



Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Sanket N D

Compiler Design

Unit 2

Parsers and Error Recovery Strategies

Preet Kanwal

Department of Computer Science & Engineering

Compiler Design

Lecture Overview



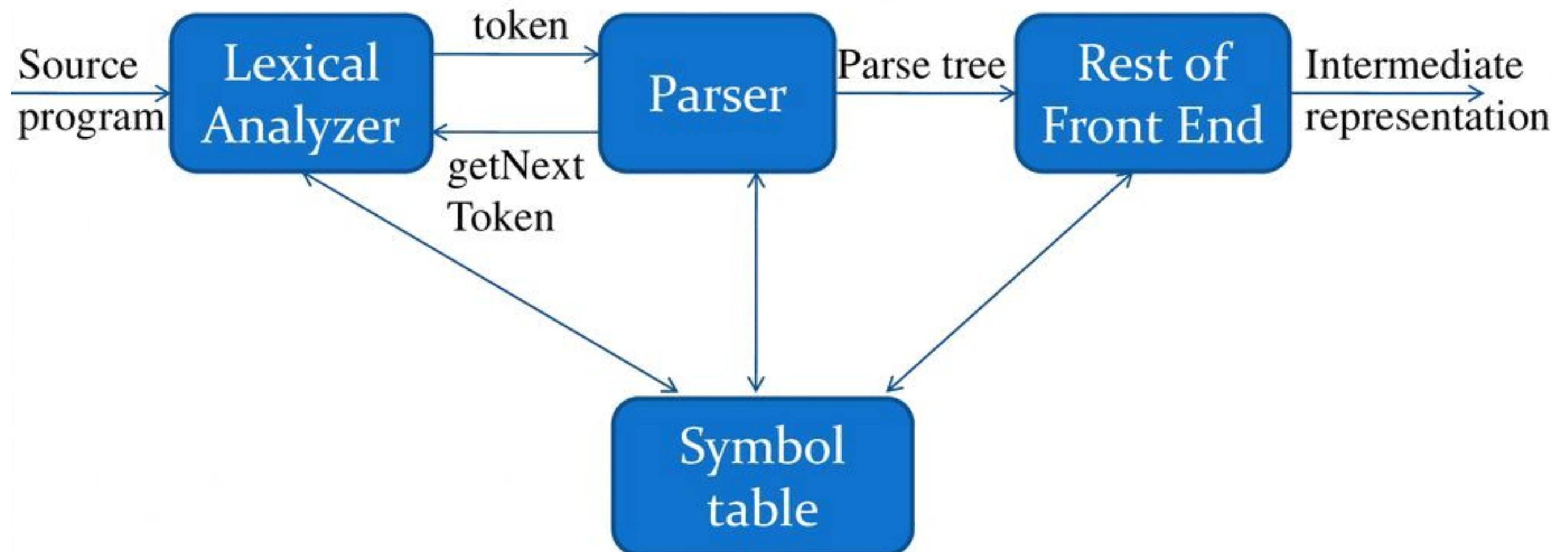
In this lecture, you will learn about -

- **What are parsers?**
- **Role of a Parser**
- **Error Recovery Strategies in Parsers**
- **Overview of Types of Parsers**
 - **Top-down and Bottom-Up Parsers**

- Parsing is the process of determining of a string of tokens can be generated by the grammar or not.
- Parser, also called a Syntax Analyzer, is a program which takes as input a grammar G and a string w to produce a parse tree for w , if the string w belongs to the language of grammar G else throws an error.
- Note - Input is read from left to right.

Compiler Design

Role of a Parser



Compiler Design

Role of a Parser



The Role of a parser is-

- **To validate the syntax of the program.**
- **To construct a parse tree and pass it to the semantic analyzer in the compiler.**
- **To store information in the symbol table about tokens received from the lexer.**
- **Report syntax errors.**

The compiler must report errors by generating messages of the following format -

- The message must pinpoint the error - for example, the line number of the statement containing the error.
- The message should be specific and must localise the problem.
 - Eg: Undeclared variable in function foo() at line 29.
- The message should be clear and understandable.
- Messages should not be redundant. For example, if an undeclared variable is referenced 20 times, the error should be displayed only once.

Error Recovery Strategies

- There is no universally accepted strategy for error recovery in Parsers.
- Some simple strategies would be to quit with an informative error message (eg. python), or gather additional errors before quitting (eg. C), presuming it is possible to restore the parser state where it can continue processing.
- There are four commonly used error recovery strategies -
 1. Panic Mode Recovery
 2. Phrase level Recovery
 3. Error Productions
 4. Global Corrections

1. Panic Mode Recovery

- The parser will discard the input symbols until it finds a delimiter like ; or) or etc, and then restart the parsing process.
- Such delimiters are called **synchronizing tokens**.
- Consider the following expression -

(1 + + 2) + 3

In this case, we skip the input till) and then continue parsing.

2. Phrase level recovery

- This involves performing local corrections in the input program, in case it contains errors.
- For example,
 - missing semicolon ; - insert it
 - missing closing bracket) - insert it
- The choice of the correction is left to the compiler designer. However, it needs to be done carefully.
 - Incorrectly inserting characters could cause the program to perform erroneously.

3. Error productions

- The idea here is to specify in the grammar known common mistakes.

For example,

- $S \rightarrow \text{if (cond) } \{S\} \text{ else } \{S\}$

is grammar for valid if-else code.

- $S \rightarrow \text{fi (cond) } \{S\} \text{ else } \{S\}$

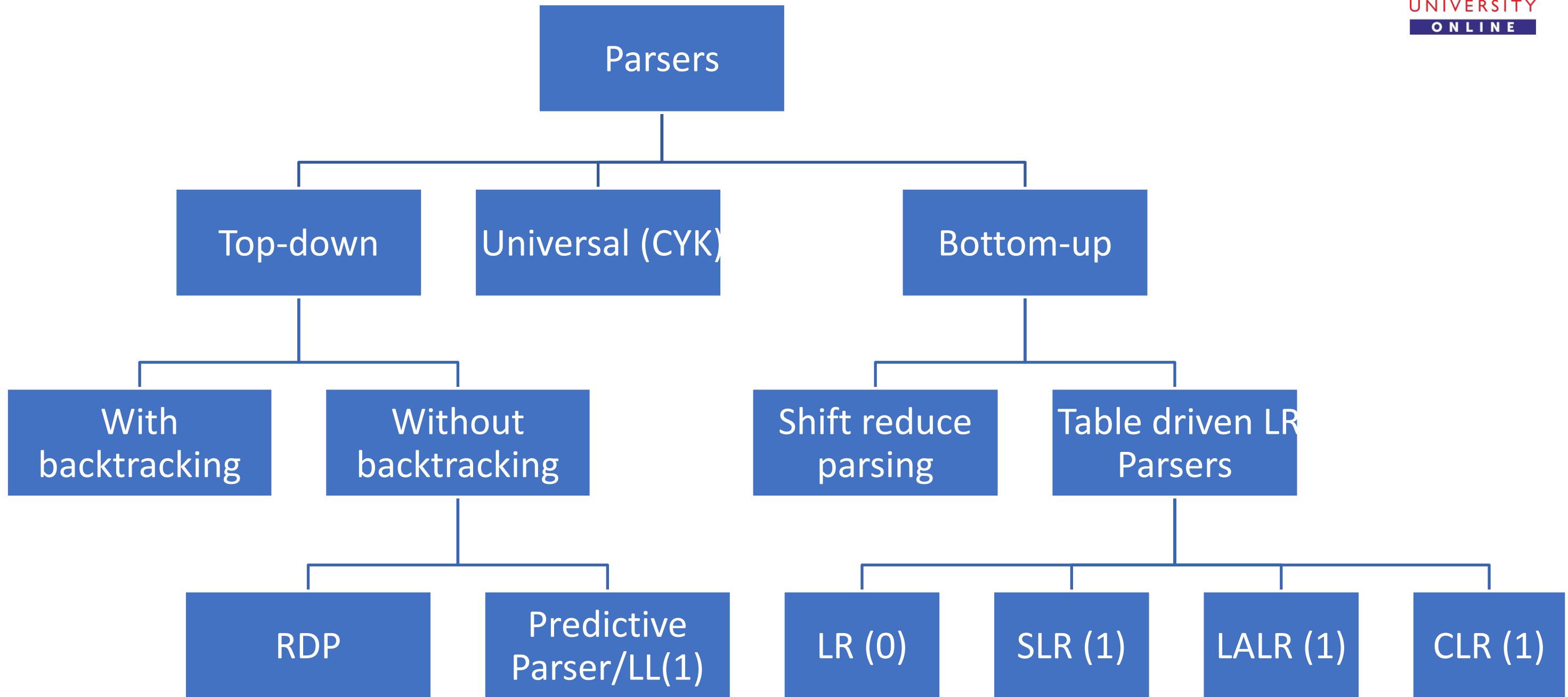
can be written as an error production, with action being a more informative message about the error to the user.

- Disadvantage - It complicates the grammar.

4. Global Correction

- Extremely hard to implement
- Takes the grammar and the input program (with errors) as input, and passes them to the algorithm to find a program y that is closest correct program to the original program x.
- The algorithm must have made minimal changes to x.
- Disadvantages
 - The closest correct program y may not exactly be what the programmer intended to write.

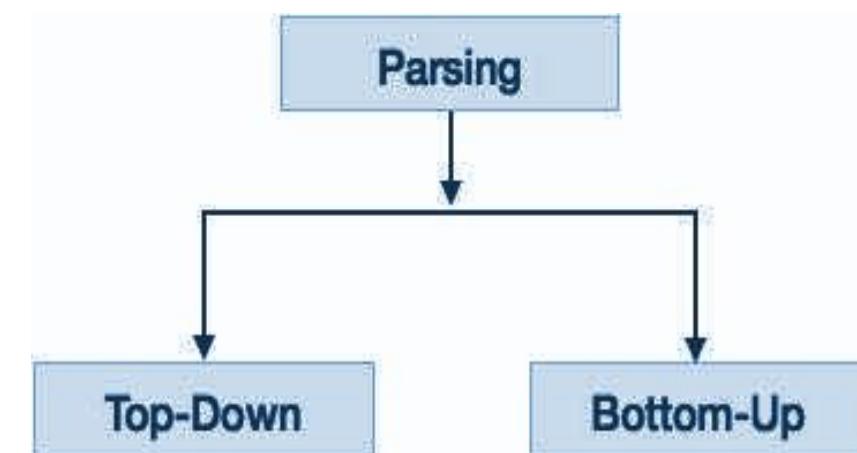
Types of Parsers



Compiler Design

Types of Parsers

- There are mainly two types of parsers - **top-down parsers** and **bottom-up parsers**.
- A **Top-down Parser** generates parse tree for the given input string with the help of grammar productions by expanding the non-terminals.
 - It starts from the start symbol and ends on the terminals.
 - It uses left most derivation.
- A **Bottom-up Parser** generates the parse tree for the given input string with the help of grammar productions by compressing the non-terminals
 - It starts from non-terminals and ends on the start symbol.
 - It uses the reverse of the rightmost derivation.





THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu



Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Sanket N D

Compiler Design

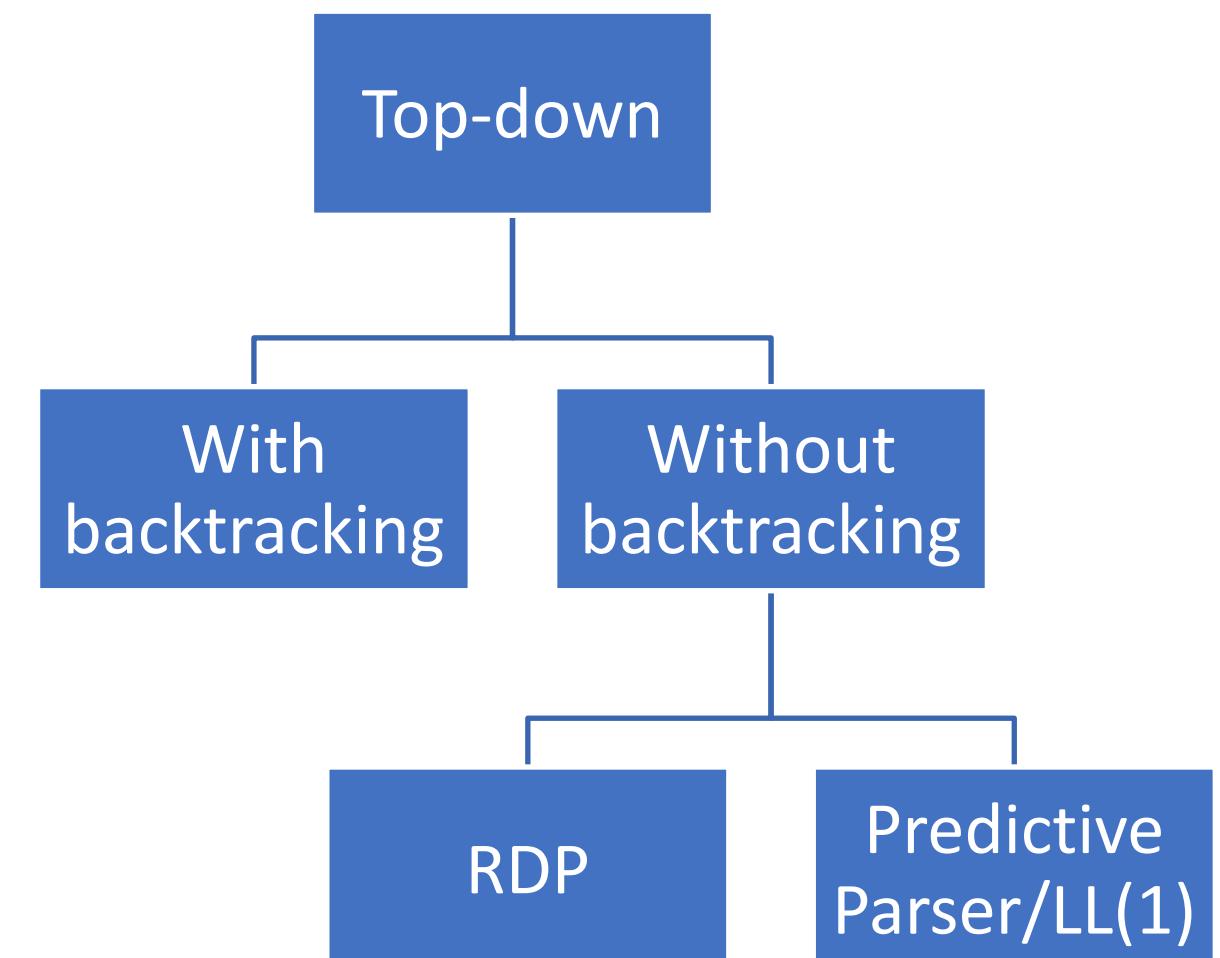
Unit 2

Recursive Descent Parsers with Backtracking

Preet Kanwal

Department of Computer Science & Engineering

- Top down parsing involves constructing a parse tree from the start symbol and attempting to transform the start symbol to the input.
- Top down parsing is based on **Left Most Derivation** of input string.
- There are two types of top-down parsing techniques:
 - With Backtracking
 - Without Backtracking



- Top-down parsers with backtracking can be implemented by using procedures.
- This implementation design is called Recursive Descent Parsing (RDP).
- In RDP, procedures are written for each non terminal in the grammar.

For example, the pseudocode below shows procedures written for the non terminals present in the following grammar -

$$\begin{array}{lcl} S & \rightarrