
(5)	<i>parameter</i>	→	<b>id</b>
(6)	<i>expr</i>	→	<b>id</b> ( <i>expr_list</i> )
(7)	<i>expr</i>	→	<b>id</b>
(8)	<i>expr_list</i>	→	<i>expr_list</i> , <i>expr</i>
(9)	<i>expr_list</i>	→	<i>expr</i>

Productions involving procedure calls and array references

- After shifting the first three tokens onto the stack, a shift-reduce parser would be in configuration,

STACK  
... **id** ( **id**

INPUT  
, **id** ) ...

## Example — *continued*

STACK  
... **id** ( **id**

INPUT  
, **id** ) ...

(1)	<i>stmt</i>	→	<b>id</b> ( <i>parameter_list</i> )
(2)	<i>stmt</i>	→	<i>expr</i> := <i>expr</i>
(3)	<i>parameter_list</i>	→	<i>parameter_list</i> , <i>parameter</i>
(4)	<i>parameter_list</i>	→	<i>parameter</i>
(5)	<i>parameter</i>	→	<b>id</b>
(6)	<i>expr</i>	→	<b>id</b> ( <i>expr_list</i> )
(7)	<i>expr</i>	→	<b>id</b>
(8)	<i>expr_list</i>	→	<i>expr_list</i> , <i>expr</i>
(9)	<i>expr_list</i>	→	<i>expr</i>

Figure 10.6: Productions involving procedure calls and array references

- It is evident that the **id** on top of the stack must be reduced, but by which production?

## Example — *continued*

STACK  
... **id** ( **id**

INPUT  
, **id** ) ...

- (1)  $stmt \rightarrow id ( parameter\_list )$
- (2)  $stmt \rightarrow expr := expr$
- (3)  $parameter\_list \rightarrow parameter\_list, parameter$
- (4)  $parameter\_list \rightarrow parameter$
- (5)  $parameter \rightarrow id$
- (6)  $expr \rightarrow id ( expr\_list )$
- (7)  $expr \rightarrow id$
- (8)  $expr\_list \rightarrow expr\_list, expr$
- (9)  $expr\_list \rightarrow expr$

Figure 10.6: Productions involving procedure calls and array references

- The correct choice is production (5) if  $p$  is a procedure, but production (7) if  $p$  is an array.

## Example — *continued*

	STACK	INPUT
	...	
	<b>id</b> ( <b>id</b>	, <b>id</b> ) ...
(1)	<i>stmt</i>	→ <b>id</b> ( <i>parameter_list</i> )
(2)	<i>stmt</i>	→ <i>expr</i> := <i>expr</i>
(3)	<i>parameter_list</i>	→ <i>parameter_list</i> , <i>parameter</i>
(4)	<i>parameter_list</i>	→ <i>parameter</i>
(5)	<i>parameter</i>	→ <b>id</b>
(6)	<i>expr</i>	→ <b>id</b> ( <i>expr_list</i> )
(7)	<i>expr</i>	→ <b>id</b>
(8)	<i>expr_list</i>	→ <i>expr_list</i> , <i>expr</i>
(9)	<i>expr_list</i>	→ <i>expr</i>

### Figure 10.6: Productions involving procedure calls and array references

- The stack does not tell which information in the symbol table obtained from the declaration of *p* must be used.

## Example — *continued*

STACK  
... **id** ( **id**

INPUT  
, **id** ) ...

- (1)  $stmt \rightarrow id ( parameter\_list )$
- (2)  $stmt \rightarrow expr := expr$
- (3)  $parameter\_list \rightarrow parameter\_list, parameter$
- (4)  $parameter\_list \rightarrow parameter$
- (5)  $parameter \rightarrow id$
- (6)  $expr \rightarrow id ( expr\_list )$
- (7)  $expr \rightarrow id$
- (8)  $expr\_list \rightarrow expr\_list, expr$
- (9)  $expr\_list \rightarrow expr$

Figure 10.6: Productions involving procedure calls and array references

- One solution is to change the token **id** in production (1) to **procid**.

## Example — *continued*

STACK  
... **id** ( **id**

INPUT  
, **id** ) ...

- (1)  $stmt \rightarrow id ( parameter\_list )$
- (2)  $stmt \rightarrow expr := expr$
- (3)  $parameter\_list \rightarrow parameter\_list, parameter$
- (4)  $parameter\_list \rightarrow parameter$
- (5)  $parameter \rightarrow id$
- (6)  $expr \rightarrow id ( expr\_list )$
- (7)  $expr \rightarrow id$
- (8)  $expr\_list \rightarrow expr\_list, expr$
- (9)  $expr\_list \rightarrow expr$

### Figure 10.6: Productions involving procedure calls and array references

- To use a more sophisticated lexical analyzer that returns the token name **procid** when it recognizes a lexeme that is the name of a procedure.

## Example — *continued*

STACK  
... **id** ( **id**

INPUT  
, **id** ) ...

- (1)  $stmt \rightarrow id ( parameter\_list )$
- (2)  $stmt \rightarrow expr := expr$
- (3)  $parameter\_list \rightarrow parameter\_list, parameter$
- (4)  $parameter\_list \rightarrow parameter$
- (5)  $parameter \rightarrow id$
- (6)  $expr \rightarrow id ( expr\_list )$
- (7)  $expr \rightarrow id$
- (8)  $expr\_list \rightarrow expr\_list, expr$
- (9)  $expr\_list \rightarrow expr$

Figure 10.6: Productions involving procedure calls and array references

- Doing so would require the lexical analyzer to consult the symbol table before returning a token.

## Example — *continued*

(1)	<i>stmt</i>	→	<b>id</b> ( <i>parameter_list</i> )
(2)	<i>stmt</i>	→	<i>expr</i> := <i>expr</i>
(3)	<i>parameter_list</i>	→	<i>parameter_list</i> , <i>parameter</i>
(4)	<i>parameter_list</i>	→	<i>parameter</i>
(5)	<i>parameter</i>	→	<b>id</b>
(6)	<i>expr</i>	→	<b>id</b> ( <i>expr_list</i> )
(7)	<i>expr</i>	→	<b>id</b>
(8)	<i>expr_list</i>	→	<i>expr_list</i> , <i>expr</i>
(9)	<i>expr_list</i>	→	<i>expr</i>

### Productions involving procedure calls and array references

- If we made this modification, then on processing  $p(i, j)$  the parser would be either in the configuration,

STACK	INPUT
... <b>procid</b> ( <b>id</b>	, <b>id</b> ) ...

## Example — *continued*

- (1)  $stmt \rightarrow id ( parameter\_list )$
- (2)  $stmt \rightarrow expr := expr$
- (3)  $parameter\_list \rightarrow parameter\_list, parameter$
- (4)  $parameter\_list \rightarrow parameter$
- (5)  $parameter \rightarrow id$
- (6)  $expr \rightarrow id ( expr\_list )$
- (7)  $expr \rightarrow id$
- (8)  $expr\_list \rightarrow expr\_list, expr$
- (9)  $expr\_list \rightarrow expr$

Figure 10.10: Productions involving procedure calls and array references

- Or in the configuration,

STACK  
... **id ( id**

INPUT  
, **id ) ...**

## Example — *continued*

	STACK	INPUT
	... <b>procid</b> ( <b>id</b>	, <b>id</b> ) ...
	... <b>id</b> ( <b>id</b>	, <b>id</b> ) ...
(1)	<i>stmt</i>	→ <b>id</b> ( <i>parameter-list</i> )
(2)	<i>stmt</i>	→ <i>expr</i> := <i>expr</i>
(3)	<i>parameter-list</i>	→ <i>parameter-list</i> , <i>parameter</i>
(4)	<i>parameter-list</i>	→ <i>parameter</i>
(5)	<i>parameter</i>	→ <b>id</b>
(6)	<i>expr</i>	→ <b>id</b> ( <i>expr-list</i> )
(7)	<i>expr</i>	→ <b>id</b>
(8)	<i>expr-list</i>	→ <i>expr-list</i> , <i>expr</i>
(9)	<i>expr-list</i>	→ <i>expr</i>

- In the former case, we choose reduction by production (5).

## Example — *continued*

	STACK	INPUT
	... <b>procid</b> ( <b>id</b>	, <b>id</b> ) ...
	... <b>id</b> ( <b>id</b>	, <b>id</b> ) ...
(1)	<i>stmt</i> → <b>id</b> ( <i>parameter-list</i> )	
(2)	<i>stmt</i> → <i>expr</i> := <i>expr</i>	
(3)	<i>parameter-list</i> → <i>parameter-list</i> , <i>parameter</i>	
(4)	<i>parameter-list</i> → <i>parameter</i>	
(5)	<i>parameter</i> → <b>id</b>	
(6)	<i>expr</i> → <b>id</b> ( <i>expr-list</i> )	
(7)	<i>expr</i> → <b>id</b>	
(8)	<i>expr-list</i> → <i>expr-list</i> , <i>expr</i>	
(9)	<i>expr-list</i> → <i>expr</i>	

- In the latter case we choose reduction by production (7).

## Example — *continued*

	STACK	INPUT
	... procid ( id	, id ) ...
	STACK	INPUT
	... id ( id	, id ) ...
(1)	stmt	→ id ( parameter-list )
(2)	stmt	→ expr := expr
(3)	parameter-list	→ parameter-list , parameter
(4)	parameter-list	→ parameter
(5)	parameter	→ id
(6)	expr	→ id ( expr-list )
(7)	expr	→ id
(8)	expr-list	→ expr-list , expr
(9)	expr-list	→ expr

- Notice how the symbol third from the top of the stack determines the reduction to be made, even though it is not involved in the reduction.

## Example — *continued*

	STACK	INPUT
	... <b>procid</b> ( <b>id</b>	, <b>id</b> ) ...
	... <b>id</b> ( <b>id</b>	, <b>id</b> ) ...
(1)	<i>stmt</i>	→ <b>id</b> ( <i>parameter-list</i> )
(2)	<i>stmt</i>	→ <i>expr</i> := <i>expr</i>
(3)	<i>parameter-list</i>	→ <i>parameter-list</i> , <i>parameter</i>
(4)	<i>parameter-list</i>	→ <i>parameter</i>
(5)	<i>parameter</i>	→ <b>id</b>
(6)	<i>expr</i>	→ <b>id</b> ( <i>expr-list</i> )
(7)	<i>expr</i>	→ <b>id</b>
(8)	<i>expr-list</i>	→ <i>expr-list</i> , <i>expr</i>
(9)	<i>expr-list</i>	→ <i>expr</i>

- Shift-reduce parsing can utilize information far down in the stack to guide the parse.

# Introduction to LR Parsing: Simple LR

- The most prevalent type of bottom-up parser today is based on a concept called  $LR(k)$  parsing.
- The “L” is for left-to-right scanning of the input.
- The “R” is for constructing a rightmost derivation in reverse.
- The  $k$  is for the number of input symbols of lookahead that are used in making parsing decisions.

- The cases  $k = 0$  or  $k = 1$  are of practical interest.
- We shall only consider LR parsers with  $k \leq 1$  here.
- When  $(k)$  is omitted,  $k$  is assumed to be 1.

- Introduces the basic concepts of LR parsing.
- The easiest method for constructing shift-reduce parsers, called “simple LR” (or SLR, for short).
- Some familiarity with the basic concepts is helpful even if the LR parser itself is constructed using an automatic parser generator.

- “Items” and “parser states” are important concepts.
- The diagnostic output from an LR parser generator typically includes parser states.
- Can be used to isolate the sources of parsing conflicts.

# Why LR Parsers?

- LR parsers are table-driven, much like the nonrecursive LL parsers.
- A grammar for which we can construct a parsing table is said to be an LR grammar.
- Intuitively, for a grammar to be LR it is sufficient that a left-to-right shift-reduce parser be able to recognize handles of right-sentential forms when they appear on top of the stack.

- LR parsing is attractive for a variety of reasons:
- LR parsers can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written.
- NonLR context-free grammars exist, but these can generally be avoided for typical programming-language constructs.

- LR parsing is attractive for a variety of reasons:
- The LR-parsing method is the most general nonbacktracking shift-reduce parsing method known.
- Yet it can be implemented as efficiently as other, more primitive shift-reduce methods.

## Why LR Parsers? — *continued*

- LR parsing is attractive for a variety of reasons:
- An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

## Why LR Parsers? — *continued*

- LR parsing is attractive for a variety of reasons:
- The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive or LL methods.
- For a grammar to be  $LR(k)$ , we must be able to recognize the occurrence of the right side of a production in a right-sentential form, with  $k$  input symbols of lookahead.
- ...

## Why LR Parsers? — *continued*

- LR parsing is attractive for a variety of reasons:
  - ...
- This requirement is far less stringent than that for  $LL(k)$  grammars where we must be able to recognize the use of a production seeing only the first  $k$  symbols of what its right side derives.
- Thus, it should not be surprising that LR grammars can describe more languages than LL grammars.

## Why LR Parsers? — *continued*

- The principal drawback of the LR method is that it is too much work to construct an LR parser by hand for a typical programming-language grammar.
- A specialized tool, an LR parser generator, is needed.

## Why LR Parsers? — *continued*

- Fortunately, many such generators are available.
- One of the most commonly used ones, Yacc.
- Such a generator takes a context-free grammar and automatically produces a parser for that grammar.
- If the grammar contains ambiguities or other constructs that are difficult to parse in a left-to-right scan of the input, then the parser generator locates these constructs and provides detailed diagnostic messages.

# Items and the LR(0) Automaton

- How does a shift-reduce parser know when to shift and when to reduce with stack contents  $\$T$  and next input symbol  $*$ ?
- How does the parser know that  $T$  on the top of the stack is not a handle, so the appropriate action is to shift and not to reduce  $T$  to  $E$ ?

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$ \mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \rightarrow \mathbf{id}$
$\$ F$	$* \mathbf{id}_2 \$$	reduce by $T \rightarrow F$
$\$ T$	$* \mathbf{id}_2 \$$	shift
$\$ T *$	$\mathbf{id}_2 \$$	shift
$\$ T * \mathbf{id}_2$	$\$$	reduce by $F \rightarrow \mathbf{id}$
$\$ T * F$	$\$$	reduce by $T \rightarrow T * F$
$\$ T$	$\$$	reduce by $E \rightarrow T$
$\$ E$	$\$$	accept

Configurations of a shift-reduce parser on input  $\mathbf{id}_1 * \mathbf{id}_2$

## Items and the LR(0) Automaton — *continued*

STACK	INPUT	ACTION
\$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
\$ $\mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \rightarrow \mathbf{id}$
\$ $F$	$* \mathbf{id}_2 \$$	reduce by $T \rightarrow F$
\$ $T$	$* \mathbf{id}_2 \$$	shift
\$ $T *$	$\mathbf{id}_2 \$$	shift
\$ $T * \mathbf{id}_2$	\$	reduce by $F \rightarrow \mathbf{id}$
\$ $T * F$	\$	reduce by $T \rightarrow T * F$
\$ $T$	\$	reduce by $E \rightarrow T$
\$ $E$	\$	accept

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

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

- Why not reduce by using the  $E$  production?

## Items and the LR(0) Automaton — *continued*

- An LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in a parse.
- States represent sets of “items.”
- An  $LR(0)$  item (“item” for short) of a grammar  $G$  is a production of  $G$  with a dot at some position of the body.

## Items and the LR(0) Automaton — *continued*

- An  $LR(0)$  item (“item” for short) of a grammar  $G$  is a production of  $G$  with a dot at some position of the body.
- Thus, production  $A \rightarrow XYZ$  yields the four items,

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

- The production  $A \rightarrow \epsilon$  generates only one item,  $A \rightarrow \epsilon \cdot$ .

## Items and the LR(0) Automaton — *continued*

$A \rightarrow \cdot XYZ$

$A \rightarrow X \cdot YZ$

$A \rightarrow XY \cdot Z$

$A \rightarrow XYZ \cdot$

- Intuitively, an item indicates how much of a production we have seen at a given point in the parsing process.
- For example, the item  $A \rightarrow \cdot XYZ$  indicates that we hope to see a string derivable from  $XYZ$  next on the input.
- Item  $A \rightarrow X \cdot YZ$  indicates that we have just seen on the input a string derivable from  $X$  and that we hope next to see a string derivable from  $YZ$ .
- Item  $A \rightarrow XYZ \cdot$  indicates that we have seen the body  $XYZ$  and that it may be time to reduce  $XYZ$  to  $A$ .

## Items and the LR(0) Automaton — *continued*

$A \rightarrow \cdot XYZ$

$A \rightarrow X \cdot YZ$

$A \rightarrow XY \cdot Z$

$A \rightarrow XYZ \cdot$

- Intuitively, an item indicates how much of a production we have seen at a given point in the parsing process.
- For example, the item  $A \rightarrow \cdot XYZ$  indicates that we hope to see a string derivable from  $XYZ$  next on the input.
- Item  $A \rightarrow X \cdot YZ$  indicates that we have just seen on the input a string derivable from  $X$  and that we hope next to see a string derivable from  $YZ$ .
- Item  $A \rightarrow XYZ \cdot$  indicates that we have seen the body  $XYZ$  and that it may be time to reduce  $XYZ$  to  $A$ .

## Items and the LR(0) Automaton — *continued*

$A \rightarrow \cdot XYZ$

$A \rightarrow X \cdot YZ$

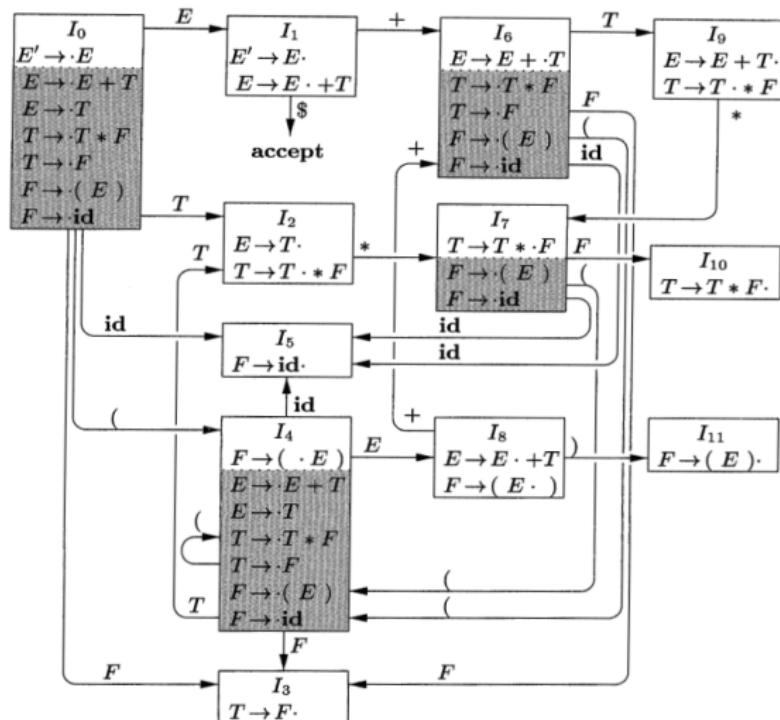
$A \rightarrow XY \cdot Z$

$A \rightarrow XYZ \cdot$

- Intuitively, an item indicates how much of a production we have seen at a given point in the parsing process.
- For example, the item  $A \rightarrow \cdot XYZ$  indicates that we hope to see a string derivable from  $XYZ$  next on the input.
- Item  $A \rightarrow X \cdot YZ$  indicates that we have just seen on the input a string derivable from  $X$  and that we hope next to see a string derivable from  $YZ$ .
- Item  $A \rightarrow XYZ \cdot$  indicates that we have seen the body  $XYZ$  and that it may be time to reduce  $XYZ$  to  $A$ .

- One collection of sets of LR(0) items, called the canonical LR(0) collection, provides the basis for constructing a deterministic finite automaton that is used to make parsing decisions.
- Such an automaton is called an LR(0) automaton.
- In particular, each state of the LR(0) automaton represents a set of items in the canonical LR(0) collection.

# Items and the LR(0) Automaton — *continued*



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

LR(0) automaton for the expression grammar

## Items and the LR(0) Automaton — *continued*

- To construct the canonical LR(0) collection for a grammar, we define an augmented grammar and two functions, CLOSURE and GOTO.

## Items and the LR(0) Automaton — *continued*

- If  $G$  is a grammar with start symbol  $S$ , then  $G'$ , the augmented grammar for  $G$  is,
  - $G$  with a new start symbol  $S'$  and
  - production  $S' \rightarrow S$ .
- The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input.
- That is, acceptance occurs when and only when the parser is about to reduce by  $S' \rightarrow S$ .

## Items and the LR(0) Automaton — *continued*

- If  $G$  is a grammar with start symbol  $S$ , then  $G'$ , the augmented grammar for  $G$  is,
  - $G$  with a new start symbol  $S'$  and
  - production  $S' \rightarrow S$ .
- The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input.
- That is, acceptance occurs when and only when the parser is about to reduce by  $S' \rightarrow S$ .

# Items and the LR(0) Automaton — Closure of Item Sets

- If  $I$  is a set of items for a grammar  $G$ , then  $\text{CLOSURE}(I)$  is the set of items constructed from  $I$  by the two rules:
  - 1 Initially, add every item in  $I$  to  $\text{CLOSURE}(I)$ .
  - 2 If  $A \rightarrow \alpha \cdot B\beta$  is in  $\text{CLOSURE}(I)$  and  $B \rightarrow \gamma$  is a production, then add the item  $B \rightarrow \cdot\gamma$  to  $\text{CLOSURE}(I)$ , if it is not already there.
- Apply this rule until no more new items can be added to  $\text{CLOSURE}(I)$ .

# Items and the LR(0) Automaton — Closure of Item Sets

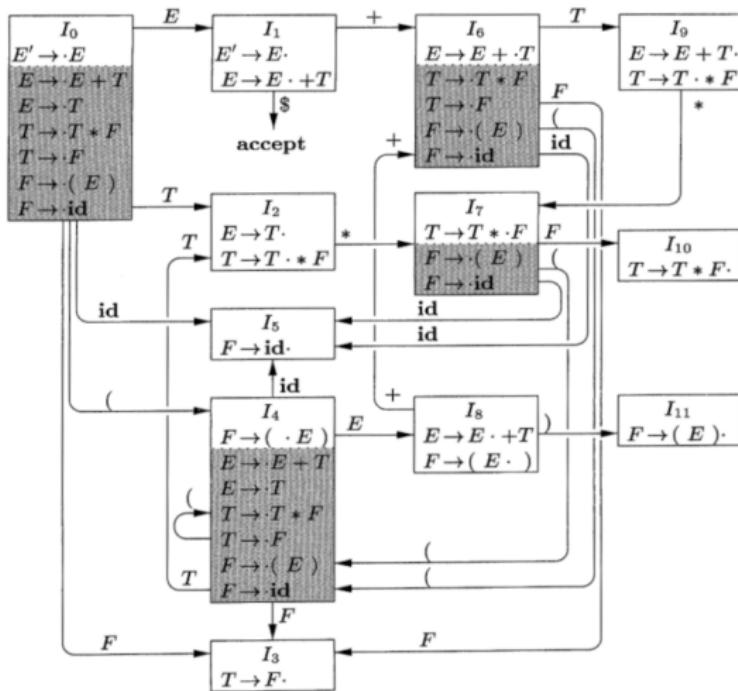
- If  $I$  is a set of items for a grammar  $G$ , then  $\text{CLOSURE}(I)$  is the set of items constructed from  $I$  by the two rules:
  - 1 Initially, add every item in  $I$  to  $\text{CLOSURE}(I)$ .
  - 2 If  $A \rightarrow \alpha \cdot B\beta$  is in  $\text{CLOSURE}(I)$  and  $B \rightarrow \gamma$  is a production, then add the item  $B \rightarrow \cdot\gamma$  to  $\text{CLOSURE}(I)$ , if it is not already there.
- Apply this rule until no more new items can be added to  $\text{CLOSURE}(I)$ .

# Example

- Consider the augmented expression grammar:

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

## Example — *continued*



- If  $I$  is the set of one item  $\{[E' \rightarrow \cdot E]\}$ , then  $\text{CLOSURE}(I)$  contains the set of items  $I_0$ .

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

$I_0$
$E' \rightarrow \cdot E$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot \mathbf{id}$

```

SetOfItems CLOSURE( $I$ ) {
     $J = I$ ;
    repeat
        for ( each item  $A \rightarrow \alpha \cdot B \beta$  in  $J$  )
            for ( each production  $B \rightarrow \gamma$  of  $G$  )
                if (  $B \rightarrow \gamma$  is not in  $J$  )
                    add  $B \rightarrow \gamma$  to  $J$ ;
    until no more items are added to  $J$  on one round;
    return  $J$ ;
}

```

- If  $I$  is the set of one item  $\{[E' \rightarrow \cdot E]\}$ , then  $\text{CLOSURE}(I)$  contains the set of items  $I_0$ .

$$\begin{array}{lcl}
 E' & \rightarrow & E \\
 E & \rightarrow & E + T \mid T \\
 T & \rightarrow & T * F \mid F \\
 F & \rightarrow & ( E ) \mid \mathbf{id}
 \end{array}$$

$I_0$
$E' \rightarrow \cdot E$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot ( E )$
$F \rightarrow \cdot \mathbf{id}$

```

SetOfItems CLOSURE( $I$ ) {
     $J = I;$ 
    repeat
        for ( each item  $A \rightarrow \alpha \cdot B \beta$  in  $J$  )
            for ( each production  $B \rightarrow \gamma$  of  $G$  )
                if (  $B \rightarrow \cdot \gamma$  is not in  $J$  )
                    add  $B \rightarrow \cdot \gamma$  to  $J$ ;
    until no more items are added to  $J$  on one round;
    return  $J$ ;
}

```

- To see how the closure is computed,  
 $E' \rightarrow \cdot E$  is put in  $\text{CLOSURE}(I)$  by rule (1).

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

$I_0$
$E' \rightarrow \cdot E$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot \mathbf{id}$

```

SetOfItems CLOSURE( $I$ ) {
     $J = I;$ 
    repeat
        for ( each item  $A \rightarrow \alpha \cdot B \beta$  in  $J$  )
            for ( each production  $B \rightarrow \gamma$  of  $G$  )
                if (  $B \rightarrow \cdot \gamma$  is not in  $J$  )
                    add  $B \rightarrow \cdot \gamma$  to  $J$ ;
    until no more items are added to  $J$  on one round;
    return  $J$ ;
}

```

- Since there is an  $E$  immediately to the right of a dot, we add the  $E$ -productions with dots at the left ends:  $E \rightarrow \cdot E + T$  and  $E \rightarrow \cdot T$ .

$$\begin{array}{lcl}
 E' & \rightarrow & E \\
 E & \rightarrow & E + T \mid T \\
 T & \rightarrow & T * F \mid F \\
 F & \rightarrow & ( E ) \mid \mathbf{id}
 \end{array}$$

$I_0$
$E' \rightarrow \cdot E$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot ( E )$
$F \rightarrow \cdot \mathbf{id}$

```

SetOfItems CLOSURE( $I$ ) {
     $J = I;$ 
    repeat
        for ( each item  $A \rightarrow \alpha \cdot B \beta$  in  $J$  )
            for ( each production  $B \rightarrow \gamma$  of  $G$  )
                if (  $B \rightarrow \cdot \gamma$  is not in  $J$  )
                    add  $B \rightarrow \cdot \gamma$  to  $J$ ;
    until no more items are added to  $J$  on one round;
    return  $J$ ;
}

```

- Now there is a  $T$  immediately to the right of a dot in the latter item, so we add  $T \rightarrow \cdot T * F$  and  $T \rightarrow \cdot F$ .

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

$I_0$
$E' \rightarrow \cdot E$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot \mathbf{id}$

```

SetOfItems CLOSURE( $I$ ) {
     $J = I;$ 
    repeat
        for ( each item  $A \rightarrow \alpha \cdot B \beta$  in  $J$  )
            for ( each production  $B \rightarrow \gamma$  of  $G$  )
                if (  $B \rightarrow \gamma$  is not in  $J$  )
                    add  $B \rightarrow \gamma$  to  $J$ ;
    until no more items are added to  $J$  on one round;
    return  $J$ ;
}

```

- Next, the  $F$  to the right of a dot forces us to add  $F \rightarrow \cdot(E)$  and  $F \rightarrow \mathbf{id}$ , but no other items need to be added.

$I_0$
$E' \rightarrow \cdot E$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot ( E )$
$F \rightarrow \cdot \text{id}$

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

```

SetOfItems CLOSURE( $I$ ) {
     $J = I$ ;
    repeat
        for ( each item  $A \rightarrow \alpha \cdot B \beta$  in  $J$  )
            for ( each production  $B \rightarrow \gamma$  of  $G$  )
                if (  $B \rightarrow \cdot \gamma$  is not in  $J$  )
                    add  $B \rightarrow \cdot \gamma$  to  $J$ ;
    until no more items are added to  $J$  on one round;
    return  $J$ ;
}

```

- A convenient way to implement the function *closure* is to keep a Boolean array *added*, indexed by the nonterminals of  $G$ , such that  $\text{added}[B]$  is set to **true** if and when we add the item  $B \rightarrow \cdot \gamma$  for each  $B$ -production  $B \rightarrow \gamma$ .

$I_0$
$E' \rightarrow \cdot E$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot ( E )$
$F \rightarrow \cdot \text{id}$

$$E' \rightarrow E$$

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

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

```

SetOfItems CLOSURE( $I$ ) {
     $J = I;$ 
    repeat
        for ( each item  $A \rightarrow \alpha \cdot B \beta$  in  $J$  )
            for ( each production  $B \rightarrow \gamma$  of  $G$  )
                if (  $B \rightarrow \cdot \gamma$  is not in  $J$  )
                    add  $B \rightarrow \cdot \gamma$  to  $J$ ;
    until no more items are added to  $J$  on one round;
    return  $J$ ;
}

```

- Note that if one  $B$ -production is added to the closure of  $I$  with the dot at the left end, then all  $B$ -productions will be similarly added to the closure.

$I_0$
$E' \rightarrow \cdot E$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot ( E )$
$F \rightarrow \cdot \text{id}$

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

```

SetOfItems CLOSURE( $I$ ) {
     $J = I$ ;
    repeat
        for ( each item  $A \rightarrow \alpha \cdot B \beta$  in  $J$  )
            for ( each production  $B \rightarrow \gamma$  of  $G$  )
                if (  $B \rightarrow \cdot \gamma$  is not in  $J$  )
                    add  $B \rightarrow \cdot \gamma$  to  $J$ ;
    until no more items are added to  $J$  on one round;
    return  $J$ ;
}

```

- Hence, it is not necessary in some circumstances actually to list the items  $B \rightarrow \gamma$  added to  $I$  by CLOSURE.

$I_0$
$E' \rightarrow \cdot E$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot ( E )$
$F \rightarrow \cdot \text{id}$

$$E' \rightarrow E$$

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

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

```

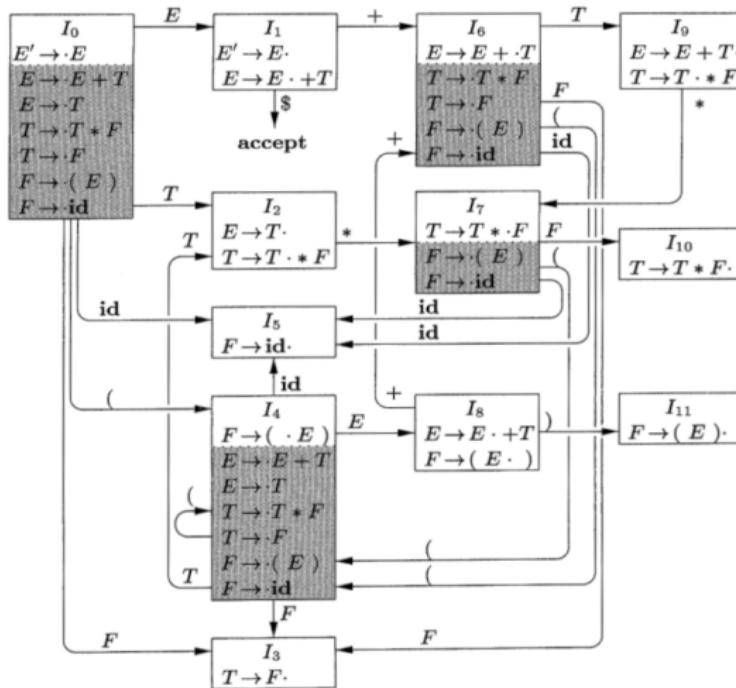
SetOfItems CLOSURE( $I$ ) {
     $J = I;$ 
    repeat
        for ( each item  $A \rightarrow \alpha \cdot B \beta$  in  $J$  )
            for ( each production  $B \rightarrow \gamma$  of  $G$  )
                if (  $B \rightarrow \cdot \gamma$  is not in  $J$  )
                    add  $B \rightarrow \cdot \gamma$  to  $J$ ;
    until no more items are added to  $J$  on one round;
    return  $J$ ;
}

```

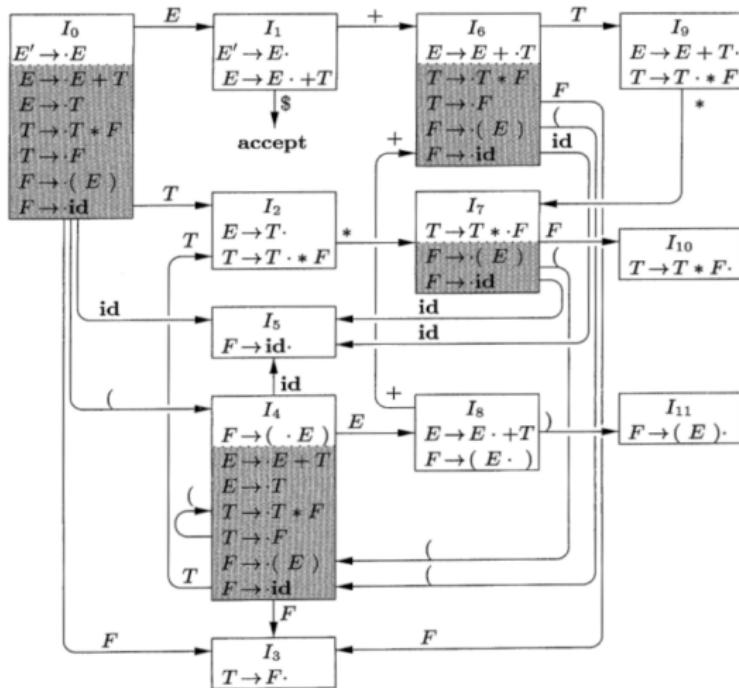
- A list of the nonterminals  $B$  whose productions were so added will suffice.

## Items and the LR(0) Automaton — Closure of Item Sets — *continued*

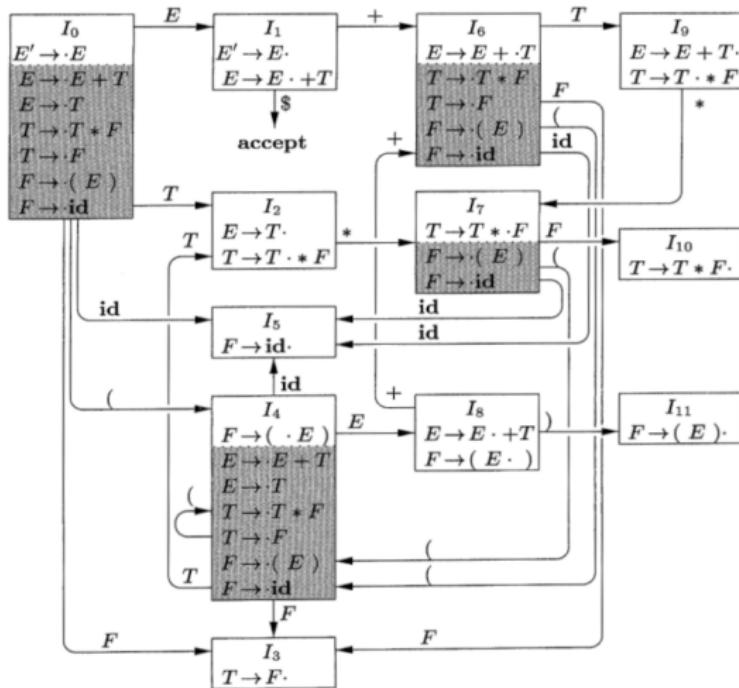
- We divide all the sets of items of interest into two classes:
  1. **Kernel items:** the initial item,  $S' \rightarrow \cdot S$ , and all items whose dots are not at the left end.
  2. **Nonkernel items:** all items with their dots at the left end, except for  $S' \rightarrow \cdot S$ .



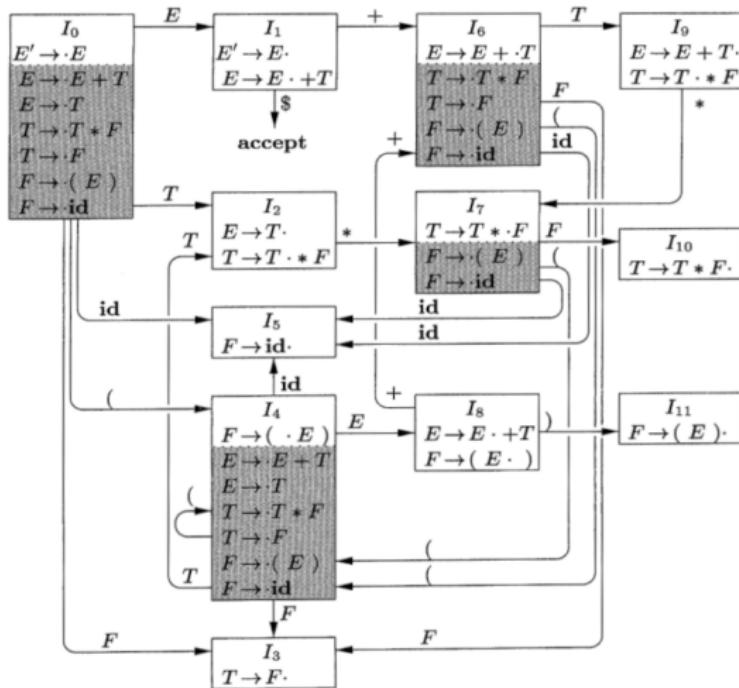
- Moreover, each set of items of interest is formed by taking the closure of a set of kernel items.



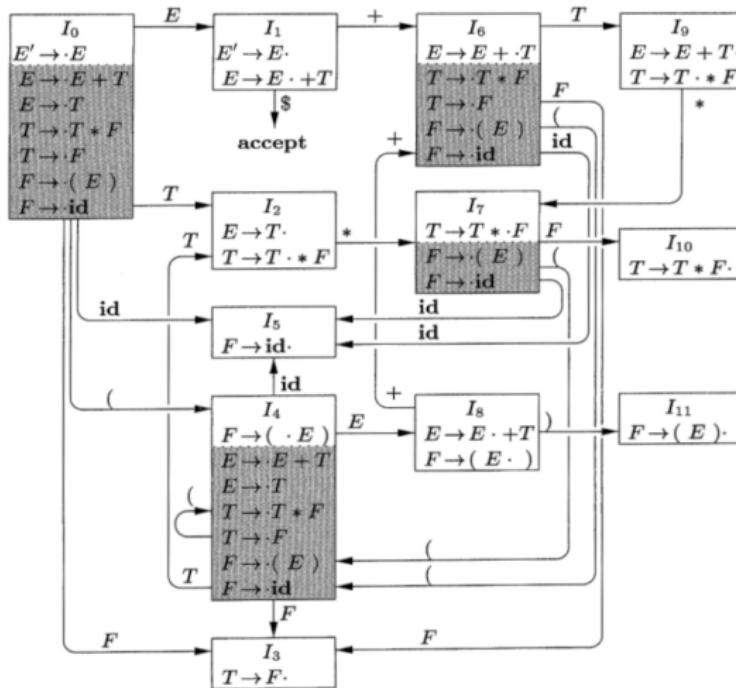
- The items added in the closure can never be kernel items, of course.



- Thus, we can represent the sets of items we are really interested in with very little storage if we throw away all nonkernel items.



- Knowing that they could be regenerated by the closure process.



- Nonkernel items are in the shaded part of the box for a state.

# Items and the LR(0) Automaton — the Function GOTO

- The second useful function is  $\text{GOTO}(I, X)$  where  $I$  is a set of items and  $X$  is a grammar symbol.
- $\text{GOTO}(I, X)$  is defined to be the closure of the set of all items  $[A \rightarrow \alpha X \cdot \beta]$  such that  $[A \rightarrow \alpha \cdot X \beta]$  is in  $I$ .
- Intuitively, the GOTO function is used to define the transitions in the LR(0) automaton for a grammar.
- The states of the automaton correspond to sets of items.
- $\text{GOTO}(I, X)$  specifies the transition from the state for  $I$  under input  $X$ .

# Example

- If  $I$  is the set of two items  $\{[E' \rightarrow E \cdot], [E \rightarrow E \cdot + T]\}$ , then  $\text{GOTO}(I, +)$  contains the items,

$$E \rightarrow E + \cdot T$$

$$T \rightarrow \cdot T * F$$

$$T \rightarrow \cdot F$$

$$F \rightarrow \cdot (E)$$

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

- We computed  $\text{GOTO}(I, +)$  by examining  $I$  for items with  $+$  immediately to the right of the dot.
- $E' \rightarrow E \cdot$  is not such an item, but  $E \rightarrow E \cdot + T$  is.
- We moved the dot over the  $+$  to get  $E \rightarrow E + \cdot T$  and then took the closure of this singleton set.

# Example

- If  $I$  is the set of two items  $\{[E' \rightarrow E \cdot], [E \rightarrow E \cdot + T]\}$ , then  $\text{GOTO}(I, +)$  contains the items,

$E \rightarrow E + \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot \text{id}$

- We computed  $\text{GOTO}(I, +)$  by examining  $I$  for items with  $+$  immediately to the right of the dot.
- $E' \rightarrow E \cdot$  is not such an item, but  $E \rightarrow E \cdot + T$  is.
- We moved the dot over the  $+$  to get  $E \rightarrow E + \cdot T$  and then took the closure of this singleton set.

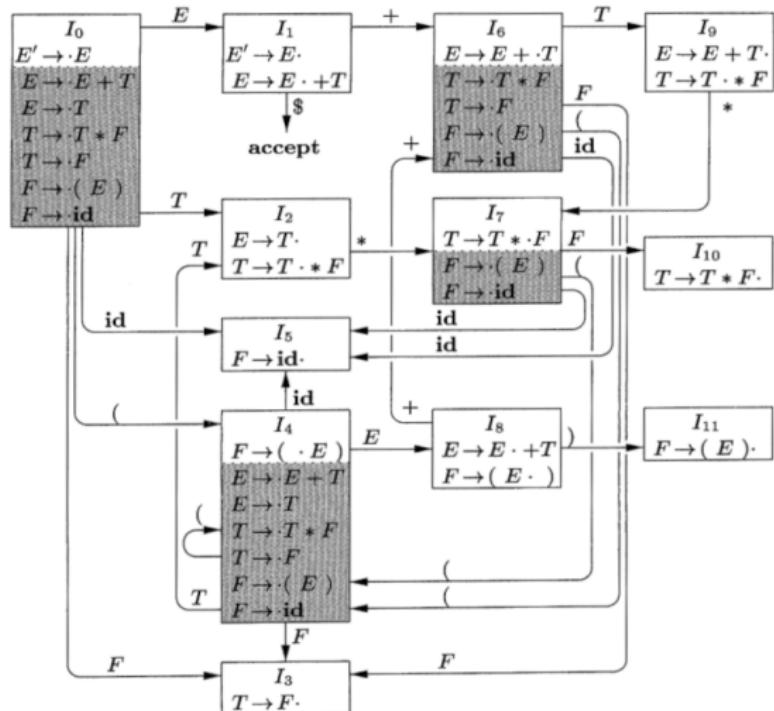
# Items and the LR(0) Automaton — the Function GOTO — *continued*

```
void items(G') {
    C = CLOSURE({[S' → ·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 on a round;
}
```

Computation of the canonical collection of sets of LR(0) items

- The algorithm to construct  $C$ , the canonical collection of sets of LR(0) items for an augmented grammar  $G'$ .

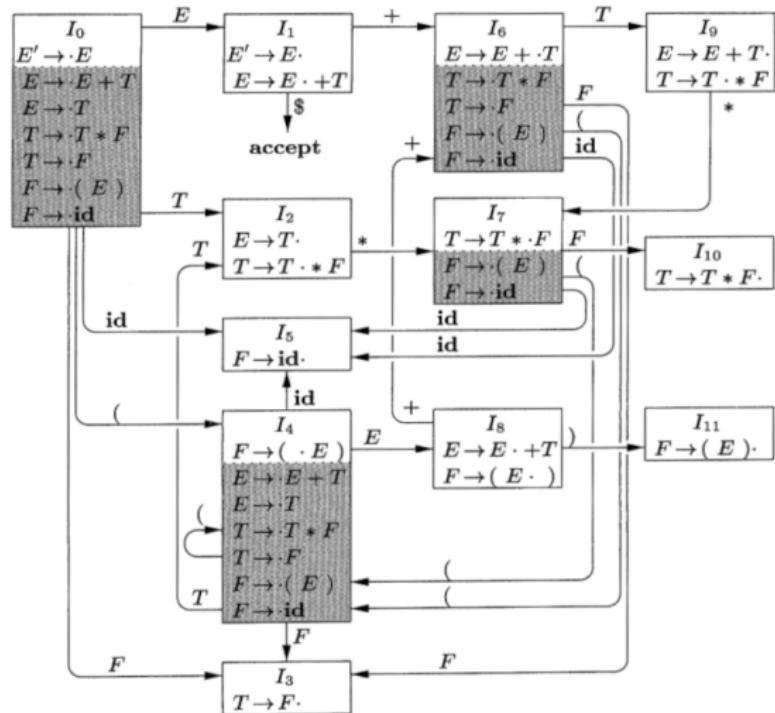
# Example



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

- The canonical collection of sets of LR(0) items for the grammar and the GOTO function are shown.

# Example

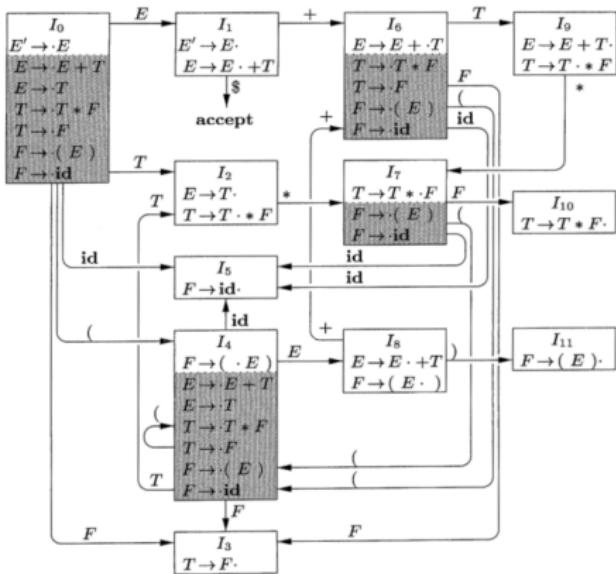


$E' \rightarrow E$   
 $E \rightarrow E + T \mid T$   
 $T \rightarrow T * F \mid F$   
 $F \rightarrow ( E ) \mid id$

- GOTO is encoded by the transitions in the figure.

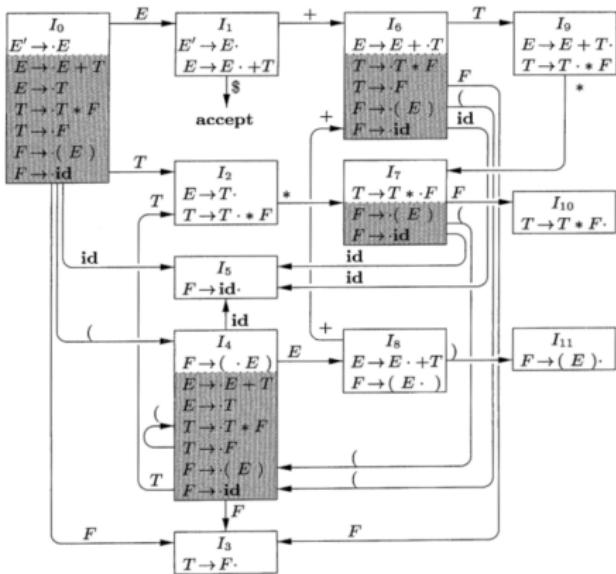
# Items and the LR(0) Automaton — Use of the LR(0) Automaton

- The central idea behind “Simple LR,” or SLR, parsing is the construction from the grammar of the LR(0) automaton.
- The states of this automaton are the sets of items from the canonical LR(0) collection, and the transitions are given by the GOTO function.



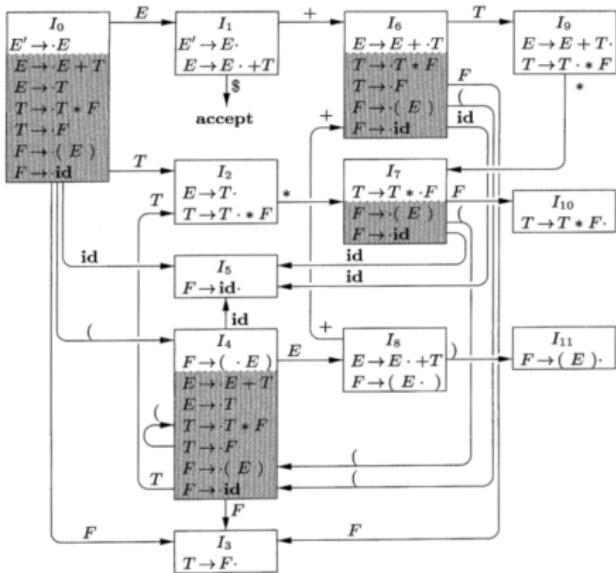
$$\begin{aligned}
 E' &\rightarrow E \\
 E &\rightarrow E + T \mid T \\
 T &\rightarrow T * F \mid F \\
 F &\rightarrow ( E ) \mid \mathbf{id}
 \end{aligned}$$

- The LR(0) automaton for the expression grammar appeared earlier.
- The start state of the LR(0) automaton is  $\text{CLOSURE}(\{[S' \rightarrow \cdot S]\})$ , where  $S'$  is the start symbol of the augmented grammar.



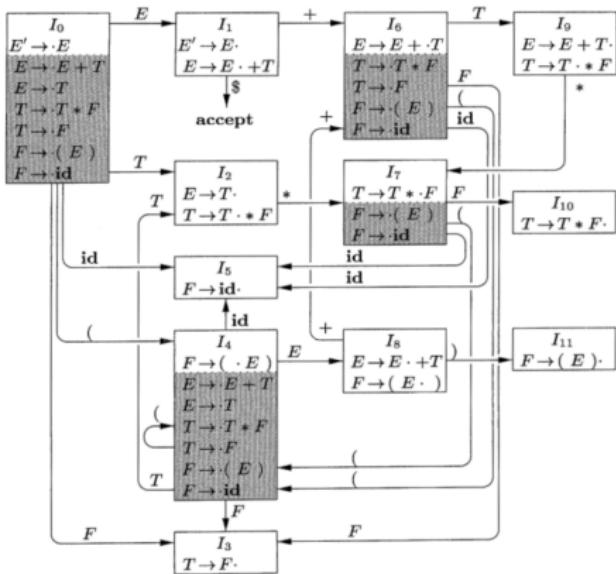
$$\begin{aligned}
 E' &\rightarrow E \\
 E &\rightarrow E + T \mid T \\
 T &\rightarrow T * F \mid F \\
 F &\rightarrow ( E ) \mid \mathbf{id}
 \end{aligned}$$

- All states are accepting states.
- We say “state  $j$ ” to refer to the state corresponding to the set of items  $I_j$ .



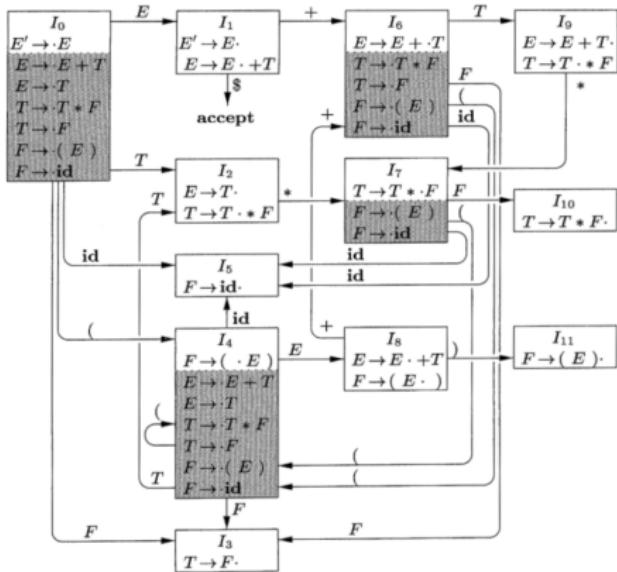
$E'$	$\rightarrow$	$E$
$E$	$\rightarrow$	$E + T \mid T$
$T$	$\rightarrow$	$T * F \mid F$
$F$	$\rightarrow$	$( E ) \mid \mathbf{id}$

- How can LR(0) automata help with shift-reduce decisions?



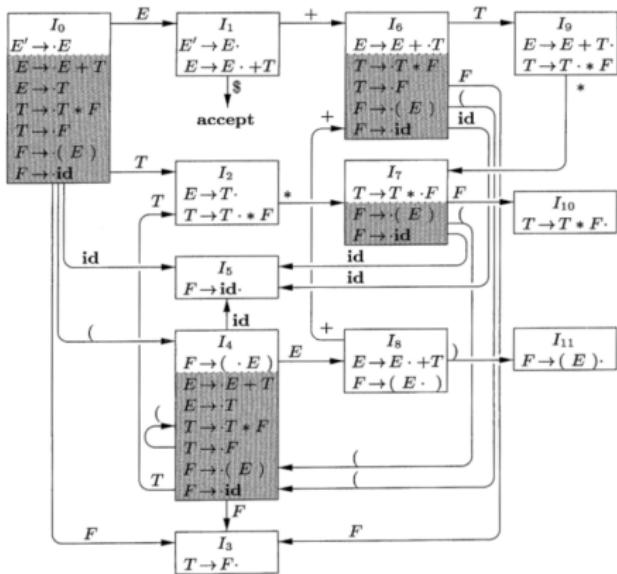
$$\begin{aligned}
 E' &\rightarrow E \\
 E &\rightarrow E + T \mid T \\
 T &\rightarrow T * F \mid F \\
 F &\rightarrow ( E ) \mid \mathbf{id}
 \end{aligned}$$

- Shift-reduce decisions can be made as follows.
- Suppose that the string  $\gamma$  of grammar symbols takes the LR(0) automaton from the start state 0 to some state  $j$ .



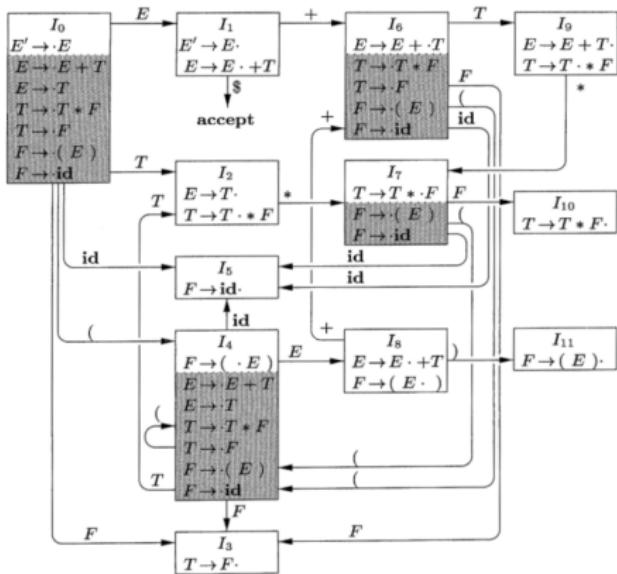
$E'$	$\rightarrow$	$E$
$E$	$\rightarrow$	$E + T \mid T$
$T$	$\rightarrow$	$T * F \mid F$
$F$	$\rightarrow$	$( E ) \mid \mathbf{id}$

- Then, shift on next input symbol  $a$  if state  $j$  has a transition on  $a$ .



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

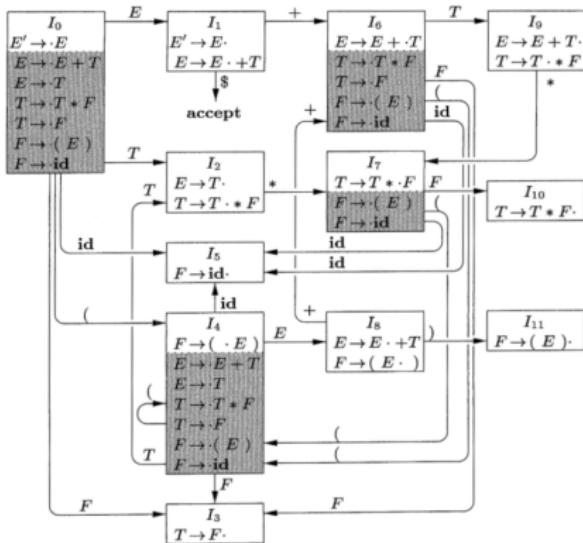
- Otherwise, we choose to reduce.
- The items in state  $j$  will tell us which production to use.



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

- The LR-parsing algorithm uses its stack to keep track of states as well as grammar symbols.
- In fact, the grammar symbol can be recovered from the state, so the stack holds states.

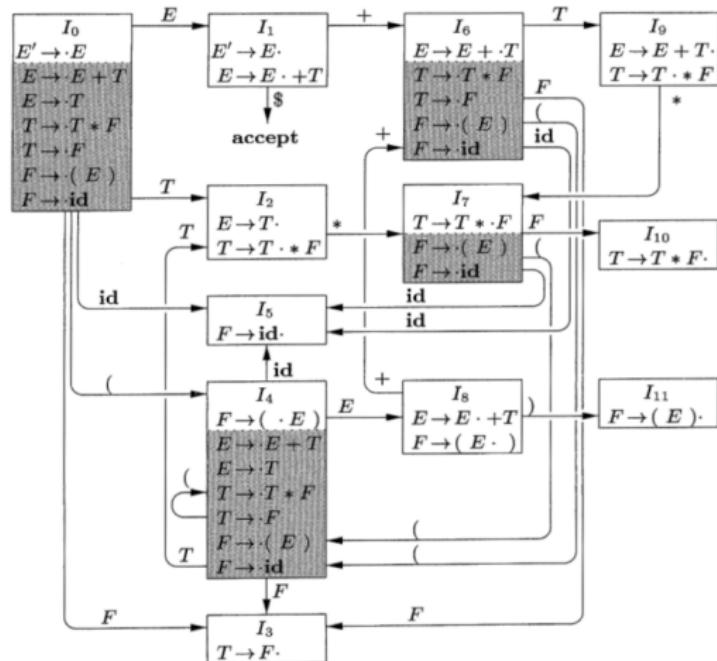
# Example



LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	<b>id</b> *	shift to 5
(2)	0 5	\$ <b>id</b>	*	reduce by $F \rightarrow id$
(3)	0 3	\$ <b>F</b>	*	reduce by $T \rightarrow F$
(4)	0 2	\$ <b>T</b>	*	shift to 7
(5)	0 2 7	\$ <b>T</b> *	<b>id</b> \$	shift to 5
(6)	0 2 7 5	\$ <b>T</b> * <b>id</b>	\$	reduce by $F \rightarrow id$
(7)	0 2 7 10	\$ <b>T</b> * <b>F</b>	\$	reduce by $T \rightarrow T * F$
(8)	0 2	\$ <b>T</b>	\$	reduce by $E \rightarrow T$
(9)	0 1	\$ <b>E</b>	\$	accept

- Illustrates the actions of a shift-reduce parser on input **id \* id**, using the LR(0) automaton.

**id \* id**



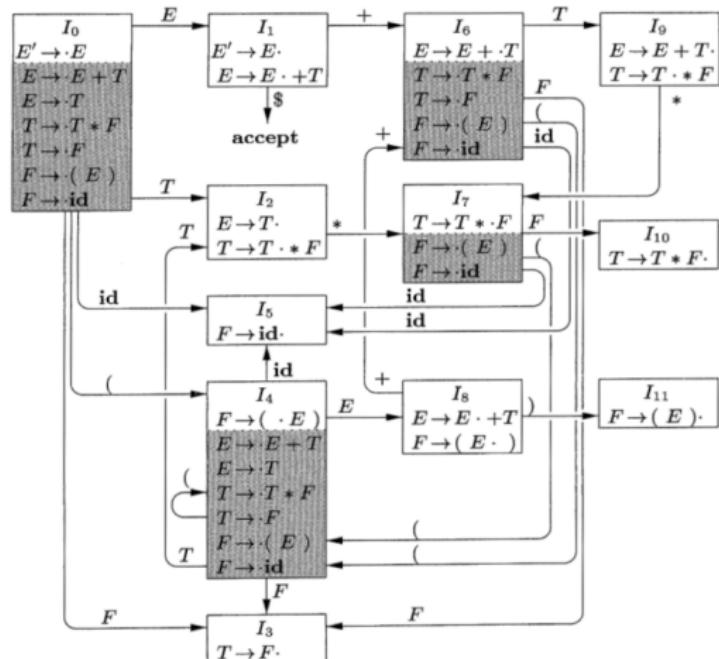
- We use a stack to hold states.
  - For clarity, the grammar symbols corresponding to the states on the stack appear in column SYMBOLS.

## **id \* id**

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	<b>id * id \$</b>	shift to 5
(2)	0 5	\$ <b>id</b>	* <b>id \$</b>	reduce by $F \rightarrow id$
(3)	0 3	\$ <b>F</b>	* <b>id \$</b>	reduce by $T \rightarrow F$
(4)	0 2	\$ <b>T</b>	* <b>id \$</b>	shift to 7
(5)	0 2 7	\$ <b>T * </b>	<b>id \$</b>	shift to 5
(6)	0 2 7 5	\$ <b>T * id</b>	\$	reduce by $F \rightarrow id$
(7)	0 2 7 10	\$ <b>T * F</b>	\$	reduce by $T \rightarrow T * F$
(8)	0 2	\$ <b>T</b>	\$	reduce by $E \rightarrow T$
(9)	0 1	\$ <b>E</b>	\$	accept

- We use a stack to hold states.
- For clarity, the grammar symbols corresponding to the states on the stack appear in column SYMBOLS.

**id \* id**



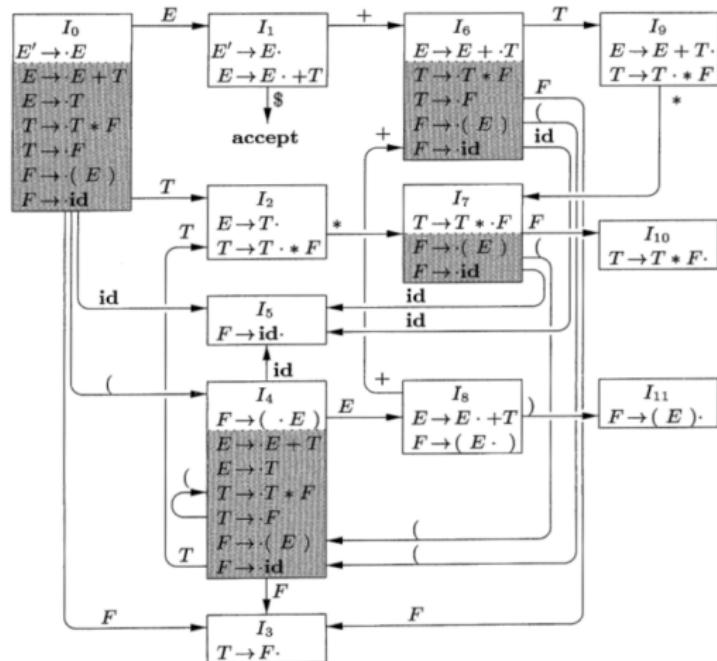
- At line (1), the stack holds the start state 0 of the automaton.
  - The corresponding symbol is the bottom-of-stack marker \$.

## **id \* id**

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	<b>id * id \$</b>	shift to 5
(2)	0 5	\$ <b>id</b>	* <b>id</b> \$	reduce by $F \rightarrow id$
(3)	0 3	\$ <b>F</b>	* <b>id</b> \$	reduce by $T \rightarrow F$
(4)	0 2	\$ <b>T</b>	* <b>id</b> \$	shift to 7
(5)	0 2 7	\$ <b>T</b> *	<b>id</b> \$	shift to 5
(6)	0 2 7 5	\$ <b>T</b> * <b>id</b>	\$	reduce by $F \rightarrow id$
(7)	0 2 7 10	\$ <b>T</b> * <b>F</b>	\$	reduce by $T \rightarrow T * F$
(8)	0 2	\$ <b>T</b>	\$	reduce by $E \rightarrow T$
(9)	0 1	\$ <b>E</b>	\$	accept

- At line (1), the stack holds the start state 0 of the automaton.
- The corresponding symbol is the bottom-of-stack marker \$.

**id \* id**



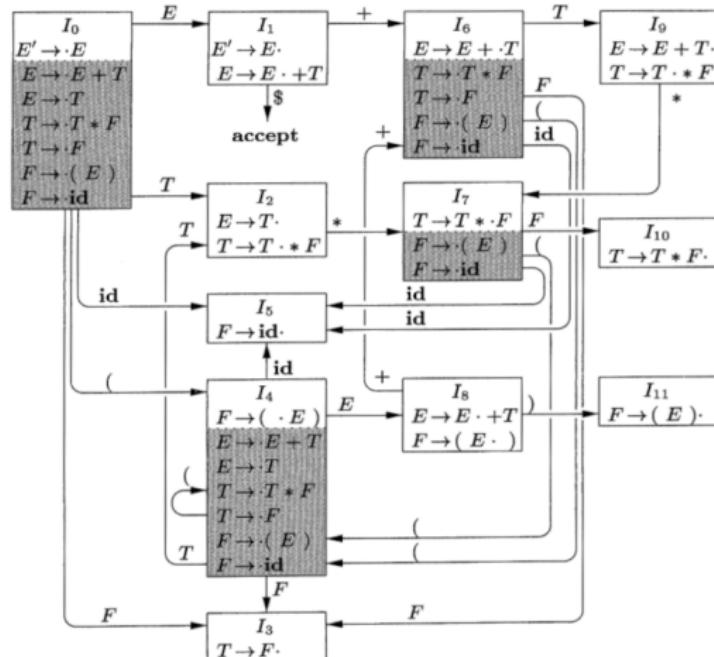
- The next input symbol is **id** and state 0 has a transition on **id** to state 5.
  - We therefore shift.

## **id \* id**

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	<b>id * id \$</b>	shift to 5
(2)	0 5	\$ <b>id</b>	* <b>id</b> \$	reduce by $F \rightarrow id$
(3)	0 3	\$ <b>F</b>	* <b>id</b> \$	reduce by $T \rightarrow F$
(4)	0 2	\$ <b>T</b>	* <b>id</b> \$	shift to 7
(5)	0 2 7	\$ <b>T * </b>	<b>id</b> \$	shift to 5
(6)	0 2 7 5	\$ <b>T * id</b>	\$	reduce by $F \rightarrow id$
(7)	0 2 7 10	\$ <b>T * F</b>	\$	reduce by $T \rightarrow T * F$
(8)	0 2	\$ <b>T</b>	\$	reduce by $E \rightarrow T$
(9)	0 1	\$ <b>E</b>	\$	accept

- The next input symbol is **id** and state 0 has a transition on **id** to state 5.
- We therefore shift.

## id \* id



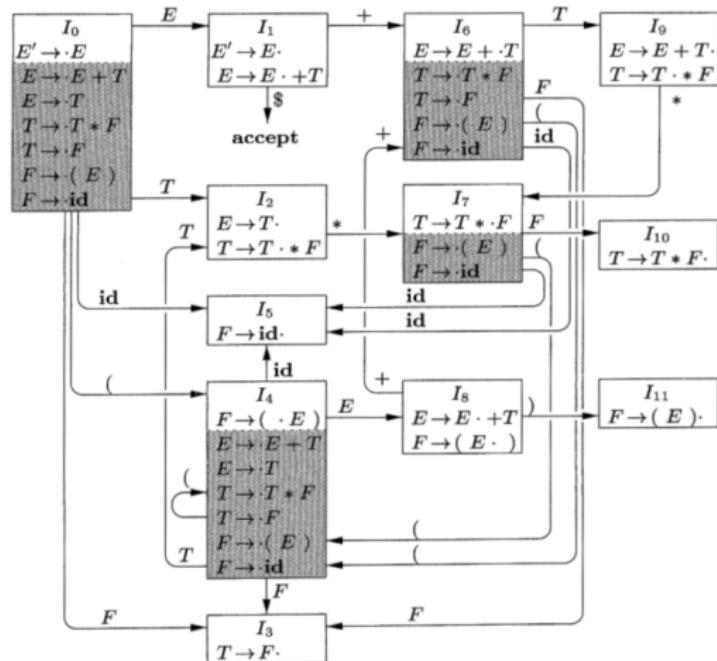
- At line (2), state 5 (symbol **id**) has been pushed onto the stack.

## **id \* id**

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	<b>id * id \$</b>	shift to 5
(2)	0 5	\$ <b>id</b>	* id \$	reduce by $F \rightarrow id$
(3)	0 3	\$ <b>F</b>	* id \$	reduce by $T \rightarrow F$
(4)	0 2	\$ <b>T</b>	* id \$	shift to 7
(5)	0 2 7	\$ <b>T * </b>	<b>id \$</b>	shift to 5
(6)	0 2 7 5	\$ <b>T * id</b>	\$	reduce by $F \rightarrow id$
(7)	0 2 7 10	\$ <b>T * F</b>	\$	reduce by $T \rightarrow T * F$
(8)	0 2	\$ <b>T</b>	\$	reduce by $E \rightarrow T$
(9)	0 1	\$ <b>E</b>	\$	accept

- At line (2), state 5 (symbol **id**) has been pushed onto the stack.

**id \* id**



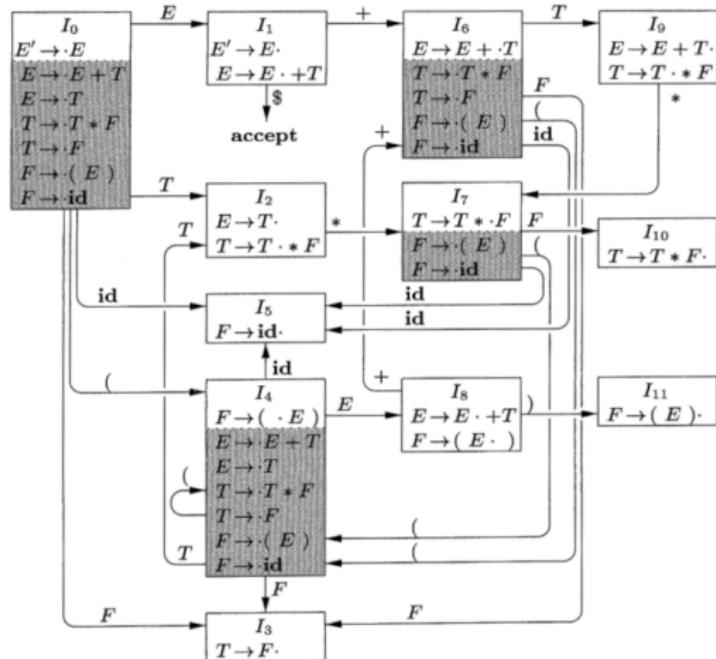
- There is no transition from state 5 on input  $*$ , so we reduce.
  - From item  $[F \rightarrow \mathbf{id} \cdot]$  in state 5, the reduction is by production  $F \rightarrow \mathbf{id}$ .

## **id \* id**

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	<b>id * id \$</b>	shift to 5
(2)	0 5	\$ <b>id</b>	* <b>id</b> \$	reduce by $F \rightarrow id$
(3)	0 3	\$ <b>F</b>	* <b>id</b> \$	reduce by $T \rightarrow F$
(4)	0 2	\$ <b>T</b>	* <b>id</b> \$	shift to 7
(5)	0 2 7	\$ <b>T * </b>	<b>id</b> \$	shift to 5
(6)	0 2 7 5	\$ <b>T * id</b>	\$	reduce by $F \rightarrow id$
(7)	0 2 7 10	\$ <b>T * F</b>	\$	reduce by $T \rightarrow T * F$
(8)	0 2	\$ <b>T</b>	\$	reduce by $E \rightarrow T$
(9)	0 1	\$ <b>E</b>	\$	accept

- There is no transition from state 5 on input \*, so we reduce.
- From item  $[F \rightarrow id \cdot]$  in state 5, the reduction is by production  $F \rightarrow id$ .

## id \* id



With symbols, a reduction is implemented by,

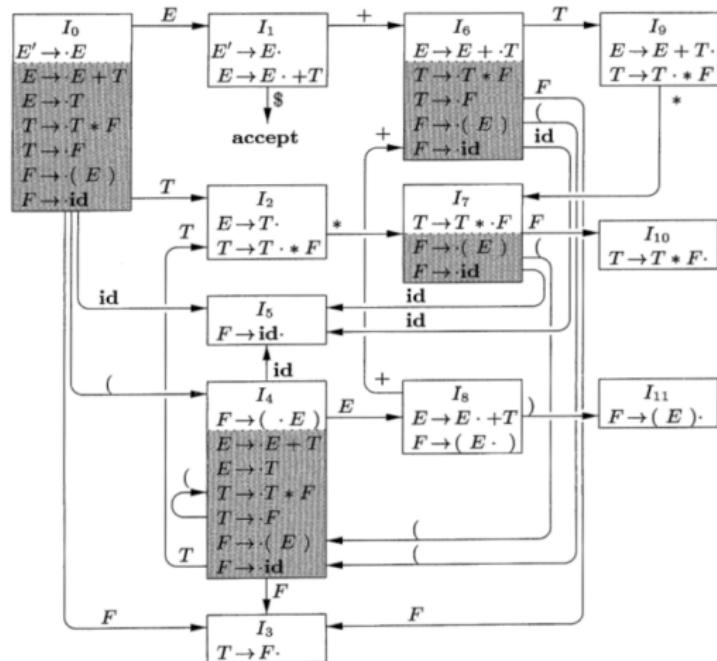
- popping the body of the production from the stack (on line (2), the body is **id**) and
- pushing the head of the production (in this case,  $F$ ).

## **id \* id**

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	<b>id * id \$</b>	shift to 5
(2)	0 5	\$ <b>id</b>	* <b>id</b> \$	reduce by $F \rightarrow id$
(3)	0 3	\$ <b>F</b>	* <b>id</b> \$	reduce by $T \rightarrow F$
(4)	0 2	\$ <b>T</b>	* <b>id</b> \$	shift to 7
(5)	0 2 7	\$ <b>T * </b>	<b>id</b> \$	shift to 5
(6)	0 2 7 5	\$ <b>T * id</b>	\$	reduce by $F \rightarrow id$
(7)	0 2 7 10	\$ <b>T * F</b>	\$	reduce by $T \rightarrow T * F$
(8)	0 2	\$ <b>T</b>	\$	reduce by $E \rightarrow T$
(9)	0 1	\$ <b>E</b>	\$	accept

- With symbols, a reduction is implemented by,
  - popping the body of the production from the stack (on line (2), the body is **id**) and
  - pushing the head of the production (in this case, **F**).

**id \* id**



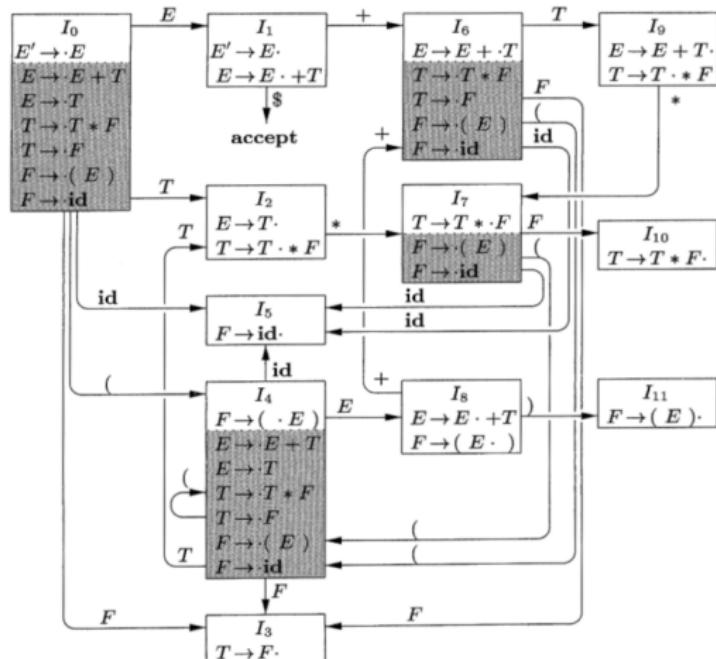
- With states, we pop state 5 for symbol **id**.
  - Which brings state 0 to the top and look for a transition on  $F$ , the head of the production.

## **id \* id**

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	<b>id * id \$</b>	shift to 5
(2)	0 5	\$ <b>id</b>	* id \$	reduce by $F \rightarrow id$
(3)	0 3	\$ <b>F</b>	* id \$	reduce by $T \rightarrow F$
(4)	0 2	\$ <b>T</b>	* id \$	shift to 7
(5)	0 2 7	\$ <b>T * </b>	<b>id \$</b>	shift to 5
(6)	0 2 7 5	\$ <b>T * id</b>	\$	reduce by $F \rightarrow id$
(7)	0 2 7 10	\$ <b>T * F</b>	\$	reduce by $T \rightarrow T * F$
(8)	0 2	\$ <b>T</b>	\$	reduce by $E \rightarrow T$
(9)	0 1	\$ <b>E</b>	\$	accept

- With states, we pop state 5 for symbol **id**.
- Which brings state 0 to the top and look for a transition on  $F$ , the head of the production.

**id \* id**



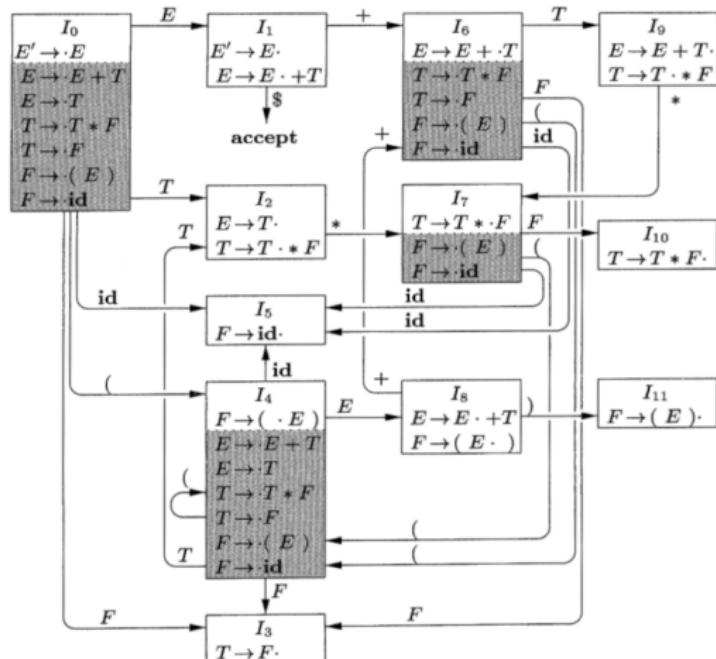
- State 0 has a transition on  $F$  to state 3, so we push state 3, with corresponding symbol  $F$ ; see line (3).

## **id \* id**

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	<b>id * id \$</b>	shift to 5
(2)	0 5	\$ <b>id</b>	* id \$	reduce by $F \rightarrow id$
(3)	0 3	\$ <b>F</b>	* id \$	reduce by $T \rightarrow F$
(4)	0 2	\$ <b>T</b>	* id \$	shift to 7
(5)	0 2 7	\$ <b>T * </b>	<b>id \$</b>	shift to 5
(6)	0 2 7 5	\$ <b>T * id</b>	\$	reduce by $F \rightarrow id$
(7)	0 2 7 10	\$ <b>T * F</b>	\$	reduce by $T \rightarrow T * F$
(8)	0 2	\$ <b>T</b>	\$	reduce by $E \rightarrow T$
(9)	0 1	\$ <b>E</b>	\$	accept

- State 0 has a transition on  $F$  to state 3, so we push state 3, with corresponding symbol  $F$ ; see line (3).

## id \* id



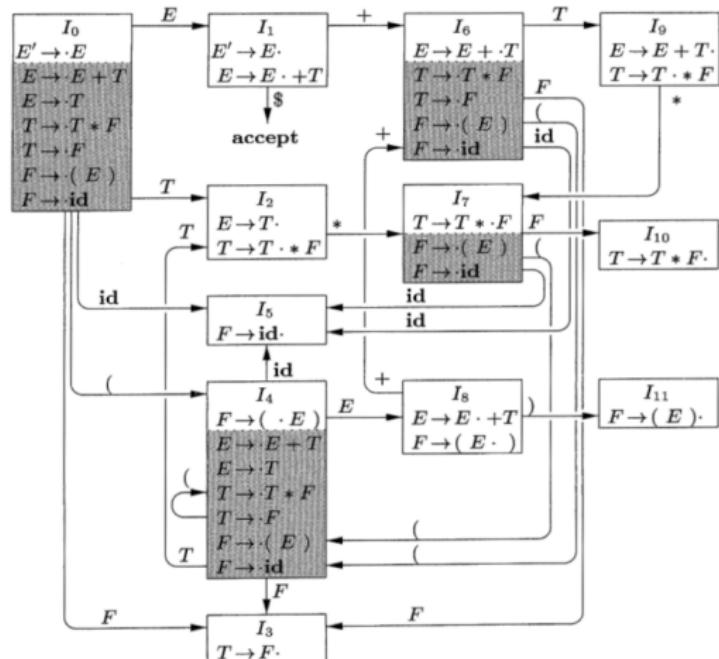
- As another example, consider line (5), with state 7 (symbol  $*$ ) on top of the stack.
- This state has a transition to state 5 on input **id**, so we push state 5 (symbol **id**).

## **id \* id**

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	<b>id</b> * <b>id</b> \$	shift to 5
(2)	0 5	\$ <b>id</b>	* <b>id</b> \$	reduce by $F \rightarrow \text{id}$
(3)	0 3	\$ $F$	* <b>id</b> \$	reduce by $T \rightarrow F$
(4)	0 2	\$ $T$	* <b>id</b> \$	shift to 7
(5)	0 2 7	\$ $T$ *	<b>id</b> \$	shift to 5
(6)	0 2 7 5	\$ $T$ * <b>id</b>	\$	reduce by $F \rightarrow \text{id}$
(7)	0 2 7 10	\$ $T$ * $F$	\$	reduce by $T \rightarrow T * F$
(8)	0 2	\$ $T$	\$	reduce by $E \rightarrow T$
(9)	0 1	\$ $E$	\$	accept

- As another example, consider line (5), with state 7 (symbol \*) on top of the stack.
- This state has a transition to state 5 on input **id**, so we push state 5 (symbol **id**).

## id \* id



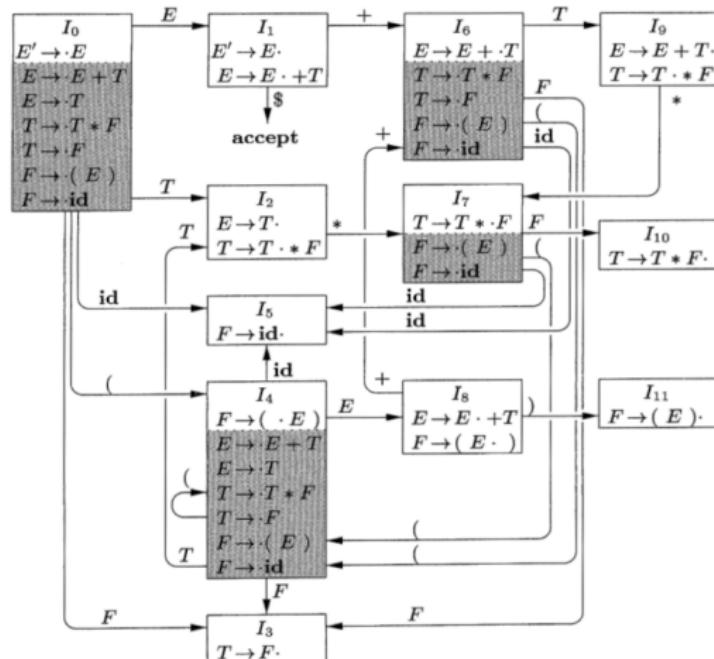
- State 5 has no transitions, so we reduce by  $F \rightarrow id$ .
- When we pop state 5 for the body **id**, state 7 comes to the top of the stack.

## **id \* id**

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	<b>id * id \$</b>	shift to 5
(2)	0 5	\$ <b>id</b>	* <b>id</b> \$	reduce by $F \rightarrow \text{id}$
(3)	0 3	\$ <b>F</b>	* <b>id</b> \$	reduce by $T \rightarrow F$
(4)	0 2	\$ <b>T</b>	* <b>id</b> \$	shift to 7
(5)	0 2 7	\$ <b>T</b> *	<b>id</b> \$	shift to 5
(6)	0 2 7 5	\$ <b>T</b> * <b>id</b>	\$	reduce by $F \rightarrow \text{id}$
(7)	0 2 7 10	\$ <b>T</b> * <b>F</b>	\$	reduce by $T \rightarrow T * F$
(8)	0 2	\$ <b>T</b>	\$	reduce by $E \rightarrow T$
(9)	0 1	\$ <b>E</b>	\$	accept

- State 5 has no transitions, so we reduce by  $F \rightarrow \text{id}$ .
- When we pop state 5 for the body **id**, state 7 comes to the top of the stack.

## id \* id



- Since state 7 has a transition on  $F$  to state 10, we push state 10 (symbol  $F$ ).

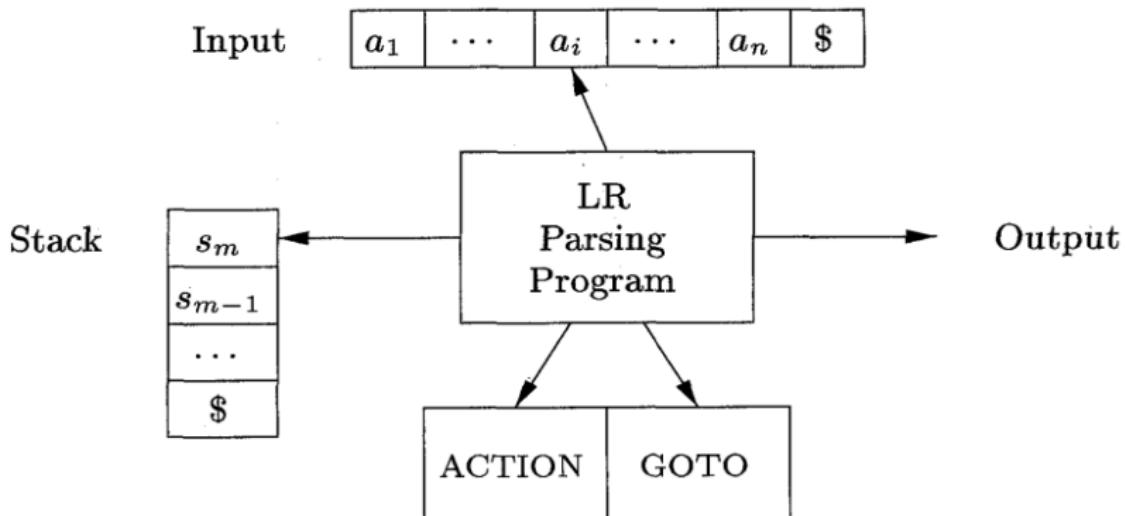
## **id \* id**

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	<b>id * id \$</b>	shift to 5
(2)	0 5	\$ <b>id</b>	* <b>id \$</b>	reduce by $F \rightarrow id$
(3)	0 3	\$ <b>F</b>	* <b>id \$</b>	reduce by $T \rightarrow F$
(4)	0 2	\$ <b>T</b>	* <b>id \$</b>	shift to 7
(5)	0 2 7	\$ <b>T * </b>	<b>id \$</b>	shift to 5
(6)	0 2 7 5	\$ <b>T * id</b>	\$	reduce by $F \rightarrow id$
(7)	0 2 7 10	\$ <b>T * F</b>	\$	reduce by $T \rightarrow T * F$
(8)	0 2	\$ <b>T</b>	\$	reduce by $E \rightarrow T$
(9)	0 1	\$ <b>E</b>	\$	accept

- Since state 7 has a transition on  $F$  to state 10, we push state 10 (symbol  $F$ ).

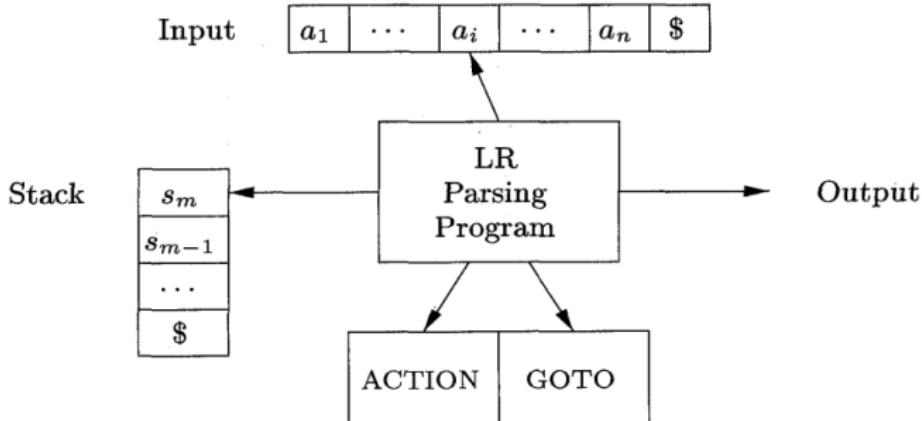
# The LR-Parsing Algorithm

- A schematic of an LR parser is shown.



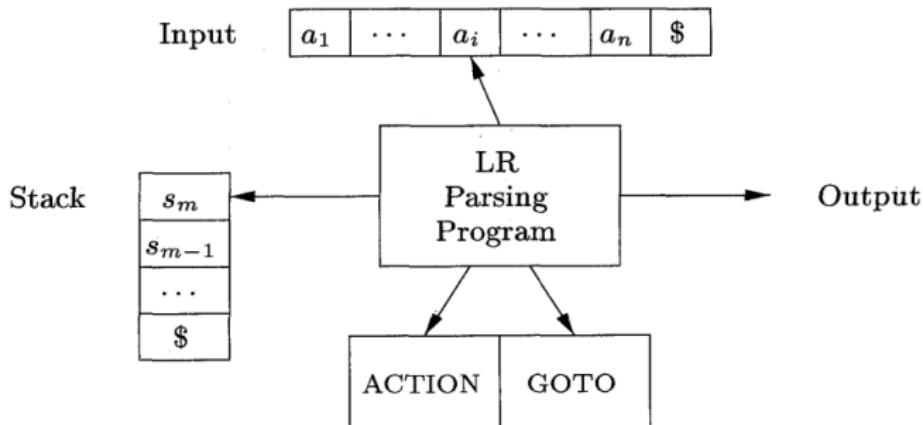
Model of an LR parser

# The LR-Parsing Algorithm — *continued*



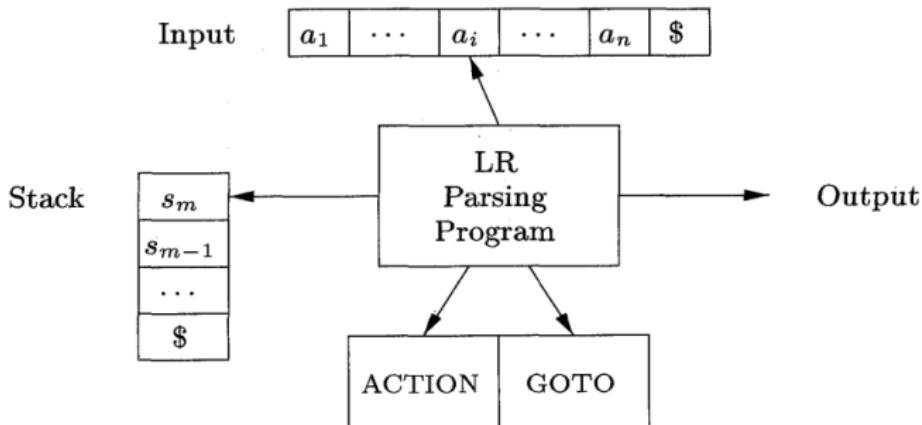
- It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (ACTION and GOTO).
- The driver program is the same for all LR parsers.
- Only the parsing table changes from one parser to another.

# The LR-Parsing Algorithm — *continued*



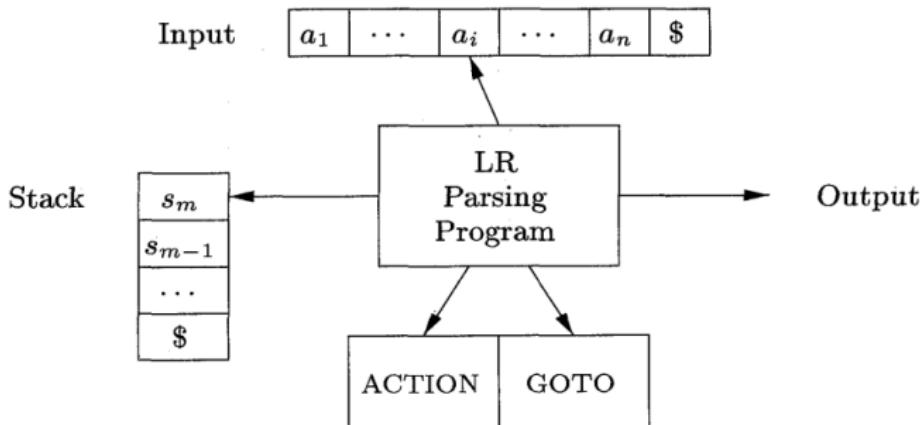
- The parsing program reads characters from an input buffer one at a time.

# The LR-Parsing Algorithm — *continued*



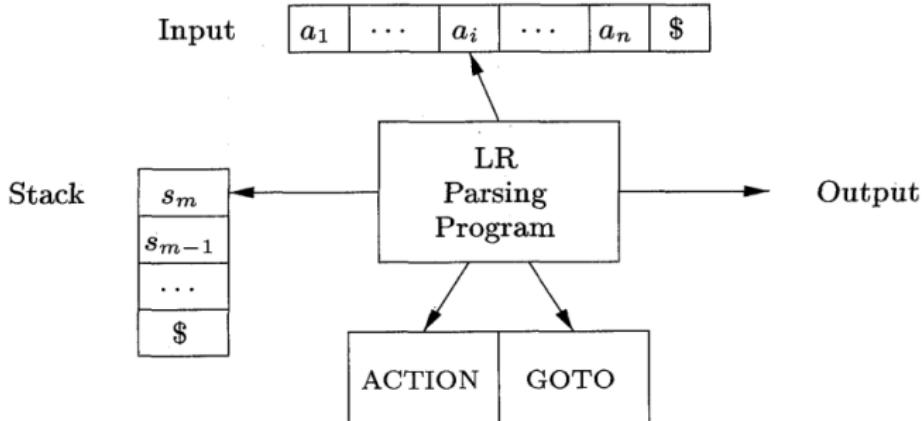
- Where a shift-reduce parser would shift a symbol, an LR parser shifts a state.
- Each state summarizes the information contained in the stack below it.

# The LR-Parsing Algorithm — *continued*



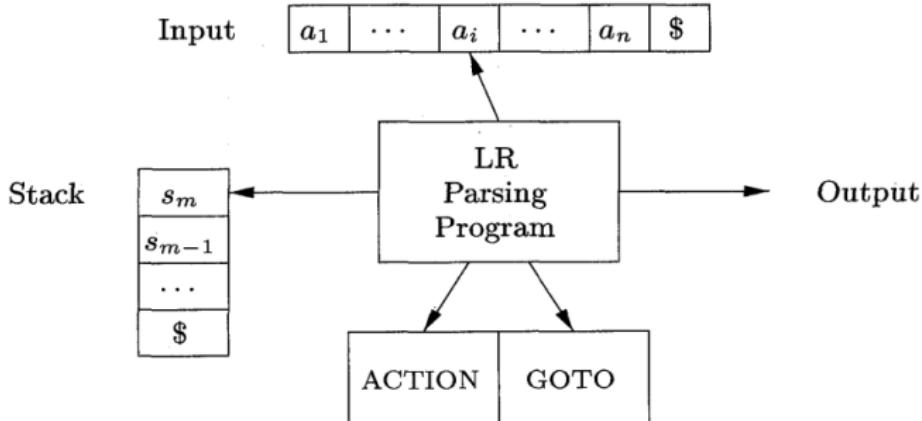
- The stack holds a sequence of states,  $s_0s_1s_2\dots s_m$ , where  $s_m$  is on top.

# The LR-Parsing Algorithm — *continued*



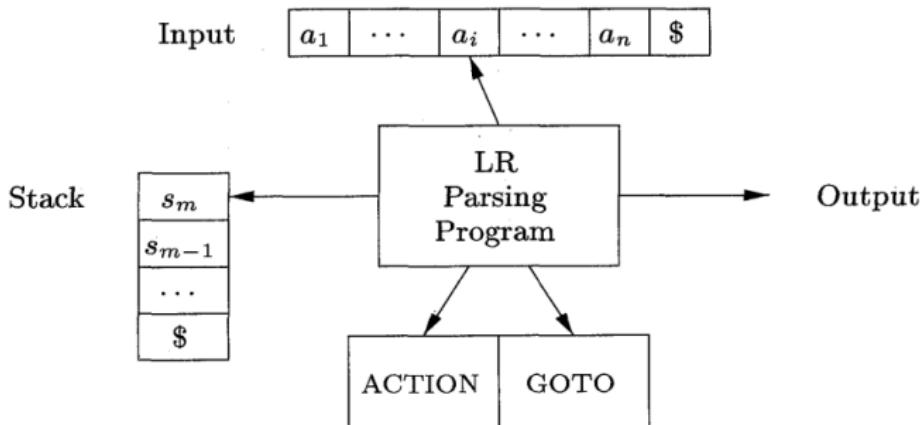
- In the SLR method, the stack holds states from the LR(0) automaton.
- The canonical LR and LALR methods are similar.

# The LR-Parsing Algorithm — *continued*



- By construction, each state has a corresponding grammar symbol.
- Recall that states correspond to sets of items.

# The LR-Parsing Algorithm — *continued*



- That there is a transition from state  $i$  to state  $j$  if  $\text{GOTO}(I_i, X) = I_j$ .
- All transitions to state  $j$  must be for the same grammar symbol  $X$ .
- Thus, each state, except the start state 0, has a unique grammar symbol associated with it.

# The LR-Parsing Algorithm — Structure of the LR Parsing Table

- The parsing table consists of two parts.
- A parsing-action function ACTION.
- A goto function GOTO.

# The LR-Parsing Algorithm — Structure of the LR Parsing Table — *continued*

- 1 The ACTION function takes as arguments a state  $i$  and a terminal  $a$  (or  $\$$ , the input endmarker).
- The value of  $\text{ACTION}[i, a]$  can have one of four forms:
  - (a) Shift  $j$ , where  $j$  is a state.
  - The action taken by the parser effectively shifts input  $a$  to the stack, but uses state  $j$  to represent  $a$ .
  - (b) Reduce  $A \rightarrow \beta$ .
  - The action of the parser effectively reduces  $\beta$  on the top of the stack to head  $A$ .
  - (c) Accept.
  - The parser accepts the input and finishes parsing.
  - (d) Error.
  - The parser discovers an error in its input and takes some corrective action.

# The LR-Parsing Algorithm — Structure of the LR Parsing Table — *continued*

- 1 The ACTION function takes as arguments a state  $i$  and a terminal  $a$  (or  $\$$ , the input endmarker).
- The value of  $\text{ACTION}[i, a]$  can have one of four forms:
  - (a) Shift  $j$ , where  $j$  is a state.
    - The action taken by the parser effectively shifts input  $a$  to the stack, but uses state  $j$  to represent  $a$ .
  - (b) Reduce  $A \rightarrow \beta$ .
    - The action of the parser effectively reduces  $\beta$  on the top of the stack to head  $A$ .
  - (c) Accept.
    - The parser accepts the input and finishes parsing.
  - (d) Error.
    - The parser discovers an error in its input and takes some corrective action.

# The LR-Parsing Algorithm — Structure of the LR Parsing Table — *continued*

- 1 The ACTION function takes as arguments a state  $i$  and a terminal  $a$  (or  $\$$ , the input endmarker).
- The value of  $\text{ACTION}[i, a]$  can have one of four forms:
  - (a) Shift  $j$ , where  $j$  is a state.
    - The action taken by the parser effectively shifts input  $a$  to the stack, but uses state  $j$  to represent  $a$ .
  - (b) Reduce  $A \rightarrow \beta$ .
    - The action of the parser effectively reduces  $\beta$  on the top of the stack to head  $A$ .
  - (c) Accept.
    - The parser accepts the input and finishes parsing.
  - (d) Error.
    - The parser discovers an error in its input and takes some corrective action.

# The LR-Parsing Algorithm — Structure of the LR Parsing Table — *continued*

- 1 The ACTION function takes as arguments a state  $i$  and a terminal  $a$  (or  $\$$ , the input endmarker).
- The value of  $\text{ACTION}[i, a]$  can have one of four forms:
  - (a) Shift  $j$ , where  $j$  is a state.
    - The action taken by the parser effectively shifts input  $a$  to the stack, but uses state  $j$  to represent  $a$ .
  - (b) Reduce  $A \rightarrow \beta$ .
    - The action of the parser effectively reduces  $\beta$  on the top of the stack to head  $A$ .
  - (c) Accept.
    - The parser accepts the input and finishes parsing.
  - (d) Error.
    - The parser discovers an error in its input and takes some corrective action.

## The LR-Parsing Algorithm — Structure of the LR Parsing Table — *continued*

- 2 We extend the GOTO function, defined on sets of items, to states.
- If  $\text{GOTO}[I_i, A] = I_j$ , then GOTO also maps a state  $i$  and a nonterminal  $A$  to state  $j$ .

# The LR-Parsing Algorithm — LR-Parser Configurations

- To describe the behavior of an LR parser, it helps to have a notation representing the complete state of the parser:
  - its stack and
  - the remaining input.
- A configuration of an LR parser is a pair:

$$\left( \underbrace{s_0 s_1 s_2 \dots s_m}_{\text{stack contents}}, \quad \underbrace{a_i a_{i+1} a_{i+2} \dots a_n \$}_{\text{remaining input}} \right)$$

- The first component is the stack contents (top on the right).
- The second component is the remaining input.

# The LR-Parsing Algorithm — LR-Parser Configurations

- To describe the behavior of an LR parser, it helps to have a notation representing the complete state of the parser:
  - its stack and
  - the remaining input.
- A configuration of an LR parser is a pair:

$$\left( \underbrace{s_0 s_1 s_2 \dots s_m}_{\text{stack contents}}, \quad \underbrace{a_i a_{i+1} a_{i+2} \dots a_n \$}_{\text{remaining input}} \right)$$

- The first component is the stack contents (top on the right).
- The second component is the remaining input.

# The LR-Parsing Algorithm — LR-Parser Configurations

$(s_0s_1s_2 \dots s_m, \quad a_ia_{i+1}a_{i+2} \dots a_n \$)$

- This configuration represents the right-sentential form,

$X_1X_2 \dots X_ma_ia_{i+1}a_{i+2} \dots a_n \$$

- This is essentially the same way as a shift-reduce parser would.

# The LR-Parsing Algorithm — LR-Parser Configurations

$(s_0s_1s_2 \dots s_m, \quad a_ia_{i+1}a_{i+2} \dots a_n \$)$

- The only difference is that instead of grammar symbols, the stack holds states from which grammar symbols can be recovered.
- That is,  $X_i$  is the grammar symbol represented by state  $s_i$ .
- $s_o$ , the start state of the parser, does not represent a grammar symbol, and serves as a bottom-of-stack marker, as well as playing an important role in the parse.

# The LR-Parsing Algorithm — Behavior of the LR Parser

- The next move of the parser from the configuration above is determined,
  - by reading  $a_i$ , the current input symbol,
  - and  $s_m$ , the state on top of the stack,
  - and then consulting the entry  $\text{ACTION}[s_m, a_i]$  in the parsing action table.
- The configurations resulting after each of the four types of move.

# The LR-Parsing Algorithm — Behavior of the LR Parser — *continued*

$(s_0s_1s_2 \dots s_m, a_i a_{i+1} a_{i+2} \dots a_n \$)$

## 1. If $\text{ACTION}[s_m, a_i] = \text{shift } s$

- The parser executes a shift move.
- It shifts the next state  $s$  onto the stack.
- Enters the configuration,

$(s_0s_1s_2 \dots s_m s, a_{i+1} a_{i+2} \dots a_n \$)$

- The symbol  $a_i$  need not be held on the stack.
- It can be recovered from  $s$ , if needed (which in practice it never is).
- The current input symbol is now  $a_{i+1}$ .

# The LR-Parsing Algorithm — Behavior of the LR Parser — *continued*

$(s_0s_1s_2 \dots s_m, a_i a_{i+1} a_{i+2} \dots a_n \$)$

2. If  $\text{ACTION}[s_m, a_i] = \text{reduce } A \rightarrow \beta$

- The parser executes a reduce move.
- Enters the configuration,

$(s_0s_1s_2 \dots s_{m-r}s, a_i a_{i+1} a_{i+2} \dots a_n \$)$

- $r$  is the length of  $\beta$ .
- $s = \text{GOTO}[s_{m-r}, A]$ .

# The LR-Parsing Algorithm — Behavior of the LR Parser — *continued*

$(s_0 s_1 s_2 \dots s_m, a_i a_{i+1} a_{i+2} \dots a_n \$)$

## 2. If $\text{ACTION}[s_m, a_i] = \text{reduce } A \rightarrow \beta$

- Here the parser first popped  $r$  state symbols off the stack, exposing state  $s_{m-r}$ .
- The parser then pushed  $s$ , the entry for  $\text{GOTO}[s_{m-r}, A]$ , onto the stack.
- The current input symbol is not changed in a reduce move.

# The LR-Parsing Algorithm — Behavior of the LR Parser — *continued*

$(s_0s_1s_2 \dots s_m, a_i a_{i+1} a_{i+2} \dots a_n \$)$

2. If  $\text{ACTION}[s_m, a_i] = \text{reduce } A \rightarrow \beta$

- For the LR parsers we shall construct,  $X_{m-r+1} \dots X_m$ , the sequence of grammar symbols corresponding to the states popped off the stack, will always match  $\beta$ , the right side of the reducing production.

# The LR-Parsing Algorithm — Behavior of the LR Parser — *continued*

$(s_0s_1s_2 \dots s_m, a_i a_{i+1} a_{i+2} \dots a_n \$)$

## 2. If $\text{ACTION}[s_m, a_i] = \text{reduce } A \rightarrow \beta$

- The output of an LR parser is generated after a reduce move by executing the semantic action associated with the reducing production.
- For the time being, we shall assume the output consists of just printing the reducing production.

# The LR-Parsing Algorithm — Behavior of the LR Parser — *continued*

$(s_0s_1s_2 \dots s_m, a_i a_{i+1} a_{i+2} \dots a_n \$)$

3. If  $\text{ACTION}[s_m, a_i] = \text{accept}$ 
  - Parsing is completed.

# The LR-Parsing Algorithm — Behavior of the LR Parser — *continued*

$(s_0 s_1 s_2 \dots s_m, a_i a_{i+1} a_{i+2} \dots a_n \$)$

## 4. If $\text{ACTION}[s_m, a_i] = \text{error}$

- The parser has discovered an error and calls an error recovery routine.

# The LR-Parsing Algorithm — Behavior of the LR Parser — *continued*

```
let  $a$  be the first symbol of  $w\$$ ;
while(1) { /* repeat forever */
    let  $s$  be the state on top of the stack;
    if (  $\text{ACTION}[s, a] = \text{shift } t$  ) {
        push  $t$  onto the stack;
        let  $a$  be the next input symbol;
    } else if (  $\text{ACTION}[s, a] = \text{reduce } A \rightarrow \beta$  ) {
        pop  $|\beta|$  symbols off the stack;
        let state  $t$  now be on top of the stack;
        push  $\text{GOTO}[t, A]$  onto the stack;
        output the production  $A \rightarrow \beta$ ;
    } else if (  $\text{ACTION}[s, a] = \text{accept}$  ) break; /* parsing is done */
    else call error-recovery routine;
}
```

Figure 4.36: LR-parsing program

- The LR-parsing algorithm.

# The LR-Parsing Algorithm — Behavior of the LR Parser — *continued*

```
let  $a$  be the first symbol of  $w\$$ ;  
while(1) { /* repeat forever */  
    let  $s$  be the state on top of the stack;  
    if (  $\text{ACTION}[s, a] = \text{shift } t$  ) {  
        push  $t$  onto the stack;  
        let  $a$  be the next input symbol;  
    } else if (  $\text{ACTION}[s, a] = \text{reduce } A \rightarrow \beta$  ) {  
        pop  $|\beta|$  symbols off the stack;  
        let state  $t$  now be on top of the stack;  
        push  $\text{GOTO}[t, A]$  onto the stack;  
        output the production  $A \rightarrow \beta$ ;  
    } else if (  $\text{ACTION}[s, a] = \text{accept}$  ) break; /* parsing is done */  
    else call error-recovery routine;  
}
```

Figure 4.36: LR-parsing program

- All LR parsers behave in this fashion.
- The only difference between one LR parser and another is the information in the ACTION and GOTO fields of the parsing table.

# The LR-Parsing Algorithm — Behavior of the LR Parser — *continued*

```
let  $a$  be the first symbol of  $w\$$ ;  
while(1) { /* repeat forever */  
    let  $s$  be the state on top of the stack;  
    if (  $\text{ACTION}[s, a] = \text{shift } t$  ) {  
        push  $t$  onto the stack;  
        let  $a$  be the next input symbol;  
    } else if (  $\text{ACTION}[s, a] = \text{reduce } A \rightarrow \beta$  ) {  
        pop  $|\beta|$  symbols off the stack;  
        let state  $t$  now be on top of the stack;  
        push  $\text{GOTO}[t, A]$  onto the stack;  
        output the production  $A \rightarrow \beta$ ;  
    } else if (  $\text{ACTION}[s, a] = \text{accept}$  ) break; /* parsing is done */  
    else call error-recovery routine;  
}
```

- **INPUT:** An input string  $w$  and an LR-parsing table with functions  $\text{ACTION}$  and  $\text{GOTO}$  for a grammar  $G$ .

# The LR-Parsing Algorithm — Behavior of the LR Parser — *continued*

```
let  $a$  be the first symbol of  $w\$$ ;  
while(1) { /* repeat forever */  
    let  $s$  be the state on top of the stack;  
    if (  $\text{ACTION}[s, a] = \text{shift } t$  ) {  
        push  $t$  onto the stack;  
        let  $a$  be the next input symbol;  
    } else if (  $\text{ACTION}[s, a] = \text{reduce } A \rightarrow \beta$  ) {  
        pop  $|\beta|$  symbols off the stack;  
        let state  $t$  now be on top of the stack;  
        push  $\text{GOTO}[t, A]$  onto the stack;  
        output the production  $A \rightarrow \beta$ ;  
    } else if (  $\text{ACTION}[s, a] = \text{accept}$  ) break; /* parsing is done */  
    else call error-recovery routine;  
}
```

- **OUTPUT:** If  $w$  is in  $L(G)$ , the reduction steps of a bottom-up parse for  $w$ ; otherwise, an error indication.

# The LR-Parsing Algorithm — Behavior of the LR Parser — *continued*

```
let  $a$  be the first symbol of  $w\$$ ;  
while(1) { /* repeat forever */  
    let  $s$  be the state on top of the stack;  
    if (  $\text{ACTION}[s, a] = \text{shift } t$  ) {  
        push  $t$  onto the stack;  
        let  $a$  be the next input symbol;  
    } else if (  $\text{ACTION}[s, a] = \text{reduce } A \rightarrow \beta$  ) {  
        pop  $|\beta|$  symbols off the stack;  
        let state  $t$  now be on top of the stack;  
        push  $\text{GOTO}[t, A]$  onto the stack;  
        output the production  $A \rightarrow \beta$ ;  
    } else if (  $\text{ACTION}[s, a] = \text{accept}$  ) break; /* parsing is done */  
    else call error-recovery routine;  
}
```

- **METHOD:** Initially, the parser has  $s_0$  on its stack, where  $s_0$  is the initial state, and  $w$  in the input buffer.
- The parser then executes the above program.

# Example

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		

Figure 4.37: Parsing table for expression grammar

- The ACTION and GOTO functions of an LR-parsing table for the expression grammar.

## Example — *continued*

STATE	ACTION					GOTO			
	<b>id</b>	+	*	(	)	\$	<i>E</i>	<i>T</i>	<i>F</i>
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			

- (1)  $E \rightarrow E + T$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow T * F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow \mathbf{id}$

■ The codes for the actions are:

1.  $s_i$  means shift and stack state  $i$ ,
2.  $r_j$  means reduce by the production numbered  $j$ ,
3. acc means accept,
4. blank means error.

## Example — *continued*

STATE	ACTION					GOTO			
	<b>id</b>	+	*	(	)	\$	<i>E</i>	<i>T</i>	<i>F</i>
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			

- (1)  $E \rightarrow E + T$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow T * F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow \mathbf{id}$

- Note that the value of  $\text{GOTO}[s, a]$  for terminal *a* is found in the ACTION field connected with the shift action on input *a* for state *s*.
- The GOTO field gives  $\text{GOTO}[s, A]$  for nonterminals *A*.

## Example — *continued*

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		<b>id * id + id \$</b>	shift
(2)	0 5	<b>id</b>	* <b>id + id \$</b>	reduce by $F \rightarrow \mathbf{id}$
(3)	0 3	<b>F</b>	* <b>id + id \$</b>	reduce by $T \rightarrow F$
(4)	0 2	<b>T</b>	* <b>id + id \$</b>	shift
(5)	0 2 7	<b>T *</b>	<b>id + id \$</b>	shift
(6)	0 2 7 5	<b>T * id</b>	+ <b>id \$</b>	reduce by $F \rightarrow \mathbf{id}$
(7)	0 2 7 10	<b>T * F</b>	+ <b>id \$</b>	reduce by $T \rightarrow T * F$
(8)	0 2	<b>T</b>	+ <b>id \$</b>	reduce by $E \rightarrow T$
(9)	0 1	<b>E</b>	+ <b>id \$</b>	shift
(10)	0 1 6	<b>E +</b>	<b>id \$</b>	shift
(11)	0 1 6 5	<b>E + id</b>	<b>\$</b>	reduce by $F \rightarrow \mathbf{id}$
(12)	0 1 6 3	<b>E + F</b>	<b>\$</b>	reduce by $T \rightarrow F$
(13)	0 1 6 9	<b>E + T</b>	<b>\$</b>	reduce by $E \rightarrow E + T$
(14)	0 1	<b>E</b>	<b>\$</b>	accept

Moves of an LR parser on **id \* id + id**

**id \* id + id**

- On input **id \* id + id**, the sequence of stack and input contents is shown.

## Example — *continued*

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		<b>id</b> * <b>id</b> + <b>id</b> \$	shift
(2)	0 5	<b>id</b>	* <b>id</b> + <b>id</b> \$	reduce by $F \rightarrow \text{id}$
(3)	0 3	<b>F</b>	* <b>id</b> + <b>id</b> \$	reduce by $T \rightarrow F$
(4)	0 2	<b>T</b>	* <b>id</b> + <b>id</b> \$	shift
(5)	0 2 7	<b>T</b> *	<b>id</b> + <b>id</b> \$	shift
(6)	0 2 7 5	<b>T</b> * <b>id</b>	+ <b>id</b> \$	reduce by $F \rightarrow \text{id}$
(7)	0 2 7 10	<b>T</b> * <b>F</b>	+ <b>id</b> \$	reduce by $T \rightarrow T * F$
(8)	0 2	<b>T</b>	+ <b>id</b> \$	reduce by $E \rightarrow T$
(9)	0 1	<b>E</b>	+ <b>id</b> \$	shift
(10)	0 1 6	<b>E</b> +	<b>id</b> \$	shift
(11)	0 1 6 5	<b>E</b> + <b>id</b>	\$	reduce by $F \rightarrow \text{id}$
(12)	0 1 6 3	<b>E</b> + <b>F</b>	\$	reduce by $T \rightarrow F$
(13)	0 1 6 9	<b>E</b> + <b>T</b>	\$	reduce by $E \rightarrow E + T$
(14)	0 1	<b>E</b>	\$	accept

**id \* id + id**

- (1)  $E \rightarrow E + T$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow T * F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow \text{id}$

- Also shown for clarity, are the sequences of grammar symbols corresponding to the states held on the stack.
- For example, at line (1) the LR parser is in state 0, the initial state with no grammar symbol, and with **id** the first input symbol.

## Example — *continued*

STATE	ACTION					GOTO			
	<b>id</b>	+	*	(	)	\$	<i>E</i>	<i>T</i>	<i>F</i>
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			

**id \* id + id**

- (1)  $E \rightarrow E + T$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow T * F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow \mathbf{id}$

- Also shown for clarity, are the sequences of grammar symbols corresponding to the states held on the stack.
- For example, at line (1) the LR parser is in state 0, the initial state with no grammar symbol, and with **id** the first input symbol.

## Example — *continued*

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		<b>id</b> * <b>id</b> + <b>id</b> \$	shift
(2)	0 5	<b>id</b>	* <b>id</b> + <b>id</b> \$	reduce by $F \rightarrow \text{id}$
(3)	0 3	<b>F</b>	* <b>id</b> + <b>id</b> \$	reduce by $T \rightarrow F$
(4)	0 2	<b>T</b>	* <b>id</b> + <b>id</b> \$	shift
(5)	0 2 7	<b>T</b> *	<b>id</b> + <b>id</b> \$	shift
(6)	0 2 7 5	<b>T</b> * <b>id</b>	+ <b>id</b> \$	reduce by $F \rightarrow \text{id}$
(7)	0 2 7 10	<b>T</b> * <b>F</b>	+ <b>id</b> \$	reduce by $T \rightarrow T * F$
(8)	0 2	<b>T</b>	+ <b>id</b> \$	reduce by $E \rightarrow T$
(9)	0 1	<b>E</b>	+ <b>id</b> \$	shift
(10)	0 1 6	<b>E</b> +	<b>id</b> \$	shift
(11)	0 1 6 5	<b>E</b> + <b>id</b>	\$	reduce by $F \rightarrow \text{id}$
(12)	0 1 6 3	<b>E</b> + <b>F</b>	\$	reduce by $T \rightarrow F$
(13)	0 1 6 9	<b>E</b> + <b>T</b>	\$	reduce by $E \rightarrow E + T$
(14)	0 1	<b>E</b>	\$	accept

**id \* id + id**

- (1)  $E \rightarrow E + T$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow T * F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow \text{id}$

- The action in row 0 and column **id** of the action field of is *s5*, meaning shift by pushing state 5.

## Example — *continued*

STATE	ACTION					GOTO			
	<b>id</b>	+	*	(	)	\$	<i>E</i>	<i>T</i>	<i>F</i>
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			

**id \* id + id**

- (1)  $E \rightarrow E + T$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow T * F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow \mathbf{id}$

- The action in row 0 and column **id** of the action field of is *s5*, meaning shift by pushing state 5.

## Example — *continued*

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		<b>id</b> * <b>id</b> + <b>id</b> \$	shift
(2)	0 5	<b>id</b>	* <b>id</b> + <b>id</b> \$	reduce by $F \rightarrow \text{id}$
(3)	0 3	<b>F</b>	* <b>id</b> + <b>id</b> \$	reduce by $T \rightarrow F$
(4)	0 2	<b>T</b>	* <b>id</b> + <b>id</b> \$	shift
(5)	0 2 7	<b>T</b> *	<b>id</b> + <b>id</b> \$	shift
(6)	0 2 7 5	<b>T</b> * <b>id</b>	+ <b>id</b> \$	reduce by $F \rightarrow \text{id}$
(7)	0 2 7 10	<b>T</b> * <b>F</b>	+ <b>id</b> \$	reduce by $T \rightarrow T * F$
(8)	0 2	<b>T</b>	+ <b>id</b> \$	reduce by $E \rightarrow T$
(9)	0 1	<b>E</b>	+ <b>id</b> \$	shift
(10)	0 1 6	<b>E</b> +	<b>id</b> \$	shift
(11)	0 1 6 5	<b>E</b> + <b>id</b>	\$	reduce by $F \rightarrow \text{id}$
(12)	0 1 6 3	<b>E</b> + <b>F</b>	\$	reduce by $T \rightarrow F$
(13)	0 1 6 9	<b>E</b> + <b>T</b>	\$	reduce by $E \rightarrow E + T$
(14)	0 1	<b>E</b>	\$	accept

**id \* id + id**

- (1)  $E \rightarrow E + T$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow T * F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow \text{id}$

- That is what has happened at line (2).
- The state symbol 5 has been pushed onto the stack.
- And **id** has been removed from the input.

## Example — *continued*

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			

**id \* id + id**

- (1)  $E \rightarrow E + T$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow T * F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow \mathbf{id}$

- That is what has happened at line (2).
- The state symbol 5 has been pushed onto the stack.
- And **id** has been removed from the input.

## Example — *continued*

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		<b>id</b> * <b>id</b> + <b>id</b> \$	shift
(2)	0 5	<b>id</b>	* <b>id</b> + <b>id</b> \$	reduce by $F \rightarrow \mathbf{id}$
(3)	0 3	<b>F</b>	* <b>id</b> + <b>id</b> \$	reduce by $T \rightarrow F$
(4)	0 2	<b>T</b>	* <b>id</b> + <b>id</b> \$	shift
(5)	0 2 7	<b>T</b> *	<b>id</b> + <b>id</b> \$	shift
(6)	0 2 7 5	<b>T</b> * <b>id</b>	+ <b>id</b> \$	reduce by $F \rightarrow \mathbf{id}$
(7)	0 2 7 10	<b>T</b> * <b>F</b>	+ <b>id</b> \$	reduce by $T \rightarrow T * F$
(8)	0 2	<b>T</b>	+ <b>id</b> \$	reduce by $E \rightarrow T$
(9)	0 1	<b>E</b>	+ <b>id</b> \$	shift
(10)	0 1 6	<b>E</b> +	<b>id</b> \$	shift
(11)	0 1 6 5	<b>E</b> + <b>id</b>	\$	reduce by $F \rightarrow \mathbf{id}$
(12)	0 1 6 3	<b>E</b> + <b>F</b>	\$	reduce by $T \rightarrow F$
(13)	0 1 6 9	<b>E</b> + <b>T</b>	\$	reduce by $E \rightarrow E + T$
(14)	0 1	<b>E</b>	\$	accept

**id \* id + id**

- (1)  $E \rightarrow E + T$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow T * F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow \mathbf{id}$

- Then, \* becomes the current input symbol.
- The action of state 5 on input \* is to reduce by  $F \rightarrow \mathbf{id}$ .

## Example — *continued*

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			

**id \* id + id**

- (1)  $E \rightarrow E + T$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow T * F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow \mathbf{id}$

- Then, \* becomes the current input symbol.
- The action of state 5 on input \* is to reduce by  $F \rightarrow \mathbf{id}$ .

## Example — *continued*

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		<b>id</b> * <b>id</b> + <b>id</b> \$	shift
(2)	0 5	<b>id</b>	* <b>id</b> + <b>id</b> \$	reduce by $F \rightarrow \text{id}$
(3)	0 3	<b>F</b>	* <b>id</b> + <b>id</b> \$	reduce by $T \rightarrow F$
(4)	0 2	<b>T</b>	* <b>id</b> + <b>id</b> \$	shift
(5)	0 2 7	<b>T</b> *	<b>id</b> + <b>id</b> \$	shift
(6)	0 2 7 5	<b>T</b> * <b>id</b>	+ <b>id</b> \$	reduce by $F \rightarrow \text{id}$
(7)	0 2 7 10	<b>T</b> * <b>F</b>	+ <b>id</b> \$	reduce by $T \rightarrow T * F$
(8)	0 2	<b>T</b>	+ <b>id</b> \$	reduce by $E \rightarrow T$
(9)	0 1	<b>E</b>	+ <b>id</b> \$	shift
(10)	0 1 6	<b>E</b> +	<b>id</b> \$	shift
(11)	0 1 6 5	<b>E</b> + <b>id</b>	\$	reduce by $F \rightarrow \text{id}$
(12)	0 1 6 3	<b>E</b> + <b>F</b>	\$	reduce by $T \rightarrow F$
(13)	0 1 6 9	<b>E</b> + <b>T</b>	\$	reduce by $E \rightarrow E + T$
(14)	0 1	<b>E</b>	\$	accept

**id \* id + id**

- (1)  $E \rightarrow E + T$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow T * F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow \text{id}$

- One state symbol is popped off the stack.
- State 0 is then exposed.
- Since the goto of state 0 on  $F$  is 3, state 3 is pushed onto the stack.

## Example — *continued*

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			

**id \* id + id**

- (1)  $E \rightarrow E + T$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow T * F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow \mathbf{id}$

- One state symbol is popped off the stack.
- State 0 is then exposed.
- Since the goto of state 0 on  $F$  is 3, state 3 is pushed onto the stack.

## Example — *continued*

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		<b>id</b> * <b>id</b> + <b>id</b> \$	shift
(2)	0 5	<b>id</b>	* <b>id</b> + <b>id</b> \$	reduce by $F \rightarrow \mathbf{id}$
(3)	0 3	<b>F</b>	* <b>id</b> + <b>id</b> \$	reduce by $T \rightarrow F$
(4)	0 2	<b>T</b>	* <b>id</b> + <b>id</b> \$	shift
(5)	0 2 7	<b>T</b> *	<b>id</b> + <b>id</b> \$	shift
(6)	0 2 7 5	<b>T</b> * <b>id</b>	+ <b>id</b> \$	reduce by $F \rightarrow \mathbf{id}$
(7)	0 2 7 10	<b>T</b> * <b>F</b>	+ <b>id</b> \$	reduce by $T \rightarrow T * F$
(8)	0 2	<b>T</b>	+ <b>id</b> \$	reduce by $E \rightarrow T$
(9)	0 1	<b>E</b>	+ <b>id</b> \$	shift
(10)	0 1 6	<b>E</b> +	<b>id</b> \$	shift
(11)	0 1 6 5	<b>E</b> + <b>id</b>	\$	reduce by $F \rightarrow \mathbf{id}$
(12)	0 1 6 3	<b>E</b> + <b>F</b>	\$	reduce by $T \rightarrow F$
(13)	0 1 6 9	<b>E</b> + <b>T</b>	\$	reduce by $E \rightarrow E + T$
(14)	0 1	<b>E</b>	\$	accept

**id \* id + id**

- (1)  $E \rightarrow E + T$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow T * F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow \mathbf{id}$

- We now have the configuration in line (3).
- Each of the remaining moves is determined similarly.

## Example — *continued*

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			

**id \* id + id**

- (1)  $E \rightarrow E + T$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow T * F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow \mathbf{id}$

- We now have the configuration in line (3).
- Each of the remaining moves is determined similarly.

# Constructing SLR-Parsing Tables

- The SLR method for constructing parsing tables is a good starting point for studying LR parsing.
- We shall refer to the parsing table constructed by this method as an SLR table.
- An LR parser using an SLR-parsing table is an SLR parser.
- The other two methods augment the SLR method with lookahead information.

# Constructing SLR-Parsing Tables

- The SLR method for constructing parsing tables is a good starting point for studying LR parsing.
- We shall refer to the parsing table constructed by this method as an SLR table.
- An LR parser using an SLR-parsing table is an SLR parser.
- **The other two methods augment the SLR method with lookahead information.**

- The SLR method begins with LR(0) items and LR(0) automata.
- That is, given a grammar,  $G$ , we augment  $G$  to produce  $G'$ , with a new start symbol  $S'$ .
- From  $G'$ , we construct  $C$ , the canonical collection of sets of items for  $G'$  together with the GOTO function.

## Constructing SLR-Parsing Tables — *continued*

- The ACTION and GOTO entries in the parsing table are then constructed using the following algorithm.
- It requires us to know  $\text{FOLLOW}(A)$  for each nonterminal  $A$  of a grammar.

# Constructing SLR-Parsing Tables — *continued*

## Algorithm: Constructing an SLR-parsing table

- **INPUT:** An augmented grammar  $G'$ .
- **OUTPUT:** The SLR-parsing table functions ACTION and GOTO for  $G'$ .

# Constructing SLR-Parsing Tables — *continued*

**Algorithm:** Constructing an SLR-parsing table

**METHOD:**

1. Construct  $C = \{I_0, I_1, I_2, \dots, I_n\}$ , the collection of sets of LR(0) items for  $G'$ .

## Algorithm: Constructing an SLR-parsing table

### METHOD:

2. State  $i$  is constructed from  $I_i$ .
  - The parsing actions for state  $i$  are determined as follows:
    - (a) If  $[A \rightarrow \alpha \cdot a\beta]$  is in  $I_i$ , and  $\text{GOTO}(I_i, a) = I_j$ , then set  $\text{ACTION}[i, a]$  to “shift  $j$ .”
      - Here  $a$  must be a terminal.
    - (b) If  $[A \rightarrow \alpha \cdot]$  is in  $I_i$ , then set  $\text{ACTION}[i, a]$  to “reduce  $A \rightarrow \alpha$ ” for all  $a$  in  $\text{FOLLOW}(A)$ .
      - Here  $A$  may not be  $S'$ .
    - (c) If  $[S' \rightarrow S \cdot]$  is in  $I_i$ , then set  $\text{ACTION}[i, \$]$  to “accept.”
  - If any conflicting actions result from the above rules, we say the grammar is not SLR(1).
    - The algorithm fails to produce a parser in this case.

**Algorithm:** Constructing an SLR-parsing table

**METHOD:**

3. The goto transitions for state  $i$  are constructed for all nonterminals  $A$  using the rule: If  $\text{GOTO}(I_i, A) = I_j$ , then  $\text{GOTO}[i, A] = j$ .

**Algorithm:** Constructing an SLR-parsing table

**METHOD:**

4. All entries not defined by rules (2) and (3) are made “error.”

**Algorithm:** Constructing an SLR-parsing table

**METHOD:**

5. The initial state of the parser is the one constructed from the set of items containing  $[S' \rightarrow \cdot S]$ .

## Constructing SLR-Parsing Tables — *continued*

- The parsing table consisting of the ACTION and GOTO functions determined by the previous algorithm is called the SLR(1) table for  $G$ .
- An LR parser using the SLR(1) table for  $G$  is called the SLR(1) parser for  $G$ .
- And a grammar having an SLR(1) parsing table is said to be SLR(1).
- We usually omit the “(1)” after the “SLR,” since we shall not deal here with parsers having more than one symbol of lookahead.

# Example

- Let us construct the SLR table for the augmented expression grammar.

$E' \rightarrow E$

(1)  $E \rightarrow E + T$

$E \rightarrow E + T \mid T$

(2)  $E \rightarrow T$

$T \rightarrow T * F \mid F$

(3)  $T \rightarrow T * F$

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

(4)  $T \rightarrow F$

(5)  $F \rightarrow ( E )$

(6)  $F \rightarrow \text{id}$

## Example — *continued*

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

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

(1)  $E \rightarrow E + T$

(2)  $E \rightarrow T$

(3)  $T \rightarrow T * F$

(4)  $T \rightarrow F$

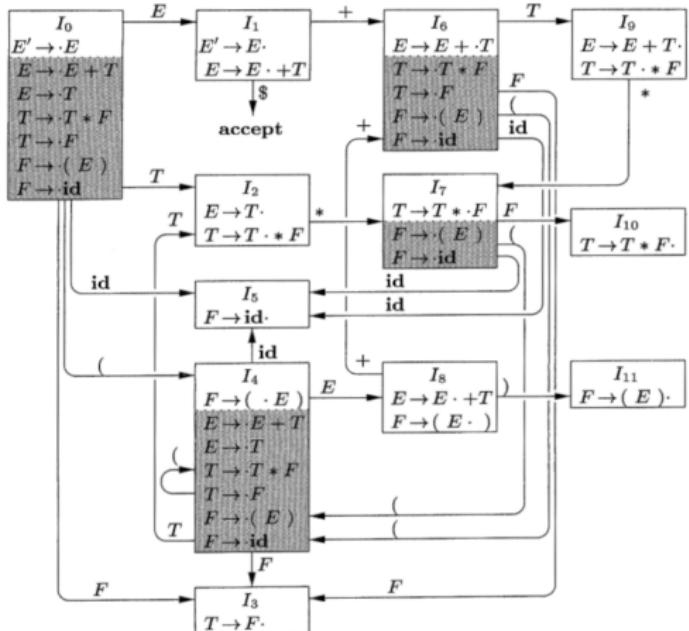
(5)  $F \rightarrow (E)$

(6)  $F \rightarrow \mathbf{id}$

■ First consider the set of items  $I_0$ .

2. State  $i$  is constructed from  $I_i$ .
  - The parsing actions for state  $i$  are determined as follows:
    - (a) If  $[A \rightarrow \alpha \cdot a\beta]$  is in  $I_i$ , and  $\text{GOTO}(I_i, a) = I_j$ , then set  $\text{ACTION}[i, a]$  to "shift  $j$ ."
      - Here  $a$  must be a terminal.
    - (b) If  $[A \rightarrow \alpha \cdot \cdot]$  is in  $I_i$ , then set  $\text{ACTION}[i, a]$  to "reduce  $A \rightarrow \alpha$ " for all  $a$  in  $\text{FOLLOW}(A)$ .
      - Here  $A$  may not be  $S'$ .
    - (c) If  $[S' \rightarrow S \cdot \cdot]$  is in  $I_i$ , then set  $\text{ACTION}[i, \$]$  to "accept."
  - If any conflicting actions result from the above rules, we say the grammar is not SLR(1).
    - The algorithm fails to produce a parser in this case.

- 
- The item  $F \rightarrow \cdot(E)$  gives rise to the entry  $\text{ACTION}[0, ()] = \text{shift 4.}$
  - And the item  $F \rightarrow \cdot\text{id}$  to the entry  $\text{ACTION}[0, \text{id}] = \text{shift 5.}$
  - Other items in  $I_0$  yield no actions.



- (1)  $E \rightarrow E + T$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow T * F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow \mathbf{id}$

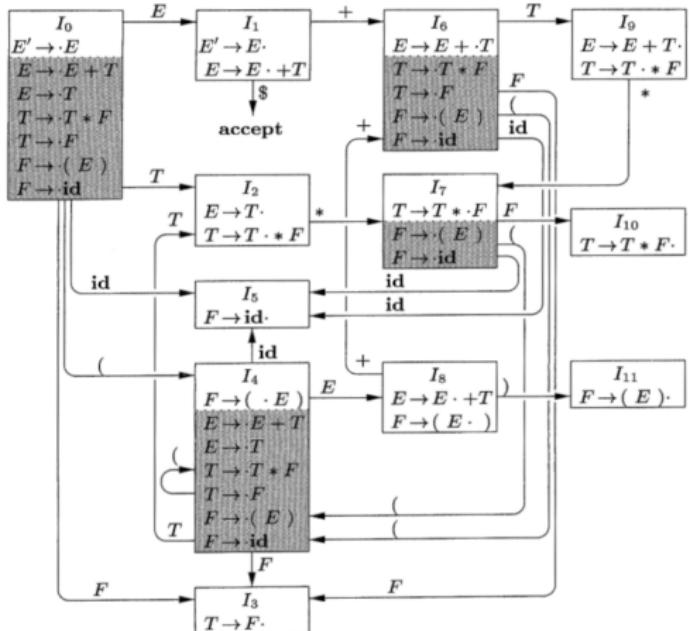
- The item  $F \rightarrow \cdot(E)$  gives rise to the entry ACTION[0, ()] = shift 4.
- And the item  $F \rightarrow \cdot\mathbf{id}$  to the entry ACTION[0, id] = shift 5.
- Other items in  $I_0$  yield no actions.

STATE	ACTION					GOTO			
	<b>id</b>	+	*	(	)	\$	<i>E</i>	<i>T</i>	<i>F</i>
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		

- The item  $F \rightarrow \cdot(E)$  gives rise to the entry ACTION[0, ()] = shift 4.
- And the item  $F \rightarrow \cdot\text{id}$  to the entry ACTION[0, **id**] = shift 5.
- Other items in  $I_0$  yield no actions.

2. State  $i$  is constructed from  $I_i$ .
  - The parsing actions for state  $i$  are determined as follows:
    - If  $[A \rightarrow \alpha \cdot a\beta]$  is in  $I_i$ , and  $\text{GOTO}(I_i, a) = I_j$ , then set  $\text{ACTION}[i, a]$  to “shift  $j$ .
      - Here  $a$  must be a terminal.
    - If  $[A \rightarrow \alpha \cdot]$  is in  $I_i$ , then set  $\text{ACTION}[i, a]$  to “reduce  $A \rightarrow \alpha$ ” for all  $a$  in  $\text{FOLLOW}(A)$ .
      - Here  $A$  may not be  $S'$ .
    - If  $[S' \rightarrow S \cdot]$  is in  $I_i$ , then set  $\text{ACTION}[i, \$]$  to “accept.”
  - If any conflicting actions result from the above rules, we say the grammar is not SLR(1).
    - The algorithm fails to produce a parser in this case.

- 
- Now consider  $I_1$ ,  $E' \rightarrow E \cdot$ ,  $E \rightarrow E \cdot + T$ .
  - The first item yields  $\text{ACTION}[1, \$] = \text{accept}$ .
  - And the second yields  $\text{ACTION}[1, +] = \text{shift } 6$ .



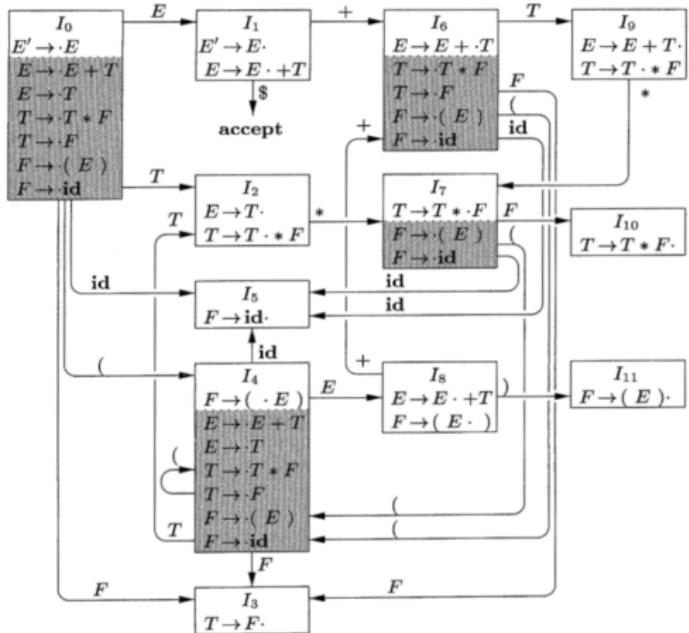
- Now consider  $I_1$ ,  $E' \rightarrow E \cdot$ ,  $E \rightarrow E \cdot + T$ .
  - The first item yields  $\text{ACTION}[1, \$] = \text{accept}$ .
  - And the second yields  $\text{ACTION}[1, +] = \text{shift } 6$ .

STATE	ACTION					GOTO			
	<b>id</b>	<b>+</b>	<b>*</b>	<b>(</b>	<b>)</b>	<b>\$</b>	<b>E</b>	<b>T</b>	<b>F</b>
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		

- Now consider  $I_1$ ,  $E' \rightarrow E \cdot, E \rightarrow E \cdot + T$ .
- The first item yields  $\text{ACTION}[1, \$] = \text{accept}$ .
- And the second yields  $\text{ACTION}[1, +] = \text{shift 6}$ .

2. State  $i$  is constructed from  $I_i$ .
  - The parsing actions for state  $i$  are determined as follows:
    - (a) If  $[A \rightarrow \alpha \cdot a\beta]$  is in  $I_i$ , and  $\text{GOTO}(I_i, a) = I_j$ , then set  $\text{ACTION}[i, a]$  to "shift  $j$ ."
      - Here  $a$  must be a terminal.
    - (b) If  $[A \rightarrow \alpha \cdot \cdot]$  is in  $I_i$ , then set  $\text{ACTION}[i, a]$  to "reduce  $A \rightarrow \alpha$ " for all  $a$  in  $\text{FOLLOW}(A)$ .
      - Here  $A$  may not be  $S'$ .
    - (c) If  $[S' \rightarrow S \cdot \cdot]$  is in  $I_i$ , then set  $\text{ACTION}[i, \$]$  to "accept."
  - If any conflicting actions result from the above rules, we say the grammar is not SLR(1).
    - The algorithm fails to produce a parser in this case.

- 
- Next consider  $I_2$ ,  $E \rightarrow T \cdot$ ,  $T \rightarrow T \cdot * F$ .
  - Since  $\text{FOLLOW}(E) = \{\$\, , +, )\}$ , the first item makes  $\text{ACTION}[2, \$] = \text{ACTION}[2, +] = \text{ACTION}[2, )] = \text{reduce } E \rightarrow T$ .
  - The second item makes  $\text{ACTION}[2, *] = \text{shift } 7$ .



- (1)  $E \rightarrow E + T$
  - (2)  $E \rightarrow T$
  - (3)  $T \rightarrow T * F$
  - (4)  $T \rightarrow F$
  - (5)  $F \rightarrow (E)$
  - (6)  $F \rightarrow \mathbf{id}$

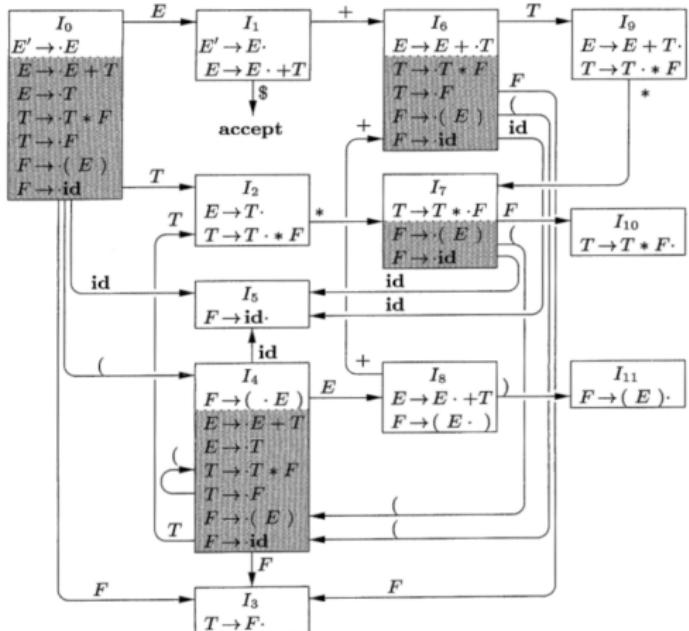
- Next consider  $I_2$ ,  $E \rightarrow T \cdot, T \rightarrow T \cdot * F$ .
  - Since  $\text{FOLLOW}(E) = \{\$, +, )\}$ , the first item makes  $\text{ACTION}[2, \$] = \text{ACTION}[2, +] = \text{ACTION}[2, )] = \text{reduce } E \rightarrow T$ .
  - The second item makes  $\text{ACTION}[2, *] = \text{shift } 7$ .

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		

- Next consider  $I_2$ ,  $E \rightarrow T \cdot$ ,  $T \rightarrow T \cdot * F$ .
- Since  $\text{FOLLOW}(E) = \{\$\text{, } +\text{, } )\}$ , the first item makes  $\text{ACTION}[2, \$] = \text{ACTION}[2, +] = \text{ACTION}[2, )] = \text{reduce } E \rightarrow T$ .
- The second item makes  $\text{ACTION}[2, *] = \text{shift } 7$ .

3. The goto transitions for state  $i$  are constructed for all nonterminals  $A$  using the rule: If  $\text{GOTO}(I_i, A) = I_j$ , then  $\text{GOTO}[i, A] = j$ .
- 

- Now consider the GOTO transitions.
- Since  $\text{GOTO}(I_0, E) = 1$ ,  $\text{GOTO}(I_0, T) = 2$ , and  $\text{GOTO}(I_0, F) = 3$ , we get  $\text{GOTO}(0, E) = 1$ ,  $\text{GOTO}(0, T) = 2$ , and  $\text{GOTO}(0, F) = 3$ .



- (1)  $E \rightarrow E + T$
  - (2)  $E \rightarrow T$
  - (3)  $T \rightarrow T * F$
  - (4)  $T \rightarrow F$
  - (5)  $F \rightarrow (E)$
  - (6)  $F \rightarrow \mathbf{id}$

- Now consider the GOTO transitions.
  - Since  $\text{GOTO}(I_0, E) = 1$ ,  $\text{GOTO}(I_0, T) = 2$ , and  $\text{GOTO}(I_0, F) = 3$ , we get  $\text{GOTO}(0, E) = 1$ ,  $\text{GOTO}(0, T) = 2$ , and  $\text{GOTO}(0, F) = 3$ .

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		

- Now consider the GOTO transitions.
- Since  $\text{GOTO}(I_0, E) = 1$ ,  $\text{GOTO}(I_0, T) = 2$ , and  $\text{GOTO}(I_0, F) = 3$ , we get  $\text{GOTO}(0, E) = 1$ ,  $\text{GOTO}(0, T) = 2$ , and  $\text{GOTO}(0, F) = 3$ .

STATE	ACTION					GOTO		
	<b>id</b>	<b>+</b>	<b>*</b>	<b>(</b>	<b>)</b>	<b>\$</b>	<b>E</b>	<b>T</b>
0	s5			s4			1	2
1		s6				acc		
2		r2	s7			r2	r2	
3		r4	r4			r4	r4	
4	s5			s4			8	2
5		r6	r6			r6	r6	
6	s5			s4				9
7	s5			s4				10
8		s6			s11			
9		r1	s7		r1	r1		
10		r3	r3		r3	r3		
11		r5	r5		r5	r5		

- 
- Continuing in this fashion we obtain the ACTION and GOTO tables

# Example

- Every SLR(1) grammar is unambiguous, but there are many unambiguous grammars that are not SLR(1).
- Consider the grammar with productions,

$$S \rightarrow L = R \mid R$$

$$L \rightarrow *R \mid \mathbf{id}$$

$$R \rightarrow L$$

- Think of  $L$  and  $R$  as standing for  $l$ -value and  $r$ -value, respectively.
- And  $*$  as an operator indicating “contents of.”

## Example — *continued*

- The canonical collection of sets of LR(0) items for the grammar.

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

$I_5: \quad L \rightarrow \text{id} \cdot$   
 $I_6: \quad S \rightarrow L = \cdot R$   
 $R \rightarrow \cdot L$   
 $L \rightarrow \cdot * R$   
 $L \rightarrow \cdot \text{id}$

$I_1: \quad S' \rightarrow S \cdot$

$I_7: \quad L \rightarrow * R \cdot$

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

$I_8: \quad R \rightarrow L \cdot$

$I_3: \quad S \rightarrow R \cdot$

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

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

Figure 4.39: Canonical LR(0) collection for grammar (4.49)

$$\begin{aligned}
 I_0: \quad & S' \rightarrow \cdot S \\
 & S \rightarrow \cdot L = R \\
 & S \rightarrow \cdot R \\
 & L \rightarrow \cdot * R \\
 & L \rightarrow \cdot \mathbf{id} \\
 & R \rightarrow \cdot L
 \end{aligned}$$

$$\begin{aligned}
 I_5: \quad & L \rightarrow \mathbf{id} \cdot \\
 I_6: \quad & S \rightarrow L = \cdot R \\
 & R \rightarrow \cdot L \\
 & L \rightarrow \cdot * R \\
 & L \rightarrow \cdot \mathbf{id}
 \end{aligned}$$

$$I_1: \quad S' \rightarrow S \cdot$$

$$I_7: \quad L \rightarrow *R \cdot$$

$$\begin{aligned}
 I_2: \quad & S \rightarrow L \cdot = R \\
 & R \rightarrow L \cdot
 \end{aligned}$$

$$I_8: \quad R \rightarrow L \cdot$$

$$I_3: \quad S \rightarrow R \cdot$$

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

$$\begin{aligned}
 I_4: \quad & L \rightarrow * \cdot R \\
 & R \rightarrow \cdot L \\
 & L \rightarrow \cdot * R \\
 & L \rightarrow \cdot \mathbf{id}
 \end{aligned}$$

■ Consider the set of items  $I_2$ .

$$\begin{aligned}
 I_0: \quad & S' \rightarrow \cdot S \\
 & S \rightarrow \cdot L = R \\
 & S \rightarrow \cdot R \\
 & L \rightarrow \cdot * R \\
 & L \rightarrow \cdot \mathbf{id} \\
 & R \rightarrow \cdot L
 \end{aligned}$$

$$\begin{aligned}
 I_5: \quad & L \rightarrow \mathbf{id} \cdot \\
 I_6: \quad & S \rightarrow L = \cdot R \\
 & R \rightarrow \cdot L \\
 & L \rightarrow \cdot * R \\
 & L \rightarrow \cdot \mathbf{id}
 \end{aligned}$$

$$I_1: \quad S' \rightarrow S \cdot$$

$$I_7: \quad L \rightarrow *R \cdot$$

$$\begin{aligned}
 I_2: \quad & S \rightarrow L \cdot = R \\
 & R \rightarrow L \cdot
 \end{aligned}$$

$$I_8: \quad R \rightarrow L \cdot$$

$$I_3: \quad S \rightarrow R \cdot$$

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

$$\begin{aligned}
 I_4: \quad & L \rightarrow * \cdot R \\
 & R \rightarrow \cdot L \\
 & L \rightarrow \cdot * R \\
 & L \rightarrow \cdot \mathbf{id}
 \end{aligned}$$

- The first item in this set makes  $\text{ACTION}[2, =]$  be “shift 6.”
- $\text{FOLLOW}(R)$  contains  $= (S \Rightarrow L = R \Rightarrow *R = R)$ .
- The second item sets  $\text{ACTION}[2, =]$  to “reduce  $R \rightarrow L$ .”

$$\begin{aligned}I_0: \quad & S' \rightarrow \cdot S \\& S \rightarrow \cdot L = R \\& S \rightarrow \cdot R \\& L \rightarrow \cdot * R \\& L \rightarrow \cdot \mathbf{id} \\& R \rightarrow \cdot L\end{aligned}$$
$$\begin{aligned}I_5: \quad & L \rightarrow \mathbf{id} \cdot \\I_6: \quad & S \rightarrow L = \cdot R \\& R \rightarrow \cdot L \\& L \rightarrow \cdot * R \\& L \rightarrow \cdot \mathbf{id}\end{aligned}$$
$$I_1: \quad S' \rightarrow S \cdot$$
$$I_7: \quad L \rightarrow * R \cdot$$
$$\begin{aligned}I_2: \quad & S \rightarrow L \cdot = R \\& R \rightarrow L \cdot\end{aligned}$$
$$I_8: \quad R \rightarrow L \cdot$$
$$I_3: \quad S \rightarrow R \cdot$$
$$I_9: \quad S \rightarrow L = R \cdot$$
$$\begin{aligned}I_4: \quad & L \rightarrow * \cdot R \\& R \rightarrow \cdot L \\& L \rightarrow \cdot * R \\& L \rightarrow \cdot \mathbf{id}\end{aligned}$$

- Since there is both a shift and a reduce entry in  $\text{ACTION}[2, =]$ , state 2 has a shift/reduce conflict on input symbol  $=$ .

## Example — *continued*

$S \rightarrow L = R \mid R$

$L \rightarrow *R \mid \mathbf{id}$

$R \rightarrow L$

- The grammar is not ambiguous.
- This shift/reduce conflict arises from the fact that the SLR parser construction method is not powerful enough to remember enough left context to decide what action the parser should take on input  $=$ , having seen a string reducible to  $L$ .

## Example — *continued*

$S \rightarrow L = R \mid R$

$L \rightarrow *R \mid \mathbf{id}$

$R \rightarrow L$

- The canonical and LALR methods, succeeds on a larger collection of grammars, including the above grammar.

## Example — *continued*

$S \rightarrow L = R \mid R$

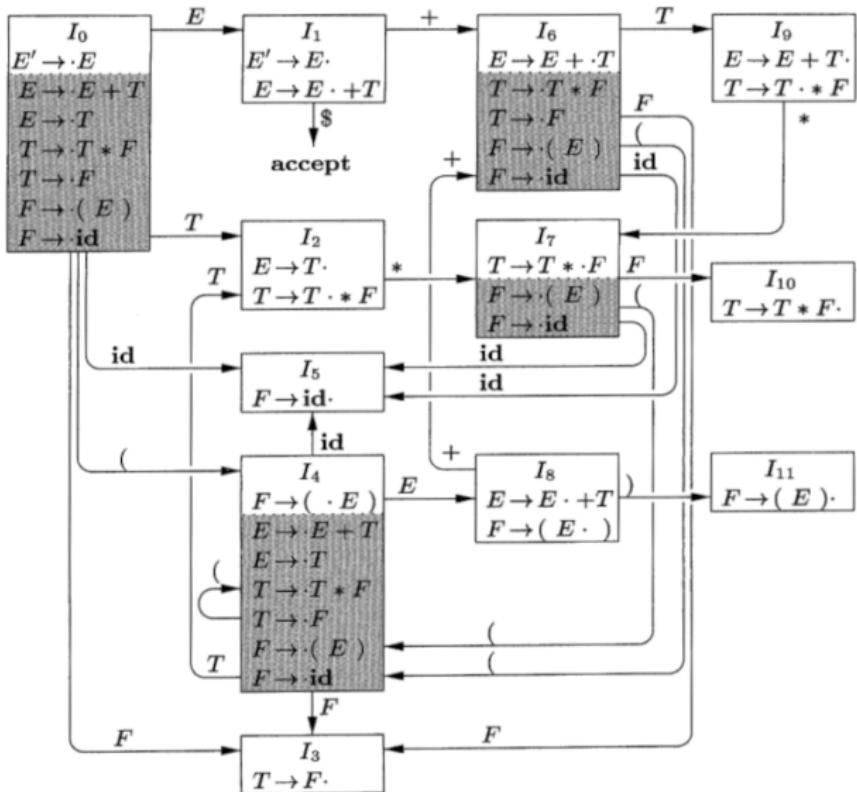
$L \rightarrow *R \mid \mathbf{id}$

$R \rightarrow L$

- Note, however, that there are unambiguous grammars for which every LR parser construction method will produce a parsing action table with parsing action conflicts.
- Fortunately, such grammars can generally be avoided in programming language applications.

- Why can LR(0) automata be used to make shift-reduce decisions?
- The LR(0) automaton for a grammar characterizes the strings of grammar symbols that can appear on the stack of a shift-reduce parser for the grammar.
- The stack contents must be a prefix of a right-sentential form.
- If the stack holds  $\alpha$  and the rest of the input is  $x$ , then a sequence of reductions will take  $\alpha x$  to  $S$ .
- In terms of derivations,  $S \xrightarrow[rm]{*} \alpha x$ .

# Viable Prefixes — *continued*



# Viable Prefixes — *continued*

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		<b>id * id + id \$</b>	shift
(2)	0 5	<b>id</b>	* <b>id + id \$</b>	reduce by $F \rightarrow \text{id}$
(3)	0 3	<b>F</b>	* <b>id + id \$</b>	reduce by $T \rightarrow F$
(4)	0 2	<b>T</b>	* <b>id + id \$</b>	shift
(5)	0 2 7	<b>T *</b>	<b>id + id \$</b>	shift
(6)	0 2 7 5	<b>T * id</b>	+ <b>id \$</b>	reduce by $F \rightarrow \text{id}$
(7)	0 2 7 10	<b>T * F</b>	+ <b>id \$</b>	reduce by $T \rightarrow T * F$
(8)	0 2	<b>T</b>	+ <b>id \$</b>	reduce by $E \rightarrow T$
(9)	0 1	<b>E</b>	+ <b>id \$</b>	shift
(10)	0 1 6	<b>E +</b>	<b>id \$</b>	shift
(11)	0 1 6 5	<b>E + id</b>	\$	reduce by $F \rightarrow \text{id}$
(12)	0 1 6 3	<b>E + F</b>	\$	reduce by $T \rightarrow F$
(13)	0 1 6 9	<b>E + T</b>	\$	reduce by $E \rightarrow E + T$
(14)	0 1	<b>E</b>	\$	accept

Moves of an LR parser on **id \* id + id**

## Viable Prefixes — *continued*

- Not all prefixes of right-sentential forms can appear on the stack, however, since the parser must not shift past the handle.
- For example, suppose,

$$E \xrightarrow[\text{rm}]{*} F * \mathbf{id} \xrightarrow[\text{rm}]{*} (E) * \mathbf{id}$$

- Then, at various times during the parse, the stack will hold  $($ ,  $(E$ , and  $(E)$ , but it must not hold  $(E)^*$ , since  $(E)$  is a handle, which the parser must reduce to  $F$  before shifting  $*$ .

- The prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called *viable prefixes*.
- They are defined as follows:

*a viable prefix is a prefix of a right-sentential form that does not continue past the right end of the right-most handle of that sentential form.*
- By this definition, it is always possible to add terminal symbols to the end of a viable prefix to obtain a right-sentential form.

- SLR parsing is based on the fact that LR(0) automata recognize viable prefixes.
- We say item  $A \rightarrow \beta_1 \cdot \beta_2$  is valid for a viable prefix  $\alpha\beta_1$  if there is a derivation  $S' \xrightarrow[\text{rm}]{}^* \alpha Aw \xrightarrow[\text{rm}]{} \alpha\beta_1\beta_2w$ .
- In general, an item will be valid for many viable prefixes.

- The fact that  $A \rightarrow \beta_1 \cdot \beta_2$  is valid for  $\alpha\beta_1$  tells us a lot about whether to shift or reduce when we find  $\alpha\beta_1$  on the parsing stack.
- In particular, if  $\beta_2 \neq \epsilon$ , then it suggests that we have not yet shifted the handle onto the stack.
- So shift is our move.
- If  $\beta_2 = \epsilon$ , then it looks as if  $A \rightarrow \beta_1$  is the handle.
- And we should reduce by this production.

- Of course, two valid items may tell us to do different things for the same viable prefix.
- Some of these conflicts can be resolved by looking at the next input symbol, and others can be resolved by the other methods.
- But we should not suppose that all parsing action conflicts can be resolved if the LR method is applied to an arbitrary grammar.

## Viable Prefixes — *continued*

- We can easily compute the set of valid items for each viable prefix that can appear on the stack of an LR parser.
- In fact, it is a central theorem of LR-parsing theory that the set of valid items for a viable prefix  $\gamma$  is exactly the set of items reached from the initial state along the path labeled  $\gamma$  in the LR(0) automaton for the grammar.

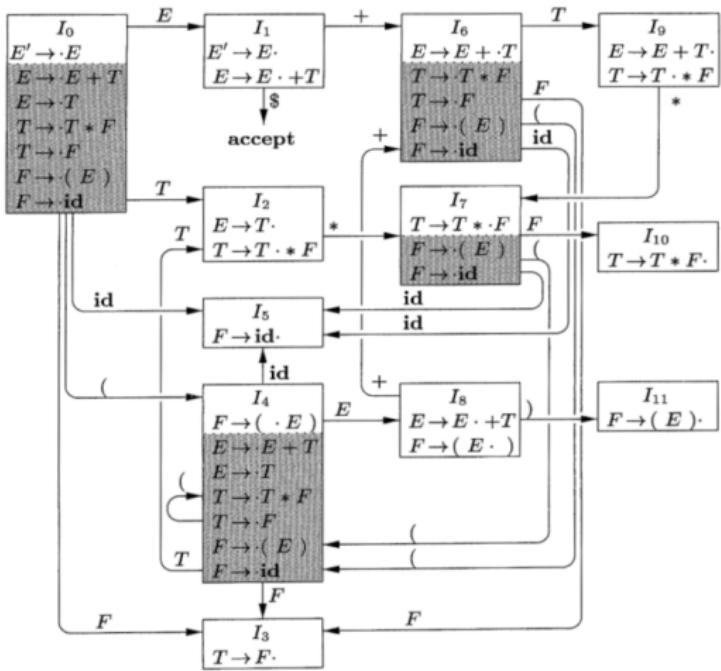
## Viable Prefixes — *continued*

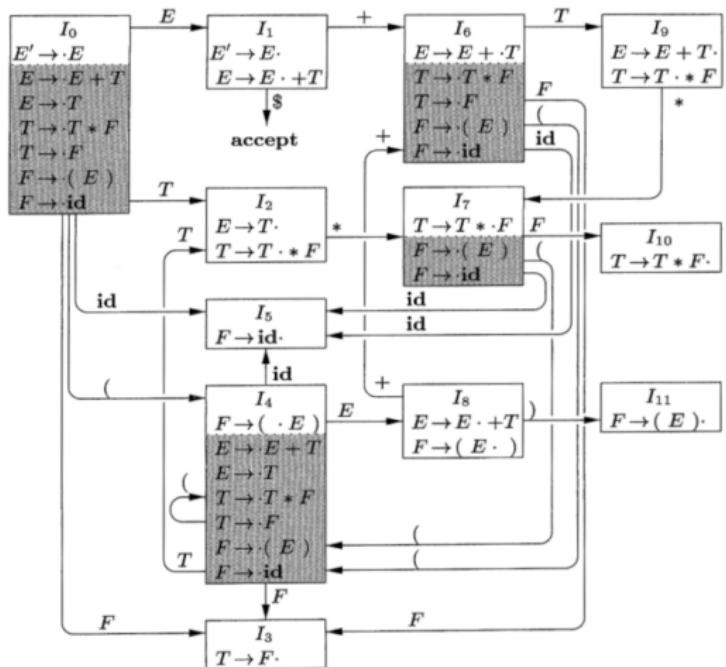
- In essence, the set of valid items embodies all the useful information that can be gleaned from the stack.

# Example

- Let us consider the augmented expression grammar again.
- Whose sets of items and GOTO function are exhibited.

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



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


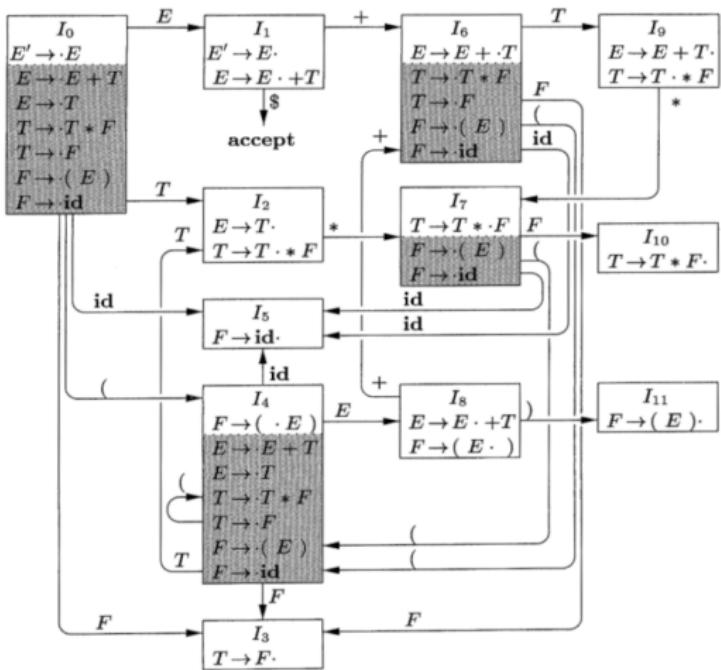
■ Clearly, the string  $E + T*$  is a viable prefix of the grammar.

$$E' \rightarrow E$$

$$E \rightarrow E + T \mid T$$

$T \rightarrow T * F \mid F$

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



- The automaton of will be in state 7 after having read  $E + T^*$ .
  - State 7 contains the items,  $T \rightarrow T * \cdot F$ ,  $F \rightarrow \cdot(E)$ ,  $F \rightarrow \cdot \text{id}$ .
  - Which are precisely the items valid for  $E + T^*$ .

- State 7 contains the items,  $T \rightarrow T * \cdot F$ ,  $F \rightarrow \cdot(E)$ ,  $F \rightarrow \cdot\text{id}$ .
  - Which are precisely the items valid for  $E + T*$ .
- 

$$\begin{array}{c} E' \xrightarrow[\text{rm}]{E' \rightarrow E} E \\ \xrightarrow[\text{rm}]{E \rightarrow E + T} E + T \\ \xrightarrow[\text{rm}]{T \rightarrow T * F} E + T * F \end{array}$$

- We say item  $A \rightarrow \beta_1 \cdot \beta_2$  is valid for a viable prefix  $\alpha\beta_1$  if there is a derivation  
 $S' \xrightarrow[\text{rm}]{*} \alpha A w \xrightarrow[\text{rm}]{*} \alpha\beta_1\beta_2 w$ .

- 
- The first derivation shows the validity of  $T \rightarrow T * \cdot F$ .
  - The second the validity of  $F \rightarrow \cdot(E)$ .
  - And the third the validity of  $F \rightarrow \cdot\text{id}$ .
  - It can be shown that there are no other valid items for  $E + T*$ , although we shall not prove that fact here.

- State 7 contains the items,  $T \rightarrow T * \cdot F$ ,  $F \rightarrow \cdot(E)$ ,  $F \rightarrow \cdot \text{id}$ .
  - Which are precisely the items valid for  $E + T *$ .
- 

$$\begin{array}{c} E' \xrightarrow[\text{rm}]{E' \rightarrow E} E \\ \xrightarrow[\text{rm}]{E \rightarrow E + T} E + T \\ \xrightarrow[\text{rm}]{T \rightarrow T * F} E + T * F \\ \xrightarrow[\text{rm}]{F \rightarrow (E)} E + T * (E) \end{array}$$

- We say item  $A \rightarrow \beta_1 \cdot \beta_2$  is valid for a viable prefix  $\alpha \beta_1$  if there is a derivation

$$S' \xrightarrow[\text{rm}]{*} \alpha A w \xrightarrow[\text{rm}]{*} \alpha \beta_1 \beta_2 w.$$

- 
- The first derivation shows the validity of  $T \rightarrow T * \cdot F$ .
  - The second the validity of  $F \rightarrow \cdot(E)$ .
  - And the third the validity of  $F \rightarrow \cdot \text{id}$ .
  - It can be shown that there are no other valid items for  $E + T *$ , although we shall not prove that fact here.

- State 7 contains the items,  $T \rightarrow T * \cdot F$ ,  $F \rightarrow \cdot(E)$ ,  $F \rightarrow \cdot \text{id}$ .
  - Which are precisely the items valid for  $E + T *$ .
- 

$$\begin{array}{c} E' \xrightarrow[\text{rm}]{E' \rightarrow E} E \\ \xrightarrow[\text{rm}]{E \rightarrow E + T} E + T \\ \xrightarrow[\text{rm}]{T \rightarrow T * F} E + T * F \\ \xrightarrow[\text{rm}]{F \rightarrow \text{id}} E + T * \text{id} \end{array}$$

- We say item  $A \rightarrow \beta_1 \cdot \beta_2$  is valid for a viable prefix  $\alpha \beta_1$  if there is a derivation
- $$S' \xrightarrow[\text{rm}]{*} \alpha A w \xrightarrow[\text{rm}]{*} \alpha \beta_1 \beta_2 w.$$

- 
- The first derivation shows the validity of  $T \rightarrow T * \cdot F$ .
  - The second the validity of  $F \rightarrow \cdot(E)$ .
  - And the third the validity of  $F \rightarrow \cdot \text{id}$ .
  - It can be shown that there are no other valid items for  $E + T *$ , although we shall not prove that fact here.

- In this section, we shall extend the previous LR parsing techniques to use one symbol of lookahead on the input.
- There are two different methods:
  1. The “canonical-LR” or just “LR” method, which makes full use of the lookahead symbol(s).
  - This method uses a large set of items, called the LR(1) items.

# More Powerful LR Parsers

- In this section, we shall extend the previous LR parsing techniques to use one symbol of lookahead on the input.
  - There are two different methods:
2. The “lookahead-LR” or “LALR” method, which is based on the LR(0) sets of items, and has many fewer states than typical parsers based on the LR(1) items.
- By carefully introducing lookaheads into the LR(0) items, we can handle many more grammars with the LALR method than with the SLR method
  - We can build parsing tables that are no bigger than the SLR tables.

- In this section, we shall extend the previous LR parsing techniques to use one symbol of lookahead on the input.
- There are two different methods:
- LALR is the method of choice in most situations.

# Canonical LR(1) Items

- We shall now present the most general technique for constructing an LR parsing table from a grammar.
- Recall that in the SLR method, state  $i$  calls for reduction by  $A \rightarrow \alpha$  if the set of items  $I_i$  contains item  $[A \rightarrow \alpha \cdot]$  and  $a$  is in  $\text{FOLLOW}(A)$ .
- In some situations, however, when state  $i$  appears on top of the stack, the viable prefix  $\beta\alpha$  on the stack is such that  $\beta A$  cannot be followed by  $a$  in any right-sentential form.
- Thus, the reduction by  $A \rightarrow \alpha$  should be invalid on input  $a$ .

# Example

- Let us reconsider the grammar with productions,

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

- $L$  and  $R$  are standing for  $\ell$ -value and  $r$ -value, respectively.
- $*$  is an operator indicating “contents of.”

$$\begin{aligned}
 I_0: \quad & S' \rightarrow \cdot S \\
 & S \rightarrow \cdot L = R \\
 & S \rightarrow \cdot R \\
 & L \rightarrow \cdot * R \\
 & L \rightarrow \cdot \mathbf{id} \\
 & R \rightarrow \cdot L
 \end{aligned}$$

$$\begin{aligned}
 I_5: \quad & L \rightarrow \mathbf{id} \cdot \\
 I_6: \quad & S \rightarrow L = \cdot R \\
 & R \rightarrow \cdot L \\
 & L \rightarrow \cdot * R \\
 & L \rightarrow \cdot \mathbf{id}
 \end{aligned}$$

$$\begin{aligned}
 I_1: \quad & S' \rightarrow S \cdot \\
 I_2: \quad & S \rightarrow L \cdot = R \\
 & R \rightarrow L \cdot
 \end{aligned}$$

$$\begin{aligned}
 I_7: \quad & L \rightarrow * R \cdot \\
 I_8: \quad & R \rightarrow L \cdot
 \end{aligned}$$

$$\begin{aligned}
 I_3: \quad & S \rightarrow R \cdot \\
 I_4: \quad & L \rightarrow * \cdot R \\
 & R \rightarrow \cdot L \\
 & L \rightarrow \cdot * R \\
 & L \rightarrow \cdot \mathbf{id}
 \end{aligned}$$

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

- In some situations, however, when state  $i$  appears on top of the stack, the viable prefix  $\beta \alpha$  on the stack is such that  $\beta A$  cannot be followed by  $a$  in any right-sentential form.
- Thus, the reduction by  $A \rightarrow \alpha$  should be invalid on input  $a$ .

- In state 2 we have item  $R \rightarrow L \cdot \cdot$ .
- This can correspond to  $A \rightarrow \alpha$  above.
- $a$  could be the  $=$  sign, which is in  $\text{FOLLOW}(R)$ .

$$\begin{aligned}
 I_0: \quad & S' \rightarrow \cdot S \\
 & S \rightarrow \cdot L = R \\
 & S \rightarrow \cdot R \\
 & L \rightarrow \cdot * R \\
 & L \rightarrow \cdot \mathbf{id} \\
 & R \rightarrow \cdot L
 \end{aligned}$$

$$\begin{aligned}
 I_5: \quad & L \rightarrow \mathbf{id} \cdot \\
 I_6: \quad & S \rightarrow L = \cdot R \\
 & R \rightarrow \cdot L \\
 & L \rightarrow \cdot * R \\
 & L \rightarrow \cdot \mathbf{id}
 \end{aligned}$$

$$I_1: \quad S' \rightarrow S \cdot$$

$$I_7: \quad L \rightarrow * R \cdot$$

$$\begin{aligned}
 I_2: \quad & S \rightarrow L \cdot = R \\
 & R \rightarrow L \cdot
 \end{aligned}$$

$$I_8: \quad R \rightarrow L \cdot$$

$$I_3: \quad S \rightarrow R \cdot$$

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

$$\begin{aligned}
 I_4: \quad & L \rightarrow * \cdot R \\
 & R \rightarrow \cdot L \\
 & L \rightarrow \cdot * R \\
 & L \rightarrow \cdot \mathbf{id}
 \end{aligned}$$

- In some situations, however, when state  $i$  appears on top of the stack, the viable prefix  $\beta \alpha$  on the stack is such that  $\beta A$  cannot be followed by  $a$  in any right-sentential form.
- Thus, the reduction by  $A \rightarrow \alpha$  should be invalid on input  $a$ .

- Thus, the SLR parser calls for reduction by  $R \rightarrow L$  in state 2 with  $=$  as the next input.

$$\begin{aligned}
 I_0: \quad & S' \rightarrow \cdot S \\
 & S \rightarrow \cdot L = R \\
 & S \rightarrow \cdot R \\
 & L \rightarrow \cdot * R \\
 & L \rightarrow \cdot \mathbf{id} \\
 & R \rightarrow \cdot L
 \end{aligned}$$

$$\begin{aligned}
 I_5: \quad & L \rightarrow \mathbf{id} \cdot \\
 I_6: \quad & S \rightarrow L = \cdot R \\
 & R \rightarrow \cdot L \\
 & L \rightarrow \cdot * R \\
 & L \rightarrow \cdot \mathbf{id}
 \end{aligned}$$

$$I_1: \quad S' \rightarrow S \cdot$$

$$I_7: \quad L \rightarrow * R \cdot$$

$$\begin{aligned}
 I_2: \quad & S \rightarrow L \cdot = R \\
 & R \rightarrow L \cdot
 \end{aligned}$$

$$I_8: \quad R \rightarrow L \cdot$$

$$I_3: \quad S \rightarrow R \cdot$$

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

$$\begin{aligned}
 I_4: \quad & L \rightarrow * \cdot R \\
 & R \rightarrow \cdot L \\
 & L \rightarrow \cdot * R \\
 & L \rightarrow \cdot \mathbf{id}
 \end{aligned}$$

- In some situations, however, when state  $i$  appears on top of the stack, the viable prefix  $\beta \alpha$  on the stack is such that  $\beta A$  cannot be followed by  $a$  in any right-sentential form.
- Thus, the reduction by  $A \rightarrow \alpha$  should be invalid on input  $a$ .

- The shift action is also called for, because of item  $S \rightarrow L \cdot = R$  in state 2.

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

$I_5: \quad L \rightarrow \mathbf{id} \cdot$   
 $I_6: \quad S \rightarrow L = \cdot R$   
 $R \rightarrow \cdot L$   
 $L \rightarrow \cdot * R$   
 $L \rightarrow \cdot \mathbf{id}$

$I_1: \quad S' \rightarrow S \cdot$

$I_7: \quad L \rightarrow * R \cdot$

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

$I_8: \quad R \rightarrow L \cdot$

$I_3: \quad S \rightarrow R \cdot$

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

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

- In some situations, however, when state  $i$  appears on top of the stack, the viable prefix  $\beta \alpha$  on the stack is such that  $\beta A$  cannot be followed by  $a$  in any right-sentential form.
- Thus, the reduction by  $A \rightarrow \alpha$  should be invalid on input  $a$ .

- However, there is no right-sentential form of this grammar that begins  $R = \dots$ .

$$\begin{aligned}
 I_0: \quad & S' \rightarrow \cdot S \\
 & S \rightarrow \cdot L = R \\
 & S \rightarrow \cdot R \\
 & L \rightarrow \cdot * R \\
 & L \rightarrow \cdot \mathbf{id} \\
 & R \rightarrow \cdot L
 \end{aligned}$$

$$I_1: \quad S' \rightarrow S \cdot$$

$$\begin{aligned}
 I_2: \quad & S \rightarrow L \cdot = R \\
 & R \rightarrow L \cdot
 \end{aligned}$$

$$I_3: \quad S \rightarrow R \cdot$$

$$\begin{aligned}
 I_4: \quad & L \rightarrow * \cdot R \\
 & R \rightarrow \cdot L \\
 & L \rightarrow \cdot * R \\
 & L \rightarrow \cdot \mathbf{id}
 \end{aligned}$$

$$\begin{aligned}
 I_5: \quad & L \rightarrow \mathbf{id} \cdot \\
 I_6: \quad & S \rightarrow L = \cdot R \\
 & R \rightarrow \cdot L \\
 & L \rightarrow \cdot * R \\
 & L \rightarrow \cdot \mathbf{id}
 \end{aligned}$$

$$I_7: \quad L \rightarrow * R \cdot$$

$$I_8: \quad R \rightarrow L \cdot$$

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

- In some situations, however, when state  $i$  appears on top of the stack, the viable prefix  $\beta \alpha$  on the stack is such that  $\beta A$  cannot be followed by  $a$  in any right-sentential form.
- Thus, the reduction by  $A \rightarrow \alpha$  should be invalid on input  $a$ .

- Thus state 2, which is the state corresponding to viable prefix  $L$  only, should *not* really call for reduction of that  $L$  to  $R$ .

# Elaboration on the Previous Example

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf>

**id = id**

$$I_2: \begin{array}{l} S \rightarrow L \cdot = R \\ R \rightarrow L \cdot \end{array}$$

- Consider parsing the expression **id = id**.
- After working our way to configurating set  $I_2$  having reduced the first **id** to  $L$ , we have a choice upon seeing  $=$  coming up in the input.

# Elaboration on the Previous Example

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf>

**id = id**

$$I_2: \begin{array}{l} S \rightarrow L \cdot = R \\ R \rightarrow L \cdot \end{array}$$

- The first item in the set wants to set Action[2, =] be shift 6.
- Which corresponds to moving on to find the rest of the assignment.
- However, = is also in FOLLOW( $R$ ) because  $S \Rightarrow L = R \Rightarrow *R = R$ .
- Thus, the second configuration wants to reduce in that slot  $R \rightarrow L$ .

# Elaboration on the Previous Example

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf>

**id = id**

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

- This is a shift/reduce conflict but not because of any problem with the grammar.
- A SLR parser does not remember enough left context to decide what should happen when it encounters a  $=$  in the input having seen a string reducible to  $L$ .

# Elaboration on the Previous Example

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf>

**id = id**

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

- Although the sequence on top of the stack could be reduced to  $R$ .
- We don't want to choose this reduction because there is no possible right sentential form that begins  $R = \dots$ .
- There is one beginning  $*R = \dots$  which is not the same.

# Elaboration on the Previous Example

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf>

**id = id**

$$I_2: \begin{array}{l} S \rightarrow L \cdot = R \\ R \rightarrow L \cdot \end{array}$$

- Thus, the correct choice is to shift.
- It's not further lookahead that the SLR tables are missing—we don't need to see additional symbols beyond the first token in the input.
- We have already seen the information that allows us to determine the correct choice.

# Elaboration on the Previous Example

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf>

**id = id**

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

- What we need is to retain a little more of the left context that brought us here.
- In this example grammar, the only time we should consider reducing by production  $R \rightarrow L$  is during a derivation that has already seen a \* or an =.

# Elaboration on the Previous Example

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf>

**id = id**

$$I_2: \begin{array}{l} S \rightarrow L \cdot = R \\ R \rightarrow L \cdot \end{array}$$

- Just using the entire FOLLOW set is not discriminating enough as the guide for when to reduce.
- The FOLLOW set contains symbols that can follow  $R$  in any position within a valid sentence.
- But it does not precisely indicate which symbols follow  $R$  at this particular point in a derivation.

# Elaboration on the Previous Example

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf>

**id = id**

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

- So we will augment our states to include information about what portion of the FOLLOW set is appropriate given the path we have taken to that state.

# Elaboration on the Previous Example

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf>

**id = id**

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

- We can be in state 2 for one of two reasons, we are trying to build from  $S \rightarrow L = R$  or from  $S \rightarrow R \rightarrow L$ .
- If the upcoming symbol is  $=$ , then that rules out the second choice and we must be building the first, which tells us to shift.

# Elaboration on the Previous Example

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf>

**id = id**

$$I_2: \begin{array}{l} S \rightarrow L \cdot = R \\ R \rightarrow L \cdot \end{array}$$

- The reduction should only be applied if the next input symbol is \$.
- Even though = is FOLLOW( $R$ ) because of the other contexts that an  $R$  can appear, in this particular situation, it is not appropriate because when deriving a sentence  $S \rightarrow R \rightarrow L, =$  cannot follow  $R$ .

## Canonical LR(1) Items — *continued*

- It is possible to carry more information in the state that will allow us to rule out some of these invalid reductions by  $A \rightarrow \alpha$ .
- By splitting states when necessary, we can arrange to have each state of an LR parser indicate exactly which input symbols can follow a handle  $\alpha$  for which there is a possible reduction to  $A$ .

## Canonical LR(1) Items — *continued*

- It is possible to carry more information in the state that will **allow us to rule out some of these invalid reductions** by  $A \rightarrow \alpha$ .
- By splitting states when necessary, we can arrange to have each state of an LR parser indicate exactly which input symbols can follow a handle  $\alpha$  for which there is a possible reduction to  $A$ .

# A Bit of Elaboration on the Issue of Canonical LR(1) Items

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf>

- LR or canonical LR parsing incorporates the required extra information into the state by redefining configurations to include a terminal symbol as an added component.
- LR(1) configurations have the general form:

$$A \rightarrow X_1 \dots X_i \cdot X_{i+1} \dots X_j, a$$

## A Bit of Elaboration on the Issue — *continued*

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf>

$A \rightarrow X_1 \dots X_i \cdot X_{i+1} \dots X_j, a$

- This means we have states corresponding to  $X_1 \dots X_i$  on the stack.
- And we are looking to put states corresponding to  $X_{i+1} \dots X_j$  on the stack and then reduce.
- But only if the token following  $X_j$  is the terminal  $a$ .

## A Bit of Elaboration on the Issue — *continued*

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf>

$$A \rightarrow X_1 \dots X_i \cdot X_{i+1} \dots X_j, a$$

- $a$  is called the lookahead of the configuration.
- The lookahead only comes into play with LR(1) configurations with a dot at the right end:

$$A \rightarrow X_1 \dots X_j \cdot, a$$

- This means we have states corresponding to  $X_1 \dots X_j$  on the stack but we may only reduce when the next symbol is  $a$ .

## A Bit of Elaboration on the Issue — *continued*

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf>

$$A \rightarrow X_1 \dots X_i \cdot X_{i+1} \dots X_j, a$$

- The symbol  $a$  is either a terminal or  $\$$  (end of input marker).

## A Bit of Elaboration on the Issue — *continued*

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf>

$$A \rightarrow X_1 \dots X_i \cdot X_{i+1} \dots X_j, a$$

- With SLR(1) parsing, we would reduce if the next token was any of those in  $\text{FOLLOW}(A)$ .
- With LR(1) parsing, we reduce only if the next token is exactly  $a$ .

## A Bit of Elaboration on the Issue — *continued*

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf>

$A \rightarrow X_1 \dots X_i \cdot X_{i+1} \dots X_j, a$

- We may have more than one symbol in the lookahead for the configuration.
- As a convenience, we list those symbols separated by a forward slash.
- Thus, the configuration,  
 $A \rightarrow \beta_1 \cdot, a/b/c$  says that it is valid to reduce  $\beta_1$  to  $A$  only if the next token is equal to  $a$ ,  $b$ , or  $c$ .

## A Bit of Elaboration on the Issue — *continued*

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf>

$$A \rightarrow X_1 \dots X_i \cdot X_{i+1} \dots X_j, a$$

- The configuration lookahead will always be a subset (or a proper subset) of  $\text{FOLLOW}(A)$ .

## A Bit of Elaboration on the Issue — *continued*

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf>

$$A \rightarrow X_1 \dots X_i \cdot X_{i+1} \dots X_j, a$$

- Recall the definition of a viable prefix.
- Viable prefixes are those prefixes of right sentential forms that can appear on the stack of a shift-reduce parser.

## A Bit of Elaboration on the Issue — *continued*

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf>

$$A \rightarrow X_1 \dots X_i \cdot X_{i+1} \dots X_j, a$$

- We say item  $A \rightarrow \beta_1 \cdot \beta_2$  is valid for a viable prefix  $\alpha\beta_1$  if there is a derivation  $S' \xrightarrow[\text{rm}]{*} \alpha Aw \xrightarrow[\text{rm}]{*} \alpha\beta_1\beta_2w$ .
- Formally we say that a configuration  $[A \rightarrow \beta_1 \cdot \beta_2, a]$  is valid for a viable prefix  $\alpha\beta_1$  if there is a derivation  $S' \xrightarrow[\text{rm}]{*} \alpha Aw \xrightarrow[\text{rm}]{*} \alpha\beta_1\beta_2w$ .
- Where either  $a$  is the first symbol of  $w$  or  $w$  is  $\epsilon$  and  $a$  is  $\$$ .

# A Bit of Elaboration on the Issue — Example

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf>

- A configuration  $[A \rightarrow \beta_1 \cdot \beta_2, a]$  is valid for a viable prefix  $\alpha\beta_1$  if there is a derivation  $S' \xrightarrow[\text{rm}]^* \alpha Aw \xrightarrow[\text{rm}]^* \alpha\beta_1\beta_2w$ .
- Where either  $a$  is the first symbol of  $w$  or  $w$  is  $\epsilon$  and  $a$  is  $\$$ .

$$S' \rightarrow ZZ$$

$$Z \rightarrow xZ \mid y$$

- There is a rightmost derivation,  
 $S' \xrightarrow[*] xxZxy \Rightarrow xxxZxy$ .
- We see that configuration  $[Z \rightarrow x \cdot Z, x]$  is valid for viable prefix  $\alpha = xx$  by letting  $A = Z$ ,  $w = xy$ ,  $\beta_1 = x$  and  $\beta_2 = Z$ .

# A Bit of Elaboration on the Issue — Example

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf>

- A configuration  $[A \rightarrow \beta_1 \cdot \beta_2, a]$  is valid for a viable prefix  $\alpha\beta_1$  if there is a derivation  $S' \xrightarrow[\text{rm}]^* \alpha Aw \xrightarrow[\text{rm}]^* \alpha\beta_1\beta_2w$ .
- Where either  $a$  is the first symbol of  $w$  or  $w$  is  $\epsilon$  and  $a$  is  $\$$ .

$$S' \rightarrow ZZ$$

$$Z \rightarrow xZ \mid y$$

- Another example is from the rightmost derivation  $S' \xrightarrow[*] ZxZ \Rightarrow ZxxZ$ .
- Making  $[Z \rightarrow x \cdot Z, \$]$  valid for viable prefix  $Zxx$ .

## Canonical LR(1) Items — *continued*

- The extra information is incorporated into the state by redefining items to include a terminal symbol as a second component.
- The general form of an item becomes  $[A \rightarrow \alpha \cdot \beta, a]$ , where  $A \rightarrow \alpha\beta$  is a production and  $a$  is a terminal or the right endmarker  $\$$ .
- We call such an object an LR(1) item.
- The 1 refers to the length of the second component, called the lookahead of the item.

- The lookahead has no effect in an item of the form  $[A \rightarrow \alpha \cdot \beta, a]$ , where  $\beta$  is *not*  $\epsilon$ .
- But an item of the form  $[A \rightarrow \alpha \cdot , a]$  calls for a reduction by  $A \rightarrow \alpha$  only if the next input symbol is  $a$ .
- Thus, we are compelled to reduce by  $A \rightarrow \alpha$  only on those input symbols  $a$  for which  $[A \rightarrow \alpha \cdot , a]$  is an LR(1) item in the state on top of the stack.
- The set of such  $a$ 's will always be a subset of  $\text{FOLLOW}(A)$ , but it could be a proper subset.

## Canonical LR(1) Items — *continued*

- Formally, we say LR(1) item  $[A \rightarrow \alpha \cdot \beta, a]$  is *valid* for a viable prefix  $\gamma$  if there is a derivation  $S \xrightarrow[\text{rm}]^* \delta Aw \xrightarrow[\text{rm}]^* \delta \alpha \beta w$ , where,
  - $\gamma = \delta \alpha$ , and
  - Either  $a$  is the first symbol of  $w$ , or  $w$  is  $\epsilon$  and  $a$  is  $\$$ .

# Example

- Let us consider the grammar

$$S \rightarrow B B$$

$$B \rightarrow a B \mid b$$

- There is a rightmost derivation  $S \xrightarrow[\text{rm}]{*} aaBab \xrightarrow[\text{rm}]{*} aaaBab$ .
- We see that item  $[B \rightarrow a \cdot B, a]$  is valid for a viable prefix  $\gamma = aaa$  by letting  $\delta = aa$ ,  $A = B$ ,  $w = ab$ ,  $\alpha = a$ , and  $\beta = B$  in the above definition.
- There is also a rightmost derivation  $S \xrightarrow[\text{rm}]{*} BaB \xrightarrow[\text{rm}]{*} BaaB$ .
- From this derivation we see that item  $[B \rightarrow a \cdot B, \$]$  is valid for viable prefix  $Baa$ .

# Constructing LR(1) Sets of Items

- The method for building the collection of sets of valid LR(1) items is essentially the same as the one for building the canonical collection of sets of LR(0) items.
- We need only to modify the two procedures CLOSURE and GOTO.

# Constructing LR(1) Sets of Items — *continued*

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

SetOfItems GOTO( $I, X$ ) {
    initialize  $J$  to be the empty set;
    for ( each item  $[A \rightarrow \alpha \cdot X\beta, a]$  in  $I$  )
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;
    return CLOSURE( $J$ );
}

void items( $G'$ ) {
    initialize  $C$  to CLOSURE( $\{[S' \rightarrow \cdot S, \$]\}$ );
    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  $C$ ;
}
```

Figure 4.40: Sets-of-LR(1)-items construction for grammar  $G'$

## Constructing LR(1) Sets of Items — *continued*

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

## Constructing LR(1) Sets of Items — *continued*

```
SetOfItems GOTO( $I, X$ ) {
    initialize  $J$  to be the empty set;
    for ( each item  $[A \rightarrow \alpha \cdot X \beta, a]$  in  $I$  )
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;
    return CLOSURE( $J$ );
}
```

# Constructing LR(1) Sets of Items — *continued*

```
void items( $G'$ ) {
    initialize  $C$  to CLOSURE( $\{[S' \rightarrow \cdot S, \$]\}$ );
    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  $C$ ;
}
```

## Constructing LR(1) Sets of Items — *continued*

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

- Let's appreciate the new definition of the CLOSURE operation.
- In particular, why  $b$  must be in  $\text{FIRST}(\beta a)$ ?
- Consider an item of the form  $[A \rightarrow \alpha \cdot B\beta, a]$  in the set of items valid for some viable prefix  $\gamma$ .

# Constructing LR(1) Sets of Items — *continued*

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

- Then there is a rightmost derivation,

$$S \xrightarrow[\text{rm}]{*} \delta A a x \xrightarrow[\text{rm}]{*} \delta \alpha B \beta a x,$$

where,  $\gamma = \delta \alpha$ .

# Constructing LR(1) Sets of Items — *continued*

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

- Suppose  $\beta ax$  derives terminal string  $by$ .
- Then for each production of the form  $B \rightarrow \eta$  for some  $\eta$ , we have derivation  $S \xrightarrow[\text{rm}]{*} \gamma B by \xrightarrow[\text{rm}]{*} \gamma \eta by$ .
- Thus,  $[B \rightarrow \cdot \eta, b]$  is valid for  $\gamma$ .
- (A configuration  $[A \rightarrow \beta_1 \cdot \beta_2, a]$  is valid for a viable prefix  $\alpha \beta_1$  if there is a derivation  $S' \xrightarrow[\text{rm}]{*} \alpha Aw \xrightarrow[\text{rm}]{*} \alpha \beta_1 \beta_2 w$ , where either  $a$  is the first symbol of  $w$  or  $w$  is  $\epsilon$  and  $a$  is  $\$$ .)

## Constructing LR(1) Sets of Items — *continued*

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

- Note that  $b$  can be the first terminal derived from  $\beta$ , or it is possible that  $\beta$  derives  $\epsilon$  in the derivation  $\beta ax \xrightarrow[\text{rm}]{*} by$ .
- And  $b$  can therefore be  $a$ .

## Constructing LR(1) Sets of Items — *continued*

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

- To summarize both possibilities we say that  $b$  can be any terminal in  $\text{FIRST}(\beta ax)$ .
- Note that  $x$  cannot contain the first terminal of  $by$ .
- So  $\text{FIRST}(\beta ax) = \text{FIRST}(\beta a)$ .

# Example

- Consider the following augmented grammar,

$$S' \rightarrow S$$

$$S \rightarrow C C$$

$$C \rightarrow c C \mid d$$

$S' \rightarrow S, S \rightarrow C\ C, C \rightarrow c\ C \mid d$

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

$I_0$
$S' \rightarrow \cdot S, \$$
$S \rightarrow \cdot C\ C, \$$
$C \rightarrow \cdot c\ C, c/d$
$C \rightarrow \cdot d, c/d$

- We begin by computing the closure of  $\{[S' \rightarrow \cdot S, \$]\}$ .
- To close, we match the item  $[S \rightarrow \cdot S, \$]$  with the item  $[A \rightarrow \alpha \cdot B\beta, a]$  in the procedure CLOSURE.
- That is,  $A = S'$ ,  $\alpha = \epsilon$ ,  $B = S$ ,  $\beta = \epsilon$ , and  $a = \$$ .

$$S' \rightarrow S, S \rightarrow C C, C \rightarrow c C \mid d$$

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

$I_0$
$S' \rightarrow \cdot S, \$$
$S \rightarrow \cdot C C, \$$
$C \rightarrow \cdot c C, c/d$
$C \rightarrow \cdot d, c/d$

- Function CLOSURE tells us to add  $[B \rightarrow \cdot \gamma, b]$  for each production  $B \rightarrow \gamma$  and terminal  $b$  in  $\text{FIRST}(\beta a)$ .
- In terms of the present grammar,  $B \rightarrow \gamma$  must be  $S \rightarrow C C$
- And since  $\beta$  is  $\epsilon$  and  $a$  is  $\$$ ,  $b$  may only be  $\$$ .
- Thus we add  $[S \rightarrow \cdot C C, \$]$ .

$$S' \rightarrow S, S \rightarrow C C, C \rightarrow c C \mid d$$

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

$I_0$
$S' \rightarrow \cdot S, \$$
$S \rightarrow \cdot C C, \$$
$C \rightarrow \cdot c C, c/d$
$C \rightarrow \cdot d, c/d$

- We continue to compute the closure by adding all items  $[C \cdot \gamma, b]$  for  $b$  in  $\text{FIRST}(C\$)$ .
- That is, matching  $[S \rightarrow \cdot C C, \$]$  against  $[A \rightarrow \alpha \cdot B\beta, a]$ , we have  $A = S$ ,  $\alpha = \epsilon$ ,  $B = C$ ,  $\beta = C$ , and  $a = \$$ .

$$S' \rightarrow S, S \rightarrow C C, C \rightarrow c C \mid d$$

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

$I_0$
$S' \rightarrow \cdot S, \$$
$S \rightarrow \cdot C C, \$$
$C \rightarrow \cdot c C, c/d$
$C \rightarrow \cdot d, c/d$

- Since  $C$  does not derive the empty string,  
 $\text{FIRST}(C\$) = \text{FIRST}(C)$ .
- Since  $\text{FIRST}(C)$  contains terminals  $c$  and  $d$ , we add items  
 $[C \rightarrow \cdot c C, c]$ ,  $[C \rightarrow \cdot c C, d]$ ,  $[C \rightarrow \cdot d, c]$  and  $[C \rightarrow \cdot d, d]$ .

$$S' \rightarrow S, S \rightarrow C C, C \rightarrow c C \mid d$$

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

$I_0$
$S' \rightarrow \cdot S, \$$
$S \rightarrow \cdot C C, \$$
$C \rightarrow \cdot c C, c/d$
$C \rightarrow \cdot d, c/d$

- None of the new items has a nonterminal immediately to the right of the dot, so we have completed our first set of LR(1) items.

$S' \rightarrow S, S \rightarrow C\ C, C \rightarrow c\ C \mid d$

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

$I_0$   
 $S' \rightarrow \cdot S, \$$   
 $S \rightarrow \cdot CC, \$$   
 $C \rightarrow \cdot cC, c/d$   
 $C \rightarrow \cdot d, c/d$

- The initial set of items is complete.
- The brackets have been omitted for notational convenience, and we use the notation  $[C \rightarrow \cdot cC, c/d]$  as a shorthand for the two items  $[C \rightarrow \cdot cC, c]$  and  $[C \rightarrow \cdot cC, d]$ .

$$S' \rightarrow S, S \rightarrow C\ C, C \rightarrow c\ C \mid d$$

```
SetOfItems GOTO( $I, X$ ) {  
    initialize  $J$  to be the empty set;  
    for ( each item  $[A \rightarrow \alpha \cdot X \beta, a]$  in  $I$  )  
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;  
    return CLOSURE( $J$ );  
}
```

$\text{GOTO}(I_0, S)$

$I_0$

$S' \rightarrow \cdot S, \$$   
 $S \rightarrow \cdot C C, \$$   
 $C \rightarrow \cdot c C, c/d$   
 $C \rightarrow \cdot d, c/d$

$I_1$

$S' \rightarrow S \cdot, \$$

- Now we compute  $\text{GOTO}(I_0, X)$  for the various values of  $X$ .
- For  $X = S$  we must close the item  $[S' \rightarrow S \cdot, \$]$ .
- No additional closure is possible, since the dot is at the right end.
- Thus we have the next set of items,  $I_1$ .

$$S' \rightarrow S, S \rightarrow C \ C, C \rightarrow c \ C \mid d$$

```
SetOfItems GOTO( $I, X$ ) {
    initialize  $J$  to be the empty set;
    for ( each item  $[A \rightarrow \alpha \cdot X \beta, a]$  in  $I$  )
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;
    return CLOSURE( $J$ );
}
```

$$\text{GOTO}(I_0, C)$$

$I_0$
$S' \rightarrow \cdot S, \$$
$S \rightarrow \cdot CC, \$$
$C \rightarrow \cdot cC, c/d$
$C \rightarrow \cdot d, c/d$

$I_2$
$S \rightarrow C \cdot C, \$$
$C \rightarrow \cdot cC, \$$
$C \rightarrow \cdot d, \$$

- For  $X = C$  we close  $[S \rightarrow C \cdot C, \$]$ .
- We add the  $C$ -productions with second component  $\$$ .
- And then can add no more, yielding  $I_2$ .

$$S' \rightarrow S, S \rightarrow C \ C, C \rightarrow c \ C \mid d$$
$$\text{GOTO}(I_0, c)$$

```
SetOfItems GOTO( $I, X$ ) {
    initialize  $J$  to be the empty set;
    for ( each item  $[A \rightarrow \alpha \cdot X \beta, a]$  in  $I$  )
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;
    return CLOSURE( $J$ );
}
```

$I_0$
$S' \rightarrow \cdot S, \$$
$S \rightarrow \cdot CC, \$$
$C \rightarrow \cdot cC, c/d$
$C \rightarrow \cdot d, c/d$

$I_3$
$C \rightarrow c \cdot C, c/d$
$C \rightarrow \cdot cC, c/d$
$C \rightarrow \cdot d, c/d$

- Next, let  $X = c$ .
- We must close  $\{[C \rightarrow c \cdot C, c/d]\}$ .
- We add the  $C$ -productions with second component  $c/d$ , yielding  $I_3$ .

$S' \rightarrow S, S \rightarrow C\ C, C \rightarrow c\ C \mid d$

$\text{GOTO}(I_0, d)$

```
SetOfItems GOTO( $I, X$ ) {
    initialize  $J$  to be the empty set;
    for ( each item  $[A \rightarrow \alpha \cdot X \beta, a]$  in  $I$  )
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;
    return CLOSURE( $J$ );
}
```

$I_0$
$S' \rightarrow \cdot S, \$$
$S \rightarrow \cdot C C, \$$
$C \rightarrow \cdot c C, c/d$
$C \rightarrow \cdot d, c/d$

$I_4$
$C \rightarrow d \cdot, c/d$

- Finally, let  $X = d$ , and we wind up with the set of items,  $I_4$ .

$$S' \rightarrow S, S \rightarrow C\ C, C \rightarrow c\ C \mid d$$

```
SetOfItems GOTO( $I, X$ ) {  
    initialize  $J$  to be the empty set;  
    for ( each item  $[A \rightarrow \alpha \cdot X \beta, a]$  in  $I$  )  
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;  
    return CLOSURE( $J$ );  
}
```

$I_0$
$S' \rightarrow \cdot S, \$$
$S \rightarrow \cdot CC, \$$
$C \rightarrow \cdot cC, c/d$
$C \rightarrow \cdot d, c/d$

- We have finished considering GOTO on  $I_0$ .

$S' \rightarrow S, S \rightarrow C\ C, C \rightarrow c\ C \mid d$

```
SetOfItems GOTO( $I, X$ ) {  
    initialize  $J$  to be the empty set;  
    for ( each item  $[A \rightarrow \alpha \cdot X \beta, a]$  in  $I$  )  
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;  
    return CLOSURE( $J$ );  
}
```

$I_1$   
 $S' \rightarrow S \cdot, \$$

- We get no new sets from  $I_1$ .

$S' \rightarrow S, S \rightarrow C \cdot C, C \rightarrow c \cdot C \mid d$

```
SetOfItems GOTO( $I, X$ ) {  
    initialize  $J$  to be the empty set;  
    for ( each item  $[A \rightarrow \alpha \cdot X \beta, a]$  in  $I$  )  
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;  
    return CLOSURE( $J$ );  
}
```

$\text{GOTO}(I_2, C)$

$I_2$

$S \rightarrow C \cdot C, \$$

$C \rightarrow \cdot c C, \$$

$C \rightarrow \cdot d, \$$

$I_5$

$S \rightarrow C C \cdot, \$$

- $I_2$  has goto's on  $C$ ,  $c$ , and  $d$ .
- For  $\text{GOTO}(I_2, C)$  we get,  $I_5 : S \rightarrow C C \cdot, \$$
- No closure being needed.

$$S' \rightarrow S, S \rightarrow C \ C, C \rightarrow c \ C \mid d$$

```
SetOfItems GOTO( $I, X$ ) {  
    initialize  $J$  to be the empty set;  
    for ( each item  $[A \rightarrow \alpha \cdot X \beta, a]$  in  $I$  )  
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;  
    return CLOSURE( $J$ );  
}
```

$\text{GOTO}(I_2, c)$

$I_2$

$$\begin{aligned} S \rightarrow & C \cdot C, \$ \\ C \rightarrow & \cdot c C, \$ \\ C \rightarrow & \cdot d, \$ \end{aligned}$$

$I_6$

$$\begin{aligned} C \rightarrow & c \cdot C, \$ \\ C \rightarrow & \cdot c C, \$ \\ C \rightarrow & \cdot d, \$ \end{aligned}$$

- To compute  $\text{GOTO}(I_2, c)$  we take the closure of  $\{[C \rightarrow c \cdot C, \$]\}$ , to obtain  $I_6$ .

$S' \rightarrow S, S \rightarrow C\ C, C \rightarrow c\ C \mid d$

$I_3$
$C \rightarrow c \cdot C, c/d$
$C \rightarrow \cdot cC, c/d$
$C \rightarrow \cdot d, c/d$

$I_6$
$C \rightarrow c \cdot C, \$$
$C \rightarrow \cdot cC, \$$
$C \rightarrow \cdot d, \$$

- Note that  $I_6$  differs from  $I_3$  only in second components.

$$S' \rightarrow S, S \rightarrow C C, C \rightarrow c C \mid d$$

$I_3$
$C \rightarrow c \cdot C, c/d$
$C \rightarrow \cdot cC, c/d$
$C \rightarrow \cdot d, c/d$

$I_6$
$C \rightarrow c \cdot C, \$$
$C \rightarrow \cdot cC, \$$
$C \rightarrow \cdot d, \$$

- We shall see that it is common for several sets of LR(1) items for a grammar to have the same first components and differ in their second components.

$$S' \rightarrow S, S \rightarrow C C, C \rightarrow c C \mid d$$

$I_3$
$C \rightarrow c \cdot C, c/d$
$C \rightarrow \cdot cC, c/d$
$C \rightarrow \cdot d, c/d$

$I_6$
$C \rightarrow c \cdot C, \$$
$C \rightarrow \cdot cC, \$$
$C \rightarrow \cdot d, \$$

- When we construct the collection of sets of LR(0) items for the same grammar, each set of LR(0) items will coincide with the set of first components of one or more sets of LR(1) items.

$$S' \rightarrow S, S \rightarrow C \cdot C, C \rightarrow c \cdot C \mid d$$

```
SetOfItems GOTO( $I, X$ ) {  
    initialize  $J$  to be the empty set;  
    for ( each item  $[A \rightarrow \alpha \cdot X \beta, a]$  in  $I$  )  
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;  
    return CLOSURE( $J$ );  
}
```

$\text{GOTO}(I_2, d)$

$I_2$   
 $S \rightarrow C \cdot C, \$$   
 $C \rightarrow \cdot c C, \$$   
 $C \rightarrow \cdot d, \$$

$I_7$   
 $C \rightarrow d \cdot, \$$

- Continuing with the GOTO function for  $I_2$ ,  $\text{GOTO}(I_2, d)$  is seen as  $I_7$ .

$S' \rightarrow S, S \rightarrow C \ C, C \rightarrow c \ C \mid d$

```
SetOfItems GOTO( $I, X$ ) {  
    initialize  $J$  to be the empty set;  
    for ( each item  $[A \rightarrow \alpha \cdot X \beta, a]$  in  $I$  )  
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;  
    return CLOSURE( $J$ );  
}
```

$\text{GOTO}(I_3, c)$

$I_3$
$C \rightarrow c \cdot C, c/d$
$C \rightarrow \cdot cC, c/d$
$C \rightarrow \cdot d, c/d$

- Turning now to  $I_3$ , the GOTO's of  $I_3$  on  $c$  and  $d$  are  $I_3$  and  $I_4$ , respectively.

$S' \rightarrow S, S \rightarrow C \ C, C \rightarrow c \ C \mid d$

```
SetOfItems GOTO( $I, X$ ) {  
    initialize  $J$  to be the empty set;  
    for ( each item  $[A \rightarrow \alpha \cdot X \beta, a]$  in  $I$  )  
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;  
    return CLOSURE( $J$ );  
}
```

$\text{GOTO}(I_3, d)$

$I_3$   
 $C \rightarrow c \cdot C, c/d$   
 $C \rightarrow \cdot cC, c/d$   
 $C \rightarrow \cdot d, c/d$

$I_4$   
 $C \rightarrow d \cdot, c/d$

- Turning now to  $I_3$ , the GOTO's of  $I_3$  on  $c$  and  $d$  are  $I_3$  and  $I_4$ , respectively.

$$S' \rightarrow S, S \rightarrow C \ C, C \rightarrow c \ C \mid d$$

```
SetOfItems GOTO( $I, X$ ) {  
    initialize  $J$  to be the empty set;  
    for ( each item  $[A \rightarrow \alpha \cdot X \beta, a]$  in  $I$  )  
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;  
    return CLOSURE( $J$ );  
}
```

$\text{GOTO}(I_3, C)$

$I_3$

$C \rightarrow c \cdot C, c/d$   
 $C \rightarrow \cdot cC, c/d$   
 $C \rightarrow \cdot d, c/d$

$I_8$

$C \rightarrow cC \cdot, c/d$

■  $\text{GOTO}(I_3, C)$  is  $I_8$ .

$$S' \rightarrow S, S \rightarrow C\ C, C \rightarrow c\ C \mid d$$

```
SetOfItems GOTO( $I, X$ ) {  
    initialize  $J$  to be the empty set;  
    for ( each item  $[A \rightarrow \alpha \cdot X \beta, a]$  in  $I$  )  
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;  
    return CLOSURE( $J$ );  
}
```

$I_4$
$C \rightarrow d \cdot, c/d$
$I_5$
$S \rightarrow C C \cdot, \$$

- $I_4$  and  $I_5$  have no GOTO's, since all items have their dots at the right end.

$S' \rightarrow S, S \rightarrow C\ C, C \rightarrow c\ C \mid d$

```
SetOfItems GOTO( $I, X$ ) {  
    initialize  $J$  to be the empty set;  
    for ( each item  $[A \rightarrow \alpha \cdot X \beta, a]$  in  $I$  )  
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;  
    return CLOSURE( $J$ );  
}
```

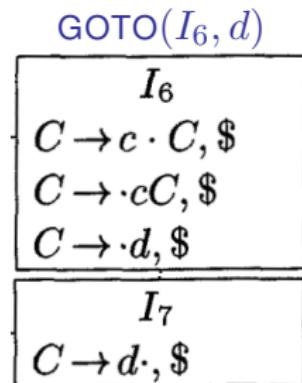
$\text{GOTO}(I_6, c)$

```
I6  
C  $\rightarrow$  c  $\cdot$  C, $  
C  $\rightarrow$   $\cdot$  cC, $  
C  $\rightarrow$   $\cdot$  d, $
```

- The GOTO's of  $I_6$  on  $c$  and  $d$  are  $I_6$  and  $I_7$ , respectively.

$S' \rightarrow S, S \rightarrow C\ C, C \rightarrow c\ C \mid d$

```
SetOfItems GOTO( $I, X$ ) {  
    initialize  $J$  to be the empty set;  
    for ( each item  $[A \rightarrow \alpha \cdot X \beta, a]$  in  $I$  )  
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;  
    return CLOSURE( $J$ );  
}
```



- The GOTO's of  $I_6$  on  $c$  and  $d$  are  $I_6$  and  $I_7$ , respectively.

$S' \rightarrow S, S \rightarrow C \ C, C \rightarrow c \ C \mid d$

```
SetOfItems GOTO( $I, X$ ) {  
    initialize  $J$  to be the empty set;  
    for ( each item  $[A \rightarrow \alpha \cdot X \beta, a]$  in  $I$  )  
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;  
    return CLOSURE( $J$ );  
}
```

$\text{GOTO}(I_6, C)$

$I_6$   
 $C \rightarrow c \cdot C, \$$   
 $C \rightarrow \cdot cC, \$$   
 $C \rightarrow \cdot d, \$$

$I_9$   
 $C \rightarrow cC \cdot, \$$

■  $\text{GOTO}(I_6, C)$  is  $I_9$ .

$S' \rightarrow S, S \rightarrow C\ C, C \rightarrow c\ C \mid d$

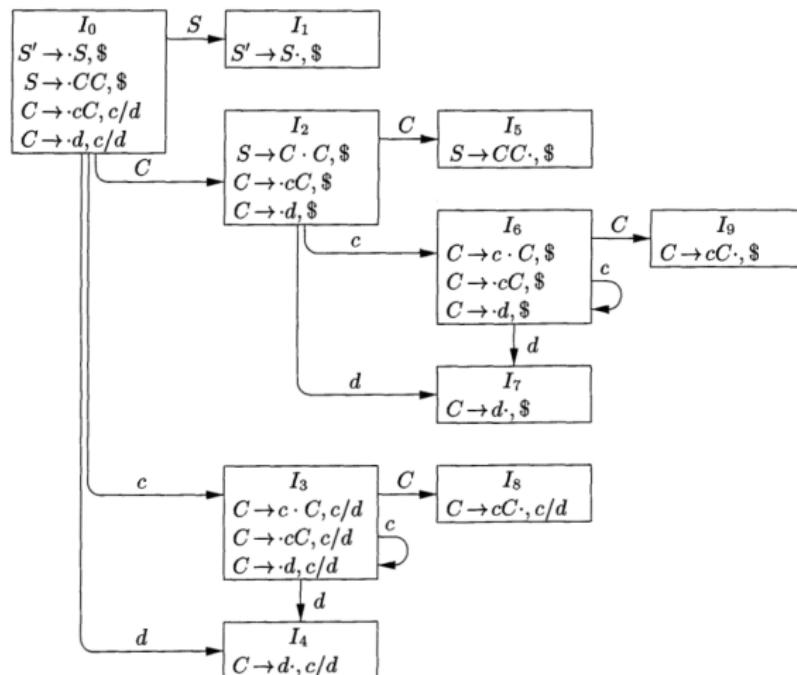
```
SetOfItems GOTO( $I, X$ ) {  
    initialize  $J$  to be the empty set;  
    for ( each item  $[A \rightarrow \alpha \cdot X \beta, a]$  in  $I$  )  
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;  
    return CLOSURE( $J$ );  
}
```

$I_7$   
 $C \rightarrow d \cdot, \$$

- The remaining sets of items yield no GOTO's, so we are done.

## Example — *continued*

$S' \rightarrow S, S \rightarrow C \ C, C \rightarrow cC \mid d$



- Shows the ten sets of items with their GOTO's.

# Canonical LR(1) Parsing Tables

- We now give the rules for constructing the LR(1) ACTION and GOTO functions from the sets of LR(1) items.
- These functions are represented by a table, as before.
- The only difference is in the values of the entries.

## Construction of canonical-LR parsing tables

- **INPUT:** An augmented grammar  $G'$ .
- **OUTPUT:** The canonical-LR parsing table functions ACTION and GOTO for  $G'$ .

## Construction of canonical-LR parsing tables

### **METHOD:**

1. Construct  $C' = \{I_0, I_1, I_2, \dots, I_n\}$ , the collection of sets of LR(1) items for  $G'$ .

## Construction of canonical-LR parsing tables

### METHOD:

2. State  $i$  of the parser is constructed from  $I_i$ .
  - The parsing actions for state  $i$  are determined as follows:
    - (a) If  $[A \rightarrow \alpha \cdot a\beta]$  is in  $I_i$ , and  $\text{GOTO}(I_i, a) = I_j$ , then set  $\text{ACTION}[i, a]$  to “shift  $j$ .”
      - Here  $a$  must be a terminal.
    - (b) If  $[A \rightarrow \alpha \cdot, a]$  is in  $I_i$ ,  $A \neq S'$ , then set  $\text{ACTION}[i, a]$  to “reduce  $A \rightarrow \alpha$ ”.
      - Here  $A$  may not be  $S'$ .
    - (c) If  $[S' \rightarrow S \cdot, \$]$  is in  $I_i$ , then set  $\text{ACTION}[i, \$]$  to “accept.”
  - If any conflicting actions result from the above rules, we say the grammar is not  $LR(1)$ .
    - The algorithm fails to produce a parser in this case.

## Construction of canonical-LR parsing tables

### **METHOD:**

3. The goto transitions for state  $i$  are constructed for all nonterminals  $A$  using the rule: If  $\text{GOTO}(I_i, A) = I_j$ , then  $\text{GOTO}[i, A] = j$ .

## Construction of canonical-LR parsing tables

### **METHOD:**

4. All entries not defined by rules (2) and (3) are made “error.”

## Construction of canonical-LR parsing tables

### **METHOD:**

5. The initial state of the parser is the one constructed from the set of items containing  $[S' \rightarrow \cdot S, \$]$ .

- The table formed from the parsing action and goto functions produced by **LR-parsing algorithm** is called the canonical LR(1) parsing table.
- An LR parser using this table is called a canonical-LR(1) parser.
- If the parsing action function has no multiply defined entries, then the given grammar is called an LR(1) grammar.
- As before, we omit the “(1)” if it is understood.

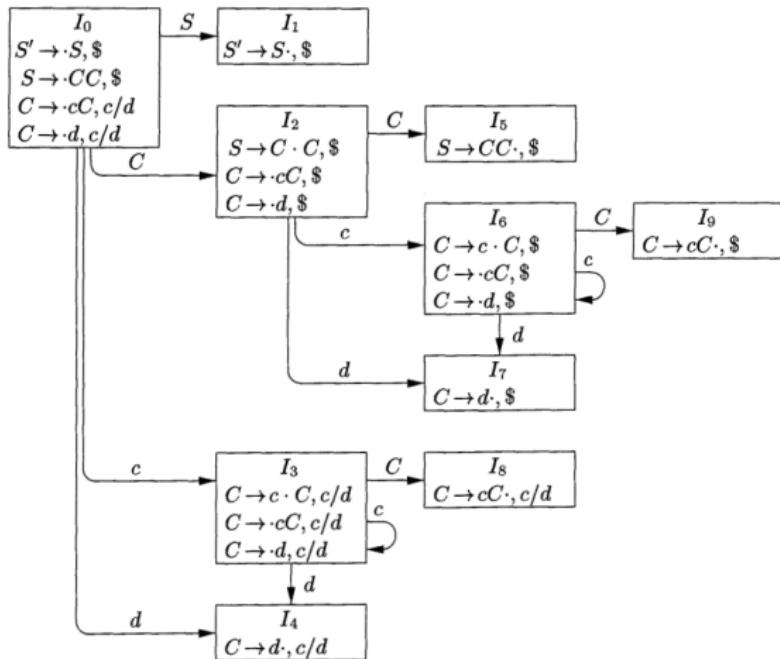
# Example

- The canonical parsing table for the augmented grammar is shown.

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

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		

- 0  $S' \rightarrow S$
  - 1  $S \rightarrow C C$
  - 2  $C \rightarrow cC$
  - 3  $C \rightarrow d$



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		

- Every SLR(1) grammar is an LR(1) grammar, but for an SLR(1) grammar the canonical LR parser may have more states than the SLR parser for the same grammar.
- The grammar of the previous examples is SLR and has an SLR parser with seven states, compared with the ten just shown.

# Constructing LALR Parsing Tables

- We now introduce our last parser construction method, the LALR (*lookahead-LR*) technique.

# Constructing LALR Parsing Tables — *continued*

- This method is often used in practice.
- Because the tables obtained by it are considerably smaller than the canonical LR tables.
- Yet most common syntactic constructs of programming languages can be expressed conveniently by an LALR grammar.

# Constructing LALR Parsing Tables — *continued*

- The same is almost true for SLR grammars, but there are a few constructs that cannot be conveniently handled by SLR techniques.

- For a comparison of parser size, the SLR and LALR tables for a grammar always have the same number of states, and this number is typically several hundred states for a language like C.
- The canonical LR table would typically have several thousand states for the same-size language.
- Thus, it is much easier and more economical to construct SLR and LALR tables than the canonical LR tables.

- For a comparison of parser size, the SLR and LALR tables for a grammar always have the same number of states, and this number is typically several hundred states for a language like C.
- The canonical LR table would typically have several thousand states for the same-size language.
- Thus, it is much easier and more economical to construct SLR and LALR tables than the canonical LR tables.

# A Bit of Elaboration on the Issue of Constructing LALR Parsing Tables

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/140%20LALR%20Parsing.pdf>

- A canonical LR(1) parser splits states based on differing lookahead sets.
- It can have many more states than the corresponding SLR(1) or LR(0) parser.
- Potentially it could require splitting a state with just one item into a different state for each subset of the possible lookaheads.
- In a pathological case, this means the entire power set of its FOLLOW set, which theoretically could contain all terminals.

# A Bit of Elaboration on the Issue of Constructing LALR Parsing Tables — *continued*

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/140%20LALR%20Parsing.pdf>

- It never actually gets that bad in practice.
- But a canonical LR(1) parser for a programming language might have an order of magnitude more states than an SLR(1) parser.
- Is there something in between?

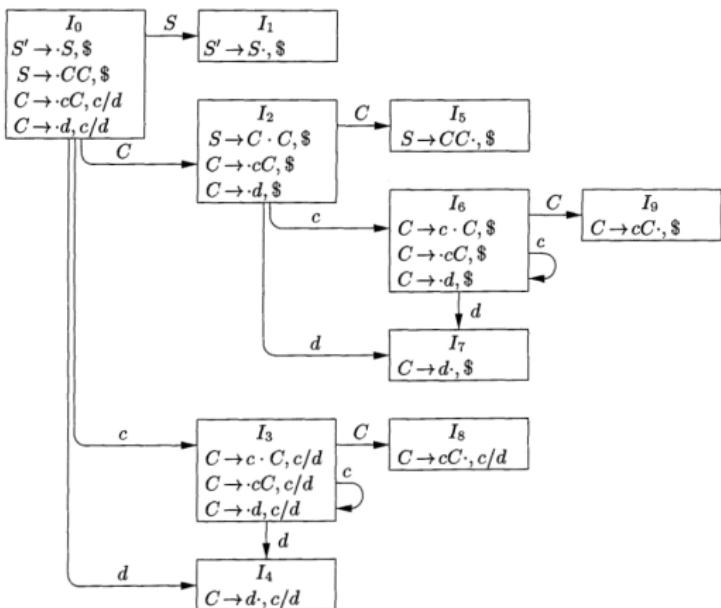
# A Bit of Elaboration on the Issue of Constructing LALR Parsing Tables — *continued*

<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/140%20LALR%20Parsing.pdf>

- With LALR (lookahead LR) parsing, we attempt to reduce the number of states in an LR(1) parser by merging similar states.
- This reduces the number of states to the same as SLR(1), but still retains some of the power of the LR(1) lookahead.

# Constructing LALR Parsing Tables — *continued*

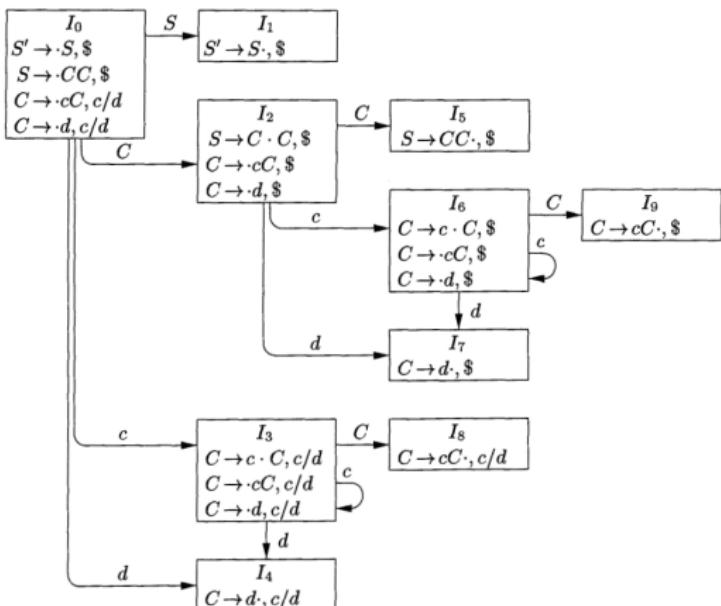
$S' \rightarrow S$   
 $S \rightarrow C C$   
 $C \rightarrow cC \mid d$



- By way of introduction, let us again consider the grammar, whose sets of LR(1) items are shown.

# Constructing LALR Parsing Tables — *continued*

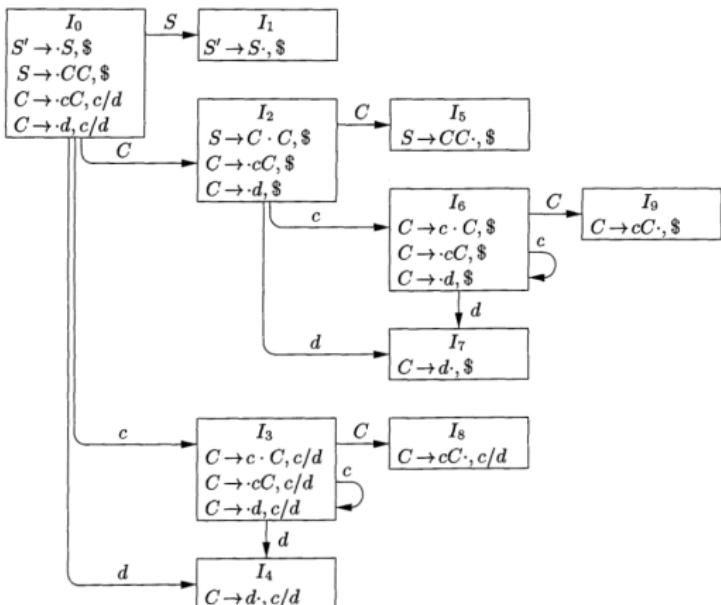
$S' \rightarrow S$   
 $S \rightarrow C C$   
 $C \rightarrow cC \mid d$



- Take a pair of similar looking states, such as  $I_4$  and  $I_7$ .
- Each of these states has only items with first component  $C \rightarrow d \dots$

# Constructing LALR Parsing Tables — *continued*

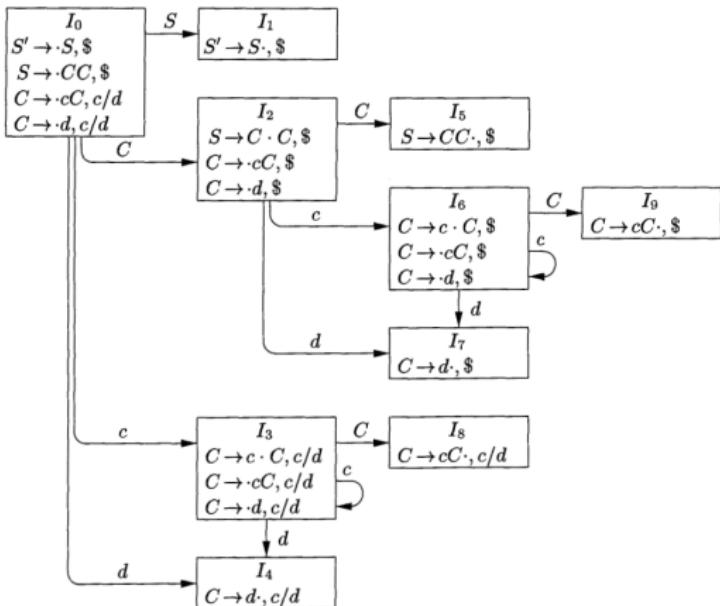
$S' \rightarrow S$   
 $S \rightarrow C C$   
 $C \rightarrow cC \mid d$



- In  $I_4$ , the lookaheads are  $c$  or  $d$ .
- In  $I_7$ ,  $\$$  is the only lookahead.

# Constructing LALR Parsing Tables — *continued*

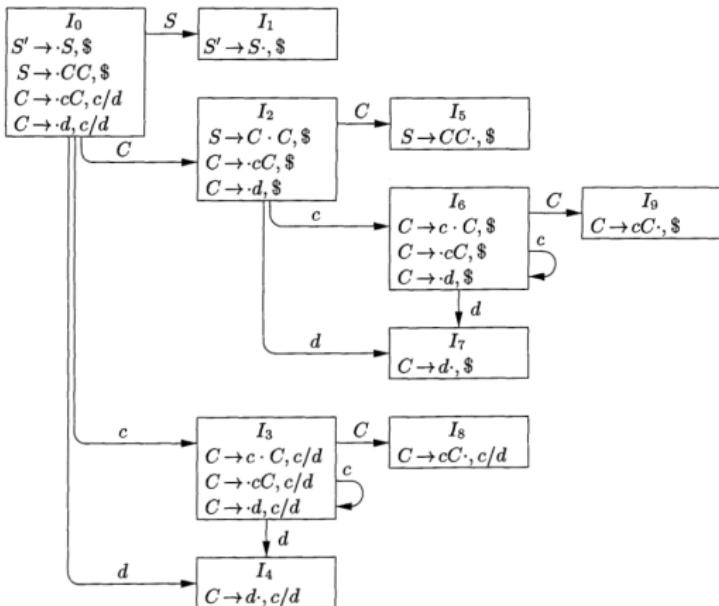
$S' \rightarrow S$   
 $S \rightarrow C C$   
 $C \rightarrow cC \mid d$



- To see the difference between the roles of  $I_4$  and  $I_7$  in the parser, note that the grammar generates the regular language  $c^*dc^*d$ .

# Constructing LALR Parsing Tables — *continued*

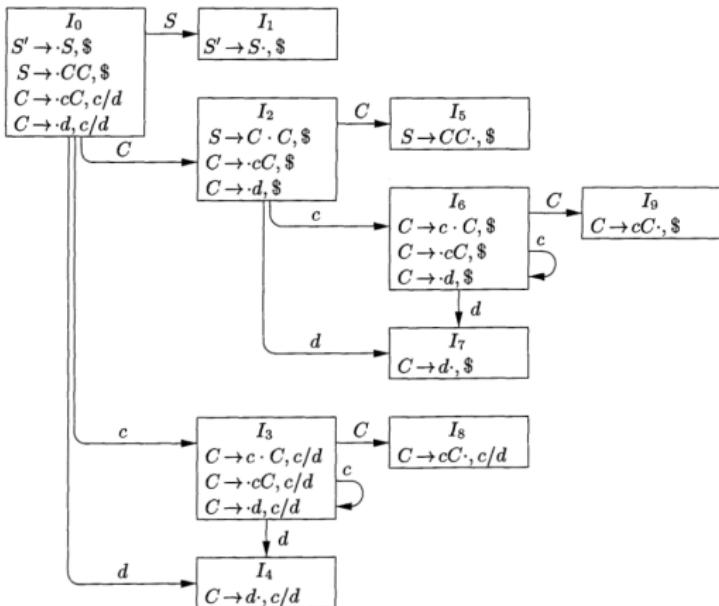
$S' \rightarrow S$   
 $S \rightarrow C C$   
 $C \rightarrow cC \mid d$



- When reading an input `cc`...`cd``cc`...`cd`, the parser shifts the first group of `c`'s and their following `d` onto the stack, entering state 4 after reading the `d`.

# Constructing LALR Parsing Tables — *continued*

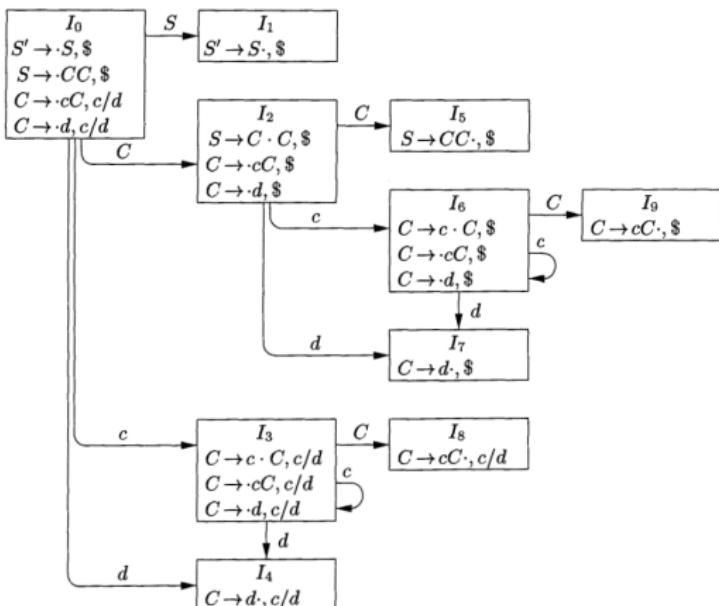
$S' \rightarrow S$   
 $S \rightarrow C C$   
 $C \rightarrow cC \mid d$



- The parser then calls for a reduction by  $C \rightarrow d$ , provided the next input symbol is  $c$  or  $d$ .
- The requirement that  $c$  or  $d$  follow makes sense, since these are the symbols that could begin strings in  $c^*d$ .

# Constructing LALR Parsing Tables — *continued*

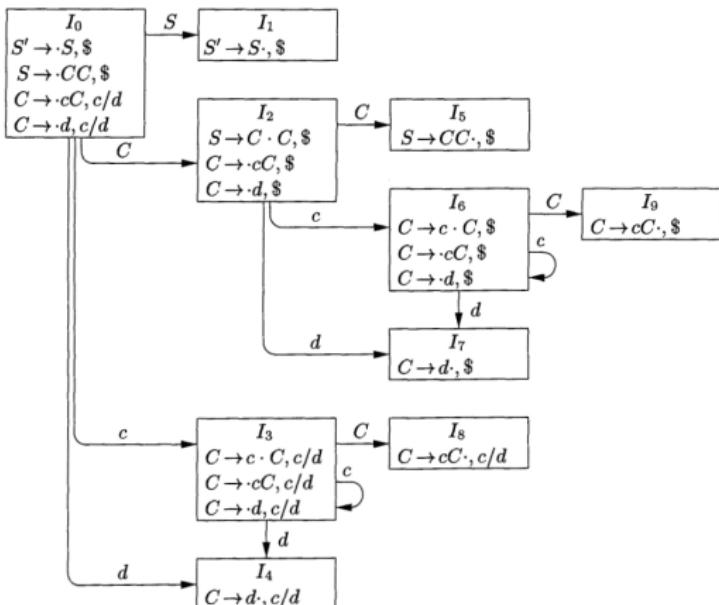
$S' \rightarrow S$   
 $S \rightarrow C C$   
 $C \rightarrow cC \mid d$



- If  $\$$  follows the first  $d$ , we have an input like  $ccd$ , which is not in the language, and state 4 correctly declares an error if  $\$$  is the next input.

# Constructing LALR Parsing Tables — *continued*

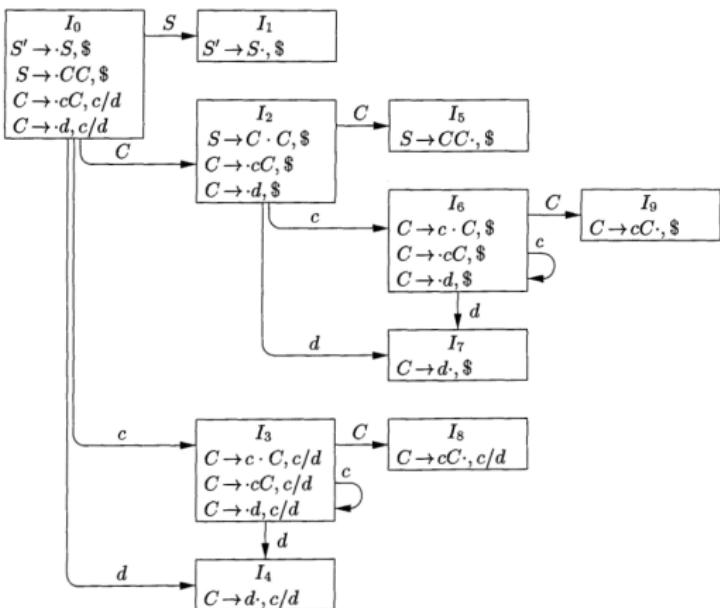
$S' \rightarrow S$   
 $S \rightarrow C C$   
 $C \rightarrow cC \mid d$



- The parser enters state 7 after reading the second  $d$ .
- Then, the parser must see  $\$$  on the input, or it started with a string not of the form  $c^*dc^*d$ .

# Constructing LALR Parsing Tables — *continued*

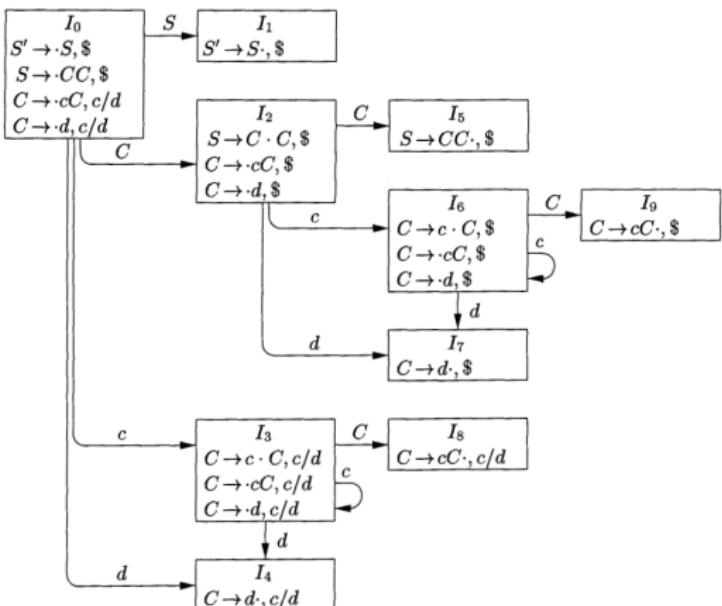
$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & C \ C \\ C & \rightarrow & cC \mid d \end{array}$$



- It thus makes sense that state 7 should reduce by  $C \rightarrow d$  on input  $\$$  and declare error on inputs  $c$  or  $d$ .

# Constructing LALR Parsing Tables — *continued*

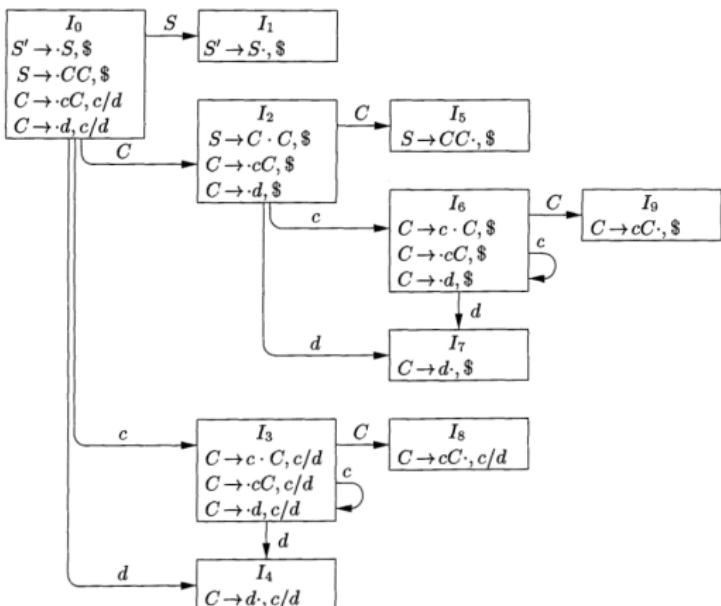
$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & C \ C \\ C & \rightarrow & cC \mid d \end{array}$$



- Let us now replace  $I_4$  and  $I_7$  by  $I_{47}$ , the union of  $I_4$  and  $I_7$ , consisting of the set of three items represented by  $[C \rightarrow d \cdot, c/d/\$]$ .

# Constructing LALR Parsing Tables — *continued*

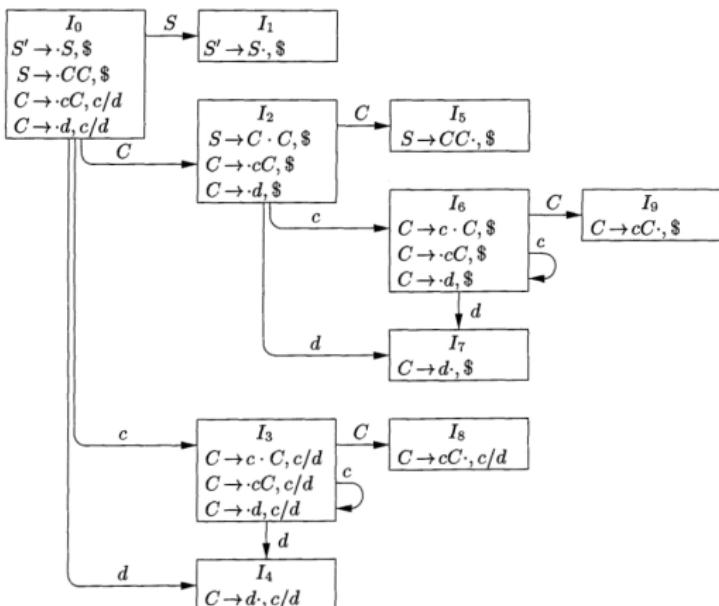
$S' \rightarrow S$   
 $S \rightarrow C C$   
 $C \rightarrow cC \mid d$



- The goto's on  $d$  to  $I_4$  or  $I_7$  from  $I_0$ ,  $I_2$ ,  $I_3$ , and  $I_6$  now enter  $I_{47}$ .

# Constructing LALR Parsing Tables — *continued*

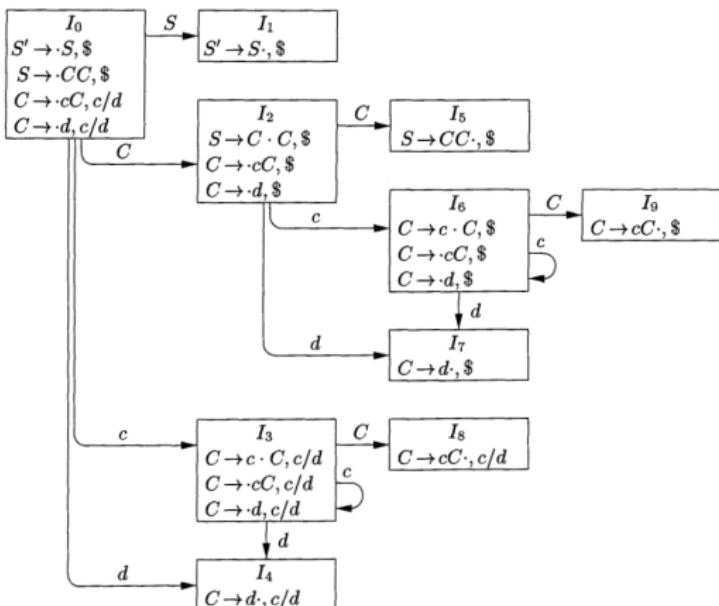
$S' \rightarrow S$   
 $S \rightarrow C C$   
 $C \rightarrow cC \mid d$



- The action of state 47 is to reduce on any input.

# Constructing LALR Parsing Tables — *continued*

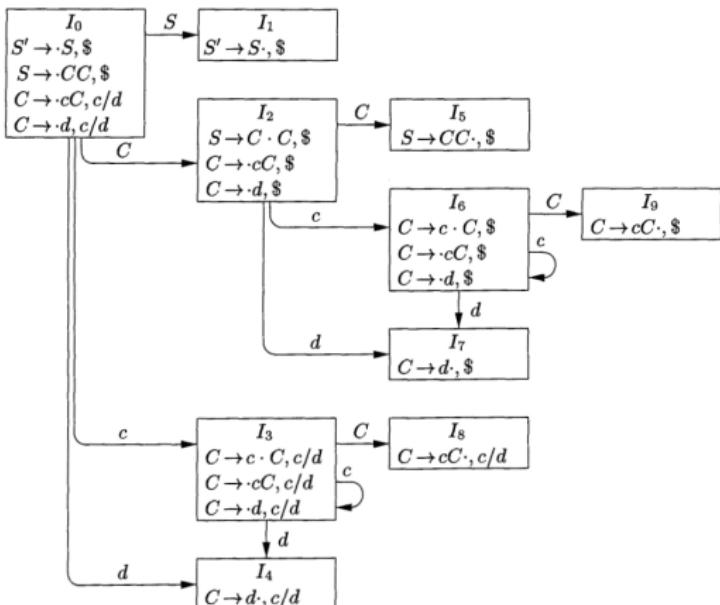
$S' \rightarrow S$   
 $S \rightarrow C C$   
 $C \rightarrow cC \mid d$



- The revised parser behaves essentially like the original, although it might reduce  $d$  to  $C$  in circumstances where the original would declare error, for example, on input like  $ccd$  or  $cdcdc$ .

# Constructing LALR Parsing Tables — *continued*

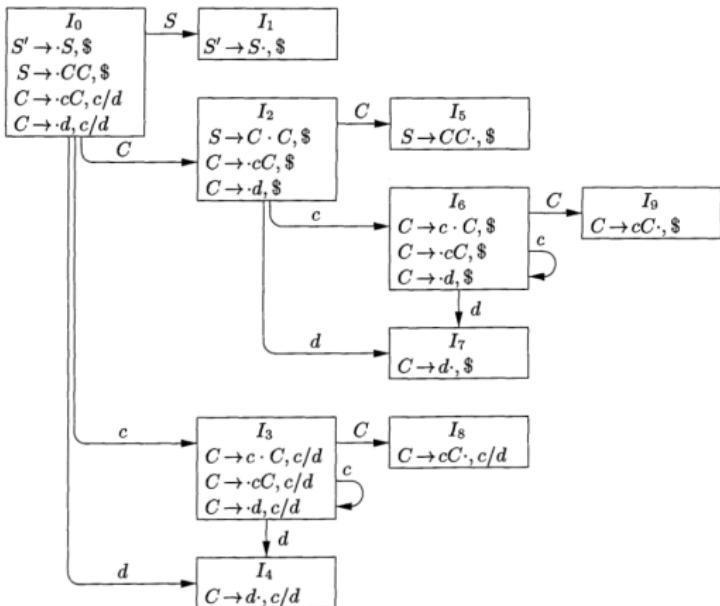
$S' \rightarrow S$   
 $S \rightarrow C C$   
 $C \rightarrow cC \mid d$



- The error will eventually be caught.
- In fact, it will be caught before any more input symbols are shifted.

# Constructing LALR Parsing Tables — *continued*

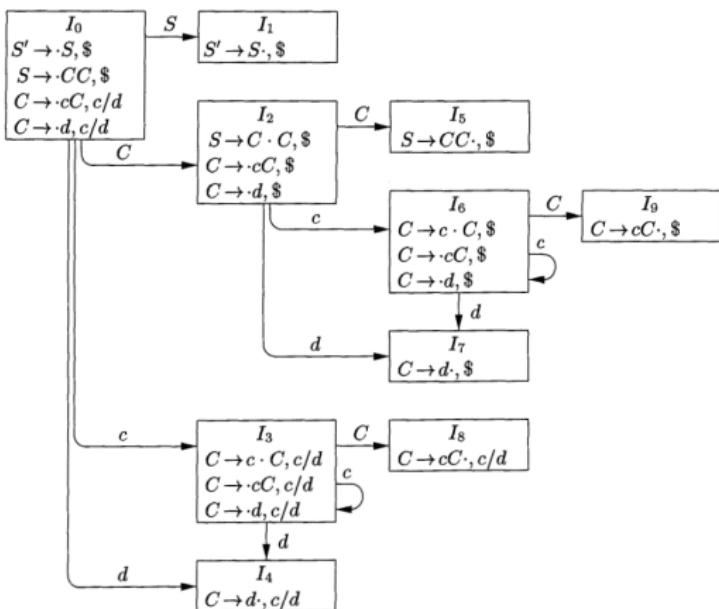
$S' \rightarrow S$   
 $S \rightarrow C C$   
 $C \rightarrow cC \mid d$



- More generally, we can look for sets of LR(1) items having the same core, that is, set of first components.
- And we may merge these sets with common cores into one set of items.

# Constructing LALR Parsing Tables — *continued*

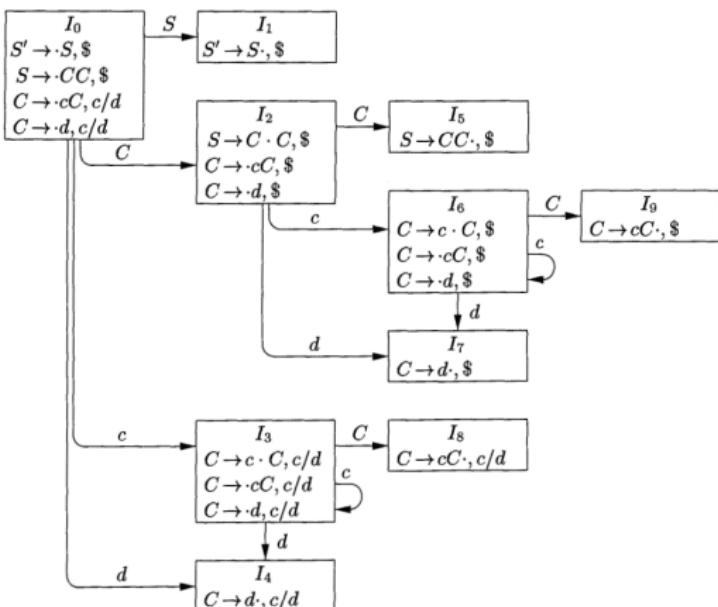
$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & C \ C \\ C & \rightarrow & cC \mid d \end{array}$$



- For example,  $I_4$  and  $I_7$  form such a pair, with core  $\{C \rightarrow d \cdot\}$ .
- Similarly,  $I_3$  and  $I_6$  form another pair, with core  $\{C \rightarrow c \cdot C, C \rightarrow \cdot cC, C \rightarrow \cdot d\}$ .

# Constructing LALR Parsing Tables — *continued*

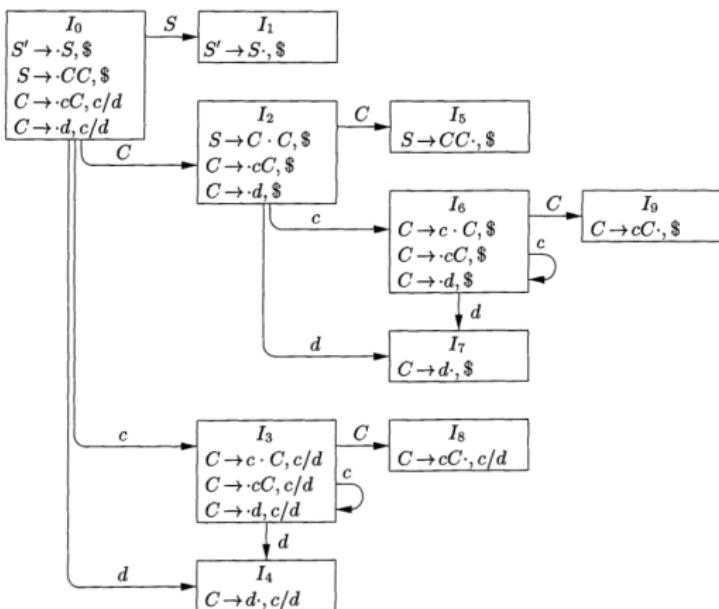
$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & C \ C \\ C & \rightarrow & cC \mid d \end{array}$$



- There is one more pair,  $I_8$  and  $I_9$ , with common core  $\{C \rightarrow cC \cdot\}$ .

# Constructing LALR Parsing Tables — *continued*

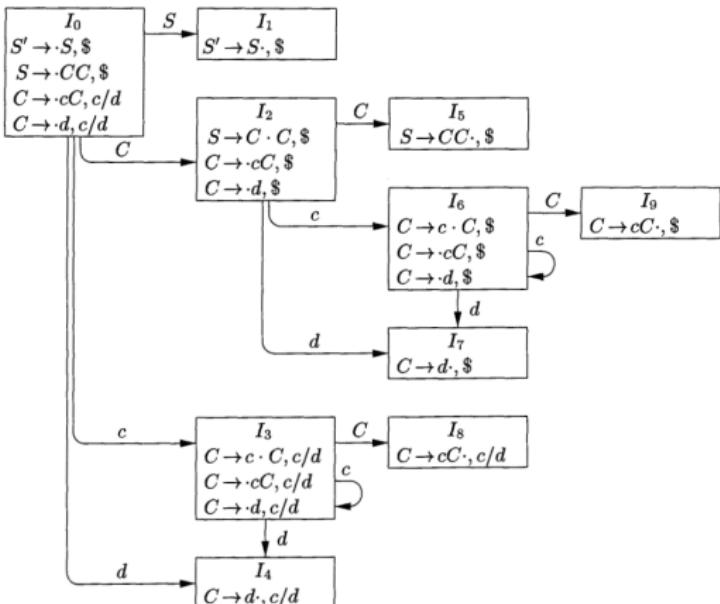
$S' \rightarrow S$   
 $S \rightarrow C C$   
 $C \rightarrow cC \mid d$



- Note that, in general, a core is a set of LR(0) items for the grammar at hand, and that an LR(1) grammar may produce more than two sets of items with the same core.

# Constructing LALR Parsing Tables — *continued*

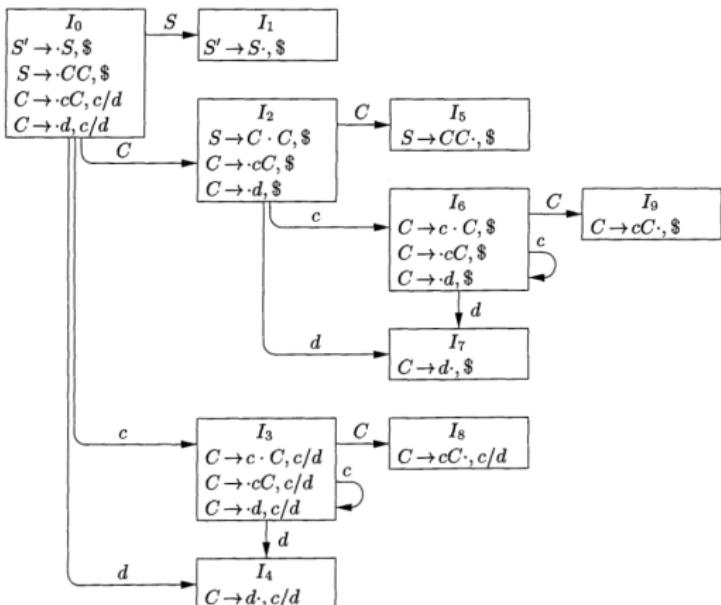
$S' \rightarrow S$   
 $S \rightarrow C C$   
 $C \rightarrow cC \mid d$



- Since the core of  $\text{GOTO}(I, X)$  depends only on the core of  $I$ , the goto's of merged sets can themselves be merged.
- Thus, there is no problem revising the goto function as we merge sets of items.

# Constructing LALR Parsing Tables — *continued*

$S' \rightarrow S$   
 $S \rightarrow C C$   
 $C \rightarrow cC \mid d$



- The action functions are modified to reflect the non-error actions of all sets of items in the merger.

- Suppose we have an LR(1) grammar, that is, one whose sets of LR(1) items produce no parsing-action conflicts.
- If we replace all states having the same core with their union, it is possible that the resulting union will have a conflict.
- But it is unlikely for the following reason.

## Constructing LALR Parsing Tables — *continued*

- Suppose in the union there is a conflict on lookahead  $a$  because there is an item  $[A \rightarrow \alpha \cdot, a]$  calling for a reduction by  $A \rightarrow \alpha$ .
- And there is another item  $[B \rightarrow \beta \cdot a\gamma, b]$  calling for a shift.
- Then some set of items from which the union was formed has item  $[A \rightarrow \alpha \cdot, a]$ , and since the cores of all these states are the same, it must have an item  $[B \rightarrow \beta \cdot a\gamma, c]$  for some  $c$ .
- But then this state has the same shift/reduce conflict on  $a$ , and the grammar was not LR(1) as we assumed.

# Constructing LALR Parsing Tables — Further Explanations

- In LR parsing, states are merged when they have the same core.
- The core includes information about the positions in the production rules but excludes the lookahead symbols.
- If there's an item  $[A \rightarrow \alpha \cdot, a]$  in the set, indicating a reduction conflict on  $a$ , and the cores are the same, it logically follows that there must be another item  $[B \rightarrow \beta \cdot a \gamma, c]$  in the same set.
- Here  $c$  is the lookahead associated with the item.

# Constructing LALR Parsing Tables — Further Explanations — *continued*

- This is because the identical cores imply that the same set of productions and positions are present in the merged states.
- And the different lookahead symbols are the key factor causing the conflict.
- In summary, the justification relies on the principle that states with identical cores contribute to the same set during merging.
- If a specific item is present in that set, a similar item with a different lookahead must also be present to account for the conflict scenario involving the lookahead symbol  $a$ .

# Constructing LALR Parsing Tables — *continued*

- Thus, the merging of states with common cores can never produce a shift/reduce conflict that was not present in one of the original states.
- Because shift actions depend only on the core, not the lookahead.
- It is possible, however, that a merger will produce a reduce/reduce conflict, as the following example shows.

# Constructing LALR Parsing Tables — *continued*

- Thus, the merging of states with common cores can never produce a shift/reduce conflict that was not present in one of the original states.
- Because shift actions depend only on the core, not the lookahead.
- It is possible, however, that a merger will produce a reduce/reduce conflict, as the following example shows.

# Example

- Consider the grammar,

$$S' \rightarrow S$$

$$S \rightarrow a A d \mid a B e \mid b B d \mid b A e$$

$$A \rightarrow c$$

$$B \rightarrow c$$

- Which generates the four strings *acd*, *ace*, *bcd*, and *bce*.
- The reader can check that the grammar is LR(1) by constructing the sets of items.

# Example

- Consider the grammar,

$$S' \rightarrow S$$

$$S \rightarrow a A d \mid a B e \mid b B d \mid b A e$$

$$A \rightarrow c$$

$$B \rightarrow c$$

- Which generates the four strings *acd*, *ace*, *bcd*, and *bce*.
- The reader can check that the grammar is LR(1) by constructing the sets of items.

## Example — *continued*

$S' \rightarrow S, S \rightarrow a A d \mid a B e \mid b B d \mid b A e, A \rightarrow c, B \rightarrow c$

---

- The set of items  $\{[A \rightarrow c \cdot, d], [B \rightarrow c \cdot, e]\}$  is valid for viable prefix  $ac$ .
  - The set of items  $\{[A \rightarrow c \cdot, e], [B \rightarrow c \cdot, d]\}$  is valid for viable prefix  $bc$ .
- 
- Neither of these sets has a conflict.
  - And their cores are the same.

## Example — *continued*

$S' \rightarrow S, S \rightarrow a A d \mid a B e \mid b B d \mid b A e, A \rightarrow c, B \rightarrow c$

---

- The set of items  $\{[A \rightarrow c \cdot, d], [B \rightarrow c \cdot, e]\}$  is valid for viable prefix  $ac$ .
  - The set of items  $\{[A \rightarrow c \cdot, e], [B \rightarrow c \cdot, d]\}$  is valid for viable prefix  $bc$ .
  - However, their union, which is,
- 

$A \rightarrow c \cdot, d/e$

$B \rightarrow c \cdot, d/e$

generates a reduce/reduce conflict.

- Reductions by both  $A \rightarrow c$  and  $B \rightarrow c$  are called for on inputs  $d$  and  $e$ .

# Constructing LALR Parsing Tables — *continued*

- We are now prepared to give the first of two LALR table-construction algorithms.
- The general idea is to construct the sets of LR(1) items, and if no conflicts arise, merge sets with common cores.
- We then construct the parsing table from the collection of merged sets of items.

- The method we are about to describe serves primarily as a definition of LALR(1) grammars.
- Constructing the entire collection of LR(1) sets of items requires too much space and time to be useful in practice.

# Constructing LALR Parsing Tables — *continued*

An easy, but space-consuming LALR table construction

- **INPUT:** An augmented grammar  $G'$ .
- **OUTPUT:** The LALR parsing-table functions ACTION and GOTO for  $G'$ .

# Constructing LALR Parsing Tables — *continued*

An easy, but space-consuming LALR table construction

## **METHOD:**

1. Construct  $C = \{I_0, I_1, I_2, \dots, I_n\}$ , the collection of sets of LR(1) items.

An easy, but space-consuming LALR table construction

## **METHOD:**

2. For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by their union.

An easy, but space-consuming LALR table construction

## METHOD:

3. Let  $C' = \{J_0, J_1, J_2, \dots, J_m\}$  be the resulting sets of LR(1) items.
  - The parsing actions for state  $i$  are constructed from  $J_i$  in the same manner as in Algorithm “Construction of canonical-LR parsing tables”.
  - If there is a parsing action conflict, the algorithm fails to produce a parser, and the grammar is said not to be LALR(1).

An easy, but space-consuming LALR table construction

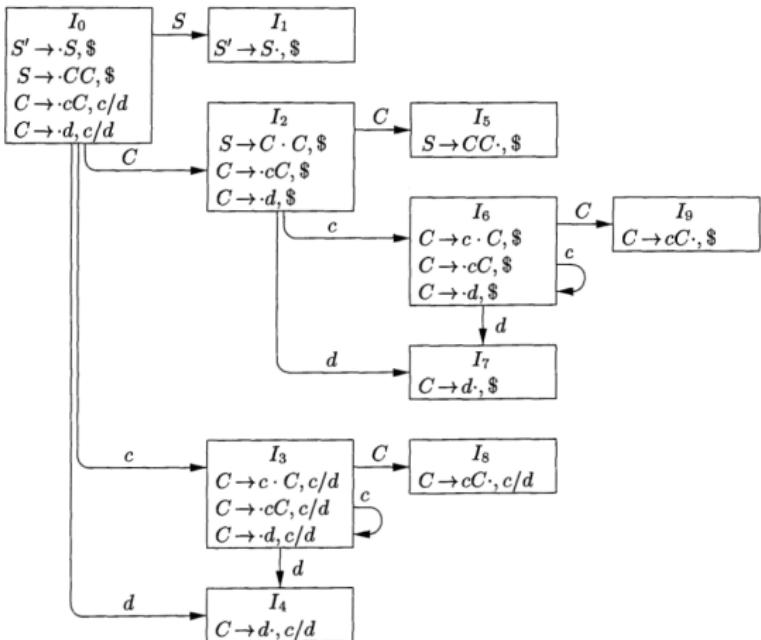
## METHOD:

4. The GOTO table is constructed as follows.
  - If  $J$  is the union of one or more sets of LR(1) items, that is,  $J = I_1 \cap I_2 \cap \dots \cap I_k$ , then the cores of  $\text{GOTO}(I_1, X)$ ,  $\text{GOTO}(I_2, X)$ ,  $\dots$ ,  $\text{GOTO}(I_k, X)$  are the same.
  - Since  $I_1, I_2, \dots, I_k$  all have the same core.
  - Let  $K$  be the union of all sets of items having the same core as  $\text{GOTO}(I_1, X)$ .
  - Then  $\text{GOTO}(J, X) = K$ .

- The table produced by the above algorithm is called the LALR parsing table for  $G$ .
- If there are no parsing action conflicts, then the given grammar is said to be an LALR(1) grammar.
- The collection of sets of items constructed in step (3) is called the LALR(1) collection.

# Example

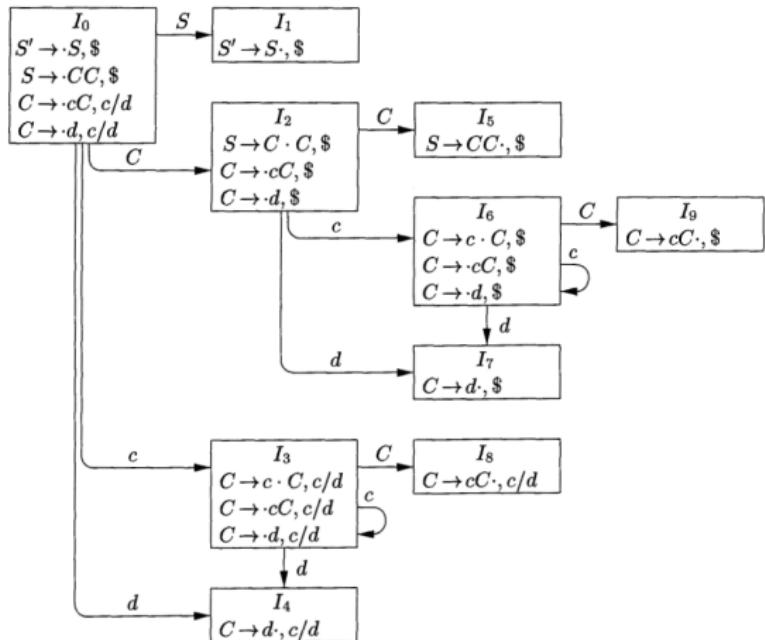
$S' \rightarrow S$   
 $S \rightarrow C C$   
 $C \rightarrow cC \mid d$



■ Again consider the grammar whose GOTO graph is shown.

## Example

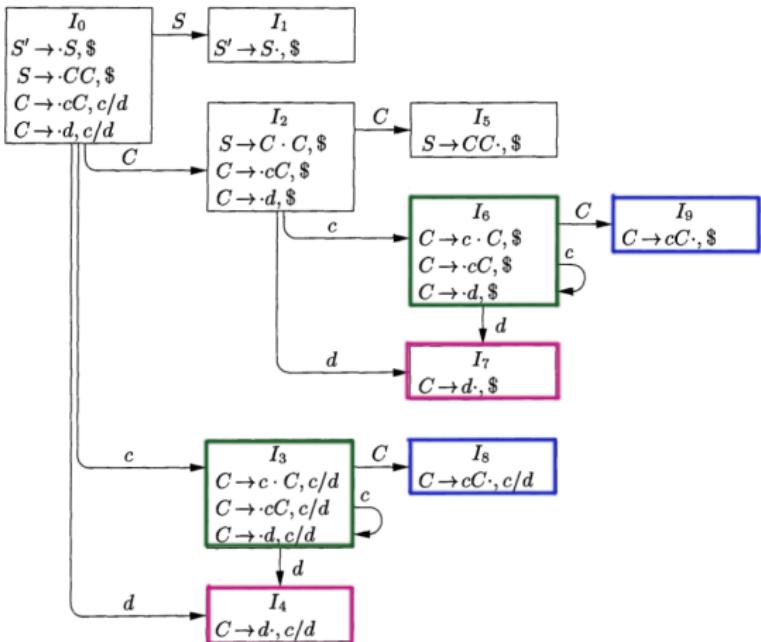
$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & C \ C \\ C & \rightarrow & cC \mid d \end{array}$$



- There are three pairs of sets of items that can be merged.

# Example

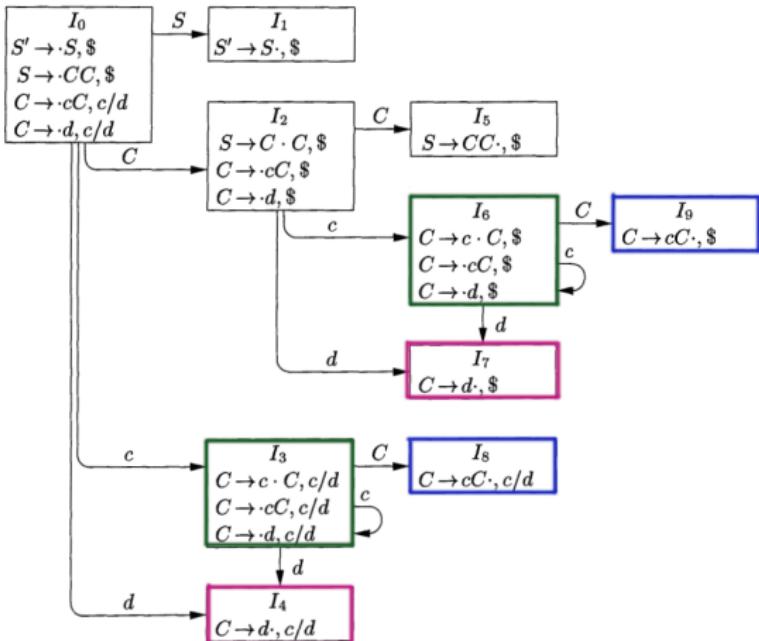
$S' \rightarrow S$   
 $S \rightarrow C C$   
 $C \rightarrow cC \mid d$



- There are three pairs of sets of items that can be merged.

# Example

$$\begin{array}{l}
 S' \rightarrow S \\
 S \rightarrow C \ C \\
 C \rightarrow cC \mid d
 \end{array}$$

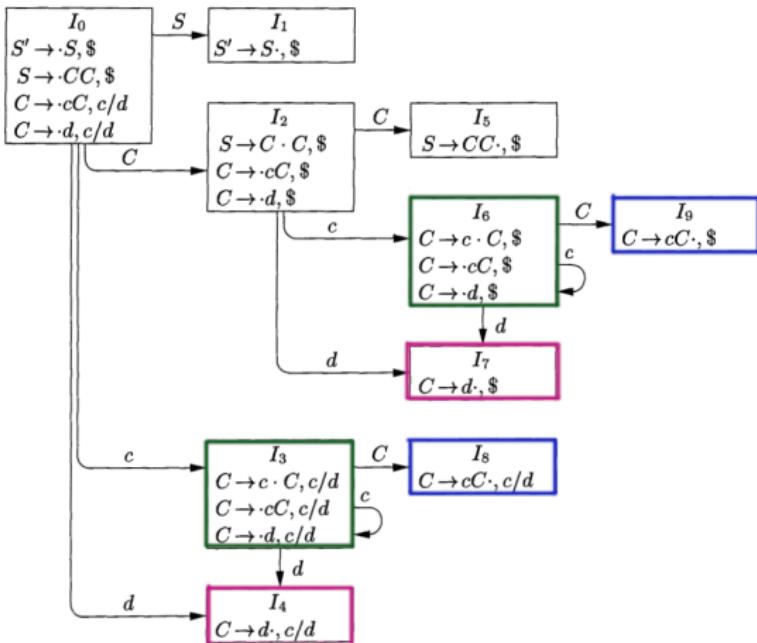


■  $I_3$  and  $I_6$  are replaced by their union:

$$\begin{array}{l}
 I_{36} : \quad C \rightarrow c \cdot C, c/d/\$ \\
 \quad \quad \quad C \rightarrow \cdot cC, c/d/\$ \\
 \quad \quad \quad C \rightarrow \cdot d, c/d/\$
 \end{array}$$

# Example

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & CC \\ C & \rightarrow & cC \mid d \end{array}$$

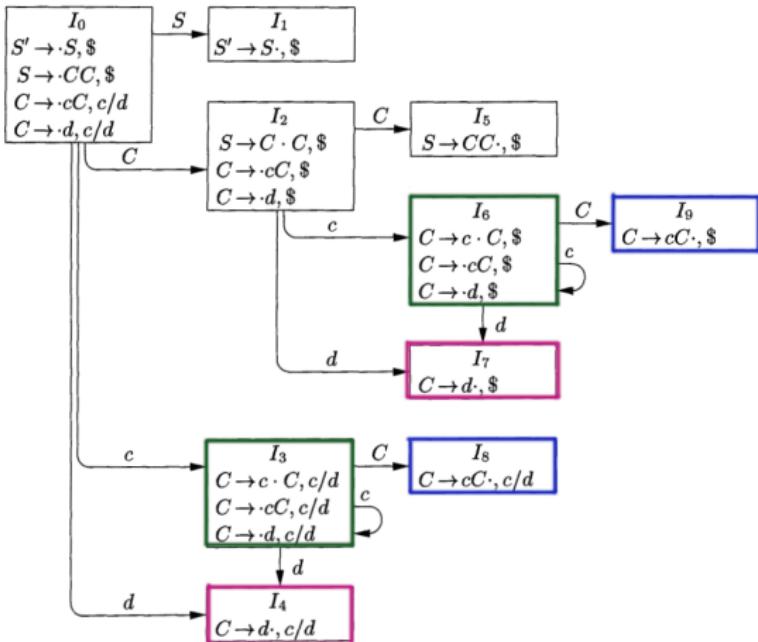


■  $I_4$  and  $I_7$  are replaced by their union:

$$I_{47} : C \rightarrow d \cdot, c/d, \$$$

# Example

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & CC \\ C & \rightarrow & cC \mid d \end{array}$$



- $I_8$  and  $I_9$  are replaced by their union:

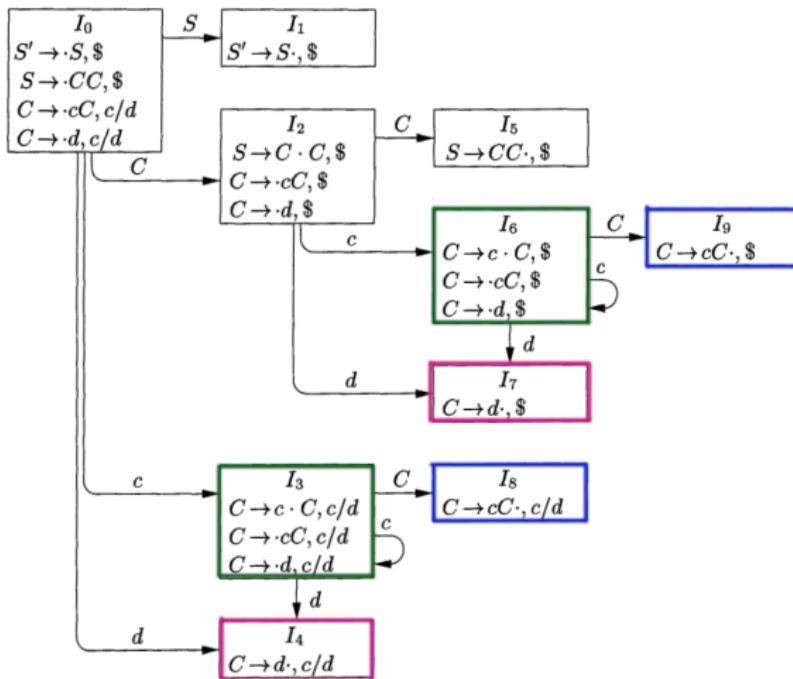
$$I_{89} : C \rightarrow cC \cdot , c/d / \$$$

$S' \rightarrow S$  $S \rightarrow C C$  $C \rightarrow cC \mid d$ 

■ The LALR action and goto functions for the condensed sets of items are shown.

STATE	ACTION			GOTO	
	<i>c</i>	<i>d</i>	\$	<i>S</i>	<i>C</i>
0	s3	s4		1	2
1			acc		
2	s6	s7		5	
3	s3	s4		8	
4	r3	r3			
5			r1		
6	s6	s7		9	
7			r3		
8	r2	r2			
9			r2		

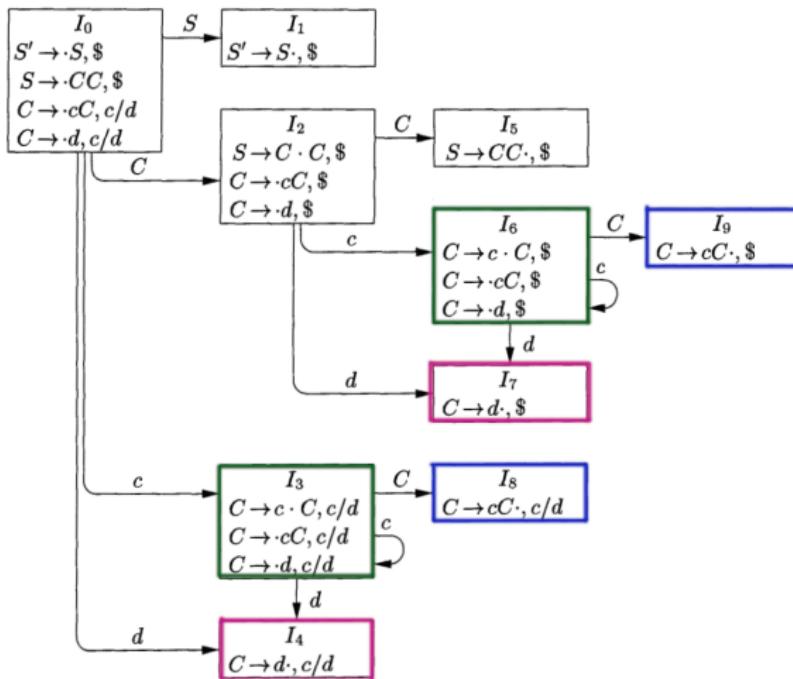
STATE	ACTION			GOTO	
	<i>c</i>	<i>d</i>	\$	<i>S</i>	<i>C</i>
0	s36	s47		1	2
1			acc		
2	s36	s47		5	
36	s36	s47		89	
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		



- To see how the GOTO's are computed, consider  $\text{GOTO}(I_{36}, C)$ .
- In the original set of LR(1) items,  $\text{GOTO}(I_3, C) = I_8$ , and  $I_8$  is now part of  $I_{89}$ , so we make  $\text{GOTO}(I_{36}, C)$  be  $I_{89}$ .

STATE	ACTION			GOTO	
	<i>c</i>	<i>d</i>	\$	<i>S</i>	<i>C</i>
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		

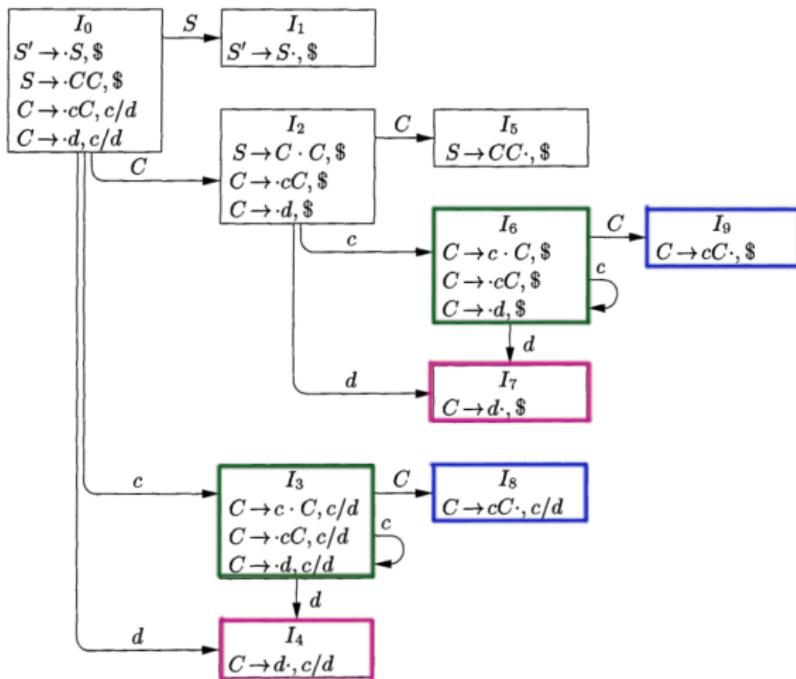
- To see how the GOTO's are computed, consider  $\text{GOTO}(I_{36}, C)$ .
- In the original set of LR(1) items,  $\text{GOTO}(I_3, C) = I_8$ , and  $I_8$  is now part of  $I_{89}$ , so we make  $\text{GOTO}(I_{36}, C)$  be  $I_{89}$ .



- We could have arrived at the same conclusion if we considered  $I_6$ , the other part of  $I_{36}$ .
- That is,  $\text{GOTO}(I_6, C) = I_9$ , and  $I_9$  is now part of  $I_{89}$ .

STATE	ACTION			GOTO	
	<i>c</i>	<i>d</i>	\$	<i>S</i>	<i>C</i>
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		

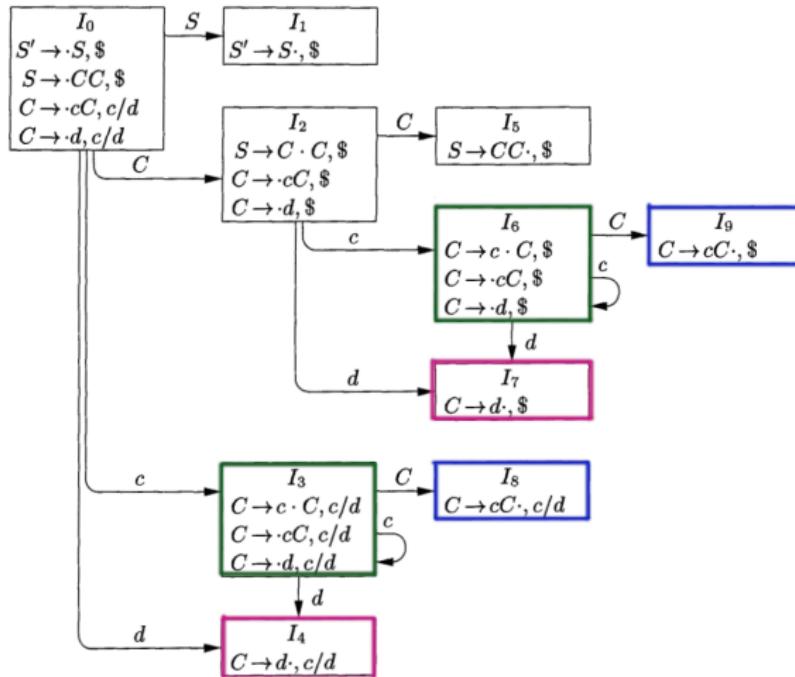
- We could have arrived at the same conclusion if we considered  $I_6$ , the other part of  $I_{36}$ .
- That is,  $\text{GOTO}(I_6, C) = I_9$ , and  $I_9$  is now part of  $I_{89}$ .



- For another example, consider  $\text{GOTO}(I_2, c)$ , an entry that is exercised after the shift action of  $I_2$  on input  $c$ .

STATE	ACTION			GOTO	
	<i>c</i>	<i>d</i>	\$	<i>S</i>	<i>C</i>
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		

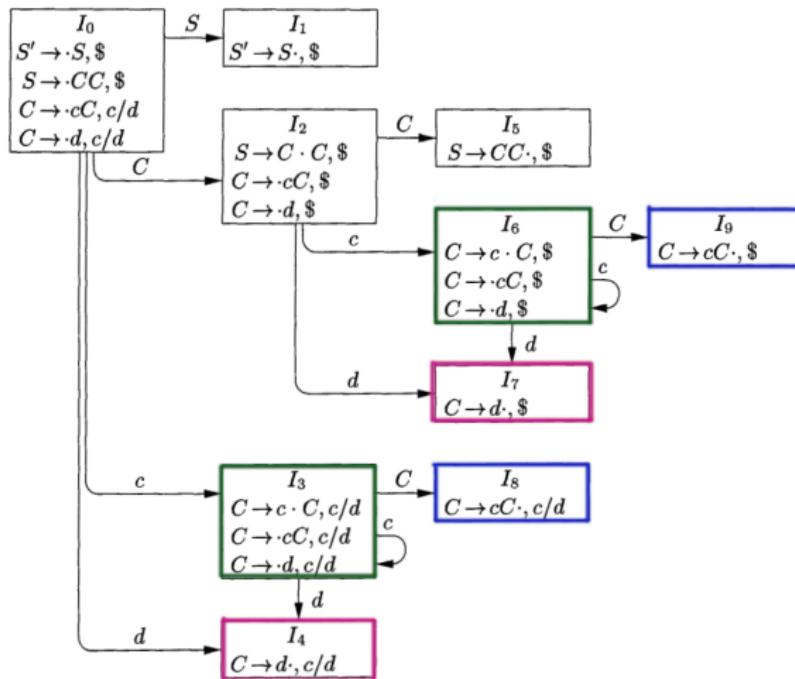
- For another example, consider  $\text{GOTO}(I_2, c)$ , an entry that is exercised after the shift action of  $I_2$  on input  $c$ .



- In the original sets of LR(1) items,  $\text{GOTO}(I_2, c) = I_6$ .
- Since  $I_6$  is now part of  $I_{36}$ ,  $\text{GOTO}(I_2, c)$  becomes  $I_{36}$ .

STATE	ACTION			GOTO	
	<i>c</i>	<i>d</i>	\$	<i>S</i>	<i>C</i>
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		

- In the original sets of LR(1) items,  $\text{GOTO}(I_2, c) = I_6$ .
- Since  $I_6$  is now part of  $I_{36}$ ,  $\text{GOTO}(I_2, c)$  becomes  $I_{36}$ .



- Thus, the entry for state 2 and input  $c$  is made s36, meaning shift and push state 36 onto the stack.

STATE	ACTION			GOTO	
	<i>c</i>	<i>d</i>	\$	<i>S</i>	<i>C</i>
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		

- Thus, the entry for state 2 and input *c* is made s36, meaning shift and push state 36 onto the stack.

## Constructing LALR Parsing Tables — *continued*

- When presented with a string from the language  $c^*dc^*d$ , both the LR parser and the LALR parser make exactly the same sequence of shifts and reductions.
- Although the names of the states on the stack may differ.
- For instance, if the LR parser puts  $I_3$  or  $I_6$  on the stack, the LALR parser will put  $I_{36}$  on the stack.
- This relationship holds in general for an LALR grammar.
- The LR and LALR parsers will mimic one another on correct inputs.

# Constructing LALR Parsing Tables — *continued*

- When presented with erroneous input, the LALR parser may proceed to do some reductions after the LR parser has declared an error.
- However, the LALR parser will never shift another symbol after the LR parser declares an error.

## Constructing LALR Parsing Tables — *continued*

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		

- For example, on input *ccd* followed by \$, the LR parser will put, 0 3 3 4, on the stack.
- And in state 4 will discover an error, because \$ is the next input symbol and state 4 has action error on \$.

# Constructing LALR Parsing Tables — *continued*

STATE	ACTION			GOTO	
	<i>c</i>	<i>d</i>	\$	<i>S</i>	<i>C</i>
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		

- (1)  $S \rightarrow C C$
- (2)  $C \rightarrow cC$
- (3)  $C \rightarrow d$

- In contrast, the LALR parser will make the corresponding moves, putting, 0 36 36 47, on the stack.
- But state 47 on input \$ has action reduce  $C \rightarrow d$ .

# Constructing LALR Parsing Tables — *continued*

STATE	ACTION			GOTO	
	<i>c</i>	<i>d</i>	\$	<i>S</i>	<i>C</i>
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		

- (1)  $S \rightarrow C C$
- (2)  $C \rightarrow cC$
- (3)  $C \rightarrow d$

- The LALR parser will thus change its stack to, 0 36 36 89.
- Now the action of state 89 on input \$ is reduce  $C \rightarrow cC$ .

# Constructing LALR Parsing Tables — *continued*

STATE	ACTION			GOTO	
	<i>c</i>	<i>d</i>	\$	<i>S</i>	<i>C</i>
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		

- (1)  $S \rightarrow C C$
- (2)  $C \rightarrow cC$
- (3)  $C \rightarrow d$

- The stack becomes, 0 36 89.
- Whereupon a similar reduction is called for, obtaining stack, 0 2.

# Constructing LALR Parsing Tables — *continued*

STATE	ACTION			GOTO	
	<i>c</i>	<i>d</i>	\$	<i>S</i>	<i>C</i>
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		

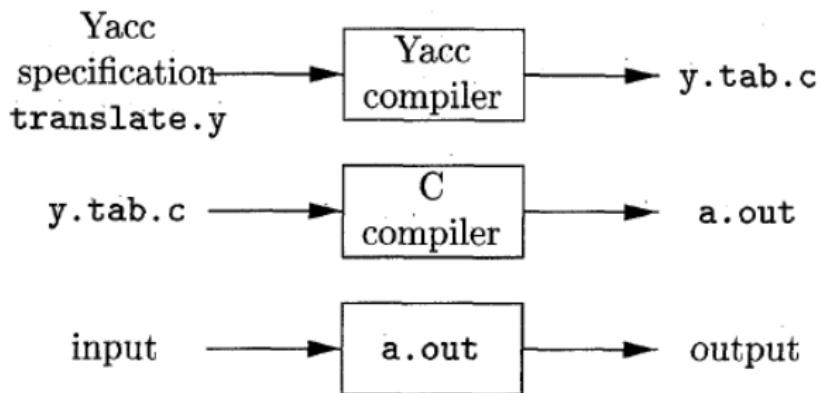
- (1)  $S \rightarrow C C$
- (2)  $C \rightarrow cC$
- (3)  $C \rightarrow d$

- Finally, state 2 has action error on input \$, so the error is now discovered.

- We show how a parser generator can be used to facilitate the construction of the front end of a compiler.
- `Yacc` stands for “yet another compiler-compiler,” reflecting the popularity of parser generators in the early 1970s when the first version of `Yacc` was created by S. C. Johnson.
- `Yacc` is available as a command on the UNIX system, and has been used to help implement many production compilers.

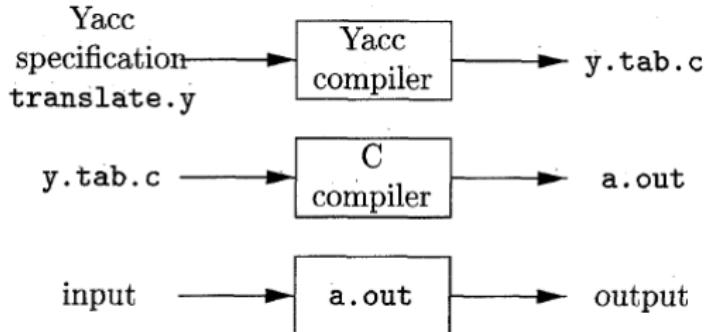
# The Parser Generator Yacc

- A translator can be constructed using Yacc in the manner illustrated in figure.



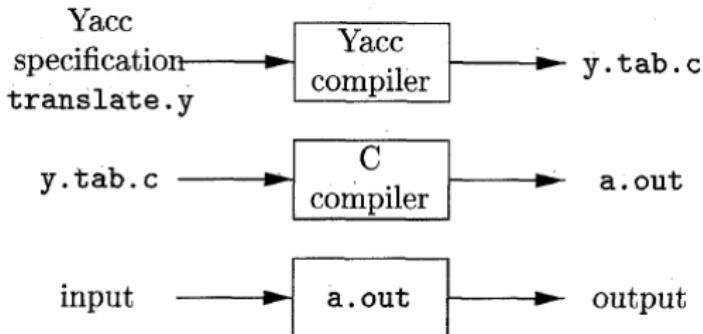
Creating an input/output translator with Yacc

# The Parser Generator Yacc — *continued*



- First, a file, say `translate.y`, containing a Yacc specification of the translator is prepared.
- The UNIX system command `Yacc translate.y` transforms the file `translate.y` into a C program called `y.tab.c`.
- The program `y.tab.c` is a representation of a parser written in C, along with other C routines that the user may have prepared.

# The Parser Generator Yacc — *continued*



- By compiling `y.tab.c` along with the `ly` library that contains the LR parsing program using the command we obtain the desired object program `a.out` that performs the translation specified by the original `Yacc` program.
- If other procedures are needed, they can be compiled or loaded with `y.tab.c`, just as with any C program.

- A Yacc source program has three parts:

declarations

%%

translation rules

%%

supporting C routines

# Example

- To illustrate how to prepare a Yacc source program, let us construct a simple desk calculator that reads an arithmetic expression, evaluates it, and then prints its numeric value.
- We shall build the desk calculator starting with the following grammar for arithmetic expressions:

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

- The token **digit** is a single digit between 0 and 9.

## Example — *continued*

- A Yacc desk calculator program derived from this grammar is shown in figure.

```
%{
#include <ctype.h>
%}

%token DIGIT

%%

line  : expr '\n'      { printf("%d\n", $1); }
;
expr  : expr '+' term  { $$ = $1 + $3; }
| term
;
term  : term '*' factor { $$ = $1 * $3; }
| factor
;
factor: '(' expr ')'
| DIGIT
;
%%

yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}

Yacc specification of a simple desk calculator
```

# The Parser Generator Yacc — *continued*

## The Declarations Part

- There are two sections in the declarations part of a Yacc program.
- Both are optional.
- In the first section, we put ordinary C declarations, delimited by %{ and %}.
- Here we place declarations of any temporaries used by the translation rules or procedures of the second and third sections.

```
%{  
#include <ctype.h>  
%}  
  
%token DIGIT
```

# The Parser Generator Yacc — *continued*

## The Declarations Part

- This section contains only the include-statement

#include <ctype.h>  
that causes the C preprocessor to  
include the standard header file  
<ctype.h> that contains the predicate  
isdigit.

```
%{  
#include <ctype.h>  
%}  
  
%token DIGIT
```

# The Parser Generator Yacc — *continued*

## The Declarations Part

- Also in the declarations part are declarations of grammar tokens.
- The statement

```
%token DIGIT
```

declares DIGIT to be a token.

- Tokens declared in this section can then be used in the second and third parts of the Yacc specification.

- If Lex is used to create the lexical analyzer that passes token to the Yacc parser, then these token declarations are also made available to the analyzer generated by Lex.

```
%{  
#include <ctype.h>  
%}  
  
%token DIGIT
```

# The Parser Generator Yacc — *continued*

## The Translation Rules Part

```
%%
line  : expr '\n'      { printf("%d\n", $1); }
      ;
expr  : expr '+' term { $$ = $1 + $3; }
      | term
      ;
term  : term '*' factor { $$ = $1 * $3; }
      | factor
      ;
factor : '(' expr ')'
      | DIGIT
      ;
```

- In the part of the Yacc specification after the first `%%` pair, we put the translation rules.
- Each rule consists of a grammar production and the associated semantic action.

# The Parser Generator Yacc — *continued*

## The Translation Rules Part

- A set of productions that we have been writing:

$$<\text{head}> \rightarrow <\text{body}>_1 \mid <\text{body}>_2 \mid \dots \mid <\text{body}>_n$$

would be written in Yacc as

```
<head> : <body>1      { <semantic action>1 }
          | <body>2      { <semantic action>2 }
          ...
          | <body>n      { <semantic action>n }
          ;
```

# The Parser Generator Yacc — *continued*

## The Translation Rules Part

- In a Yacc production, unquoted strings of letters and digits not declared to be tokens are taken to be nonterminals.
- A quoted single character, e.g. 'c', is taken to be the terminal symbol  $c$ , as well as the integer code for the token represented by that character (i.e., Lex would return the character code for 'c' to the parser, as an integer).
- Alternative bodies can be separated by a vertical bar, and a semicolon follows each head with its alternatives and their semantic actions.
- The first head is taken to be the start symbol.

```
%%
line   : expr '\n'          { printf("%d\n", $1); }
;
expr   : expr '+' term    { $$ = $1 + $3; }
| term
;
term   : term '*' factor  { $$ = $1 * $3; }
| factor
;
factor : '(' expr ')',   { $$ = $2; }
| DIGIT
;
```

- A Yacc semantic action is a sequence of C statements.
- In a semantic action, the symbol `$$` refers to the attribute value associated with the nonterminal of the head.

```

%%

line   : expr '\n'          { printf("%d\n", $1); }

;

expr   : expr '+' term    { $$ = $1 + $3; }

| term

;

term   : term '*' factor  { $$ = $1 * $3; }

| factor

;

factor : '(' expr ')',   { $$ = $2; }

| DIGIT

;

```

- While  $\$i$  refers to the value associated with the  $i$ th grammar symbol (terminal or nonterminal) of the body.
- The semantic action is performed whenever we reduce by the associated production, so normally the semantic action computes a value for  $\$\$$  in terms of the  $\$i$ 's.

# The Parser Generator Yacc — *continued*

## The Translation Rules Part

- In the Yacc specification, we have written the two  $E$ -productions and their associated semantic actions.

$$\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow ( E ) \mid \text{digit} \end{array}$$

```
%%
line   : expr '\n'          { printf("%d\n", $1); }
;
expr   : expr '+' term    { $$ = $1 + $3; }
       | term
;
term   : term '*' factor  { $$ = $1 * $3; }
       | factor
;
factor : '(' expr ')'     { $$ = $2; }
       | DIGIT
;
```

$$\begin{array}{l}
 E \rightarrow E + T \mid T \\
 T \rightarrow T * F \mid F \\
 F \rightarrow ( E ) \mid \text{digit}
 \end{array}$$

```
%%
line  : expr '\n'           { printf("%d\n", $1); }
;
expr  : expr '+' term     { $$ = $1 + $3; }
      | term
;
term  : term '*' factor  { $$ = $1 * $3; }
      | factor
;
factor : '(' expr ')'
      | DIGIT
;
```

- Note that the nonterminal `term` in the first production is the third grammar symbol of the body, while `+` is the second.

$$\begin{array}{lcl}
 E & \rightarrow & E + T \mid T \\
 T & \rightarrow & T * F \mid F \\
 F & \rightarrow & ( E ) \mid \text{digit}
 \end{array}$$

```
%%
line  : expr '\n'           { printf("%d\n", $1); }
;
expr  : expr '+' term     { $$ = $1 + $3; }
| term
;
term  : term '*' factor  { $$ = $1 * $3; }
| factor
;
factor : '(' expr ')'    { $$ = $2; }
| DIGIT
;
```

- The semantic action associated with the first production adds the value of the `expr` and the `term` of the body and assigns the result as the value for the nonterminal `expr` of the head.

$$\begin{array}{l}
 E \rightarrow E + T \mid T \\
 T \rightarrow T * F \mid F \\
 F \rightarrow ( E ) \mid \text{digit}
 \end{array}$$

```
%%
line  : expr '\n'           { printf("%d\n", $1); }
;
expr  : expr '+' term     { $$ = $1 + $3; }
| term
;
term  : term '*' factor  { $$ = $1 * $3; }
| factor
;
factor: '(' expr ')'     { $$ = $2; }
| DIGIT
;
```

- We have omitted the semantic action for the second production altogether, since copying the value is the default action for productions with a single grammar symbol in the body.
- In general, `{ $$ = $1; }` is the default semantic action.

```
%%
line  : expr '\n'          { printf("%d\n", $1); }
;
expr  : expr '+' term    { $$ = $1 + $3; }
| term
;
term  : term '*' factor { $$ = $1 * $3; }
| factor
;
factor : '(' expr ')'
| DIGIT
;
```

- Notice that we have added a new starting production

```
line : expr '\n' { printf ("\%d\n", $1); }
```

to the Yacc specification.

- This production says that an input to the desk calculator is to be an expression followed by a newline character.
- The semantic action associated with this production prints the decimal value of the expression followed by a newline character.

# The Parser Generator Yacc — *continued*

## The Supporting C-Routines Part

```
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c - '0';
        return DIGIT;
    }
    return c;
}
```

- The third part of a Yacc specification consists of supporting C-routines.

# The Parser Generator Yacc — *continued*

## The Supporting C-Routines Part

```
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

- A lexical analyzer by the name `yylex()` must be provided.
- Using Lex to produce `yylex()` is a common choice.
- Other procedures such as error recovery routines may be added as necessary.

# The Parser Generator Yacc — *continued*

## The Supporting C-Routines Part

```
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

- The lexical analyzer `yylex()` produces tokens consisting of a token name and its associated attribute value.
- If a token name such as `DIGIT` is returned, the token name must be declared in the first section of the Yacc specification.
- The attribute value associated with a token is communicated to the parser through a Yacc-defined variable `yylval`.

# The Parser Generator Yacc — *continued*

## The Supporting C-Routines Part

```
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

- The lexical analyzer here is very crude.
- It reads input characters one at a time using the C-function `getchar()`.
- If the character is a digit, the value of the digit is stored in the variable `yylval`, and the token name `DIGIT` is returned.
- Otherwise, the character itself is returned as the token name.

# Creating Yacc Lexical Analyzers with Lex

- Lex was designed to produce lexical analyzers that could be used with Yacc.
- The Lex library `ll` will provide a driver program named `yylex()`, the name required by Yacc for its lexical analyzer.
- If Lex is used to produce the lexical analyzer, we replace the routine `yylex()` in the third part of the Yacc specification by the statement `#include "lex.yy.c"` and we have each Lex action return a terminal known to Yacc.
- By using the `#include "lex.yy.c"` statement, the program `yylex` has access to Yacc's names for tokens, since the Lex output file is compiled as part of the Yacc output file `y.tab.c`.

# Creating Yacc Lexical Analyzers with Lex

- Lex was designed to produce lexical analyzers that could be used with Yacc.
- The Lex library `ll` will provide a driver program named `yylex()`, the name required by Yacc for its lexical analyzer.
- If Lex is used to produce the lexical analyzer, we replace the routine `yylex()` in the third part of the Yacc specification by the statement `#include "lex.yy.c"` and we have each Lex action return a terminal known to Yacc.
- By using the `#include "lex.yy.c"` statement, the program `yylex` has access to Yacc's names for tokens, since the Lex output file is compiled as part of the Yacc output file `y.tab.c`.

# Creating Yacc Lexical Analyzers with Lex — *continued*

- Under the UNIX system, if the Lex specification is in the file `first.l` and the Yacc specification in `second.y`, we can say

```
lex first.l
```

```
yacc second.y
```

```
cc y.tab.c -ly -ll
```

to obtain the desired translator.

# Creating Yacc Lexical Analyzers with Lex — continued

```
number    [0-9]+\.?|[0-9]*\. [0-9]+  
%%  
[ ]      { /* skip blanks */ }  
{number} { sscanf(yytext, "%lf", &yyval);  
          return NUMBER; }  
\n|.    { return yytext[0]; }
```

Lex specification for yylex()

- The Lex specification in figure can be used in place of the lexical analyzer for Yacc.
- The last pattern, meaning “any character,” must be written `\n|.`  since the dot in Lex matches any character except newline.



# End of Slides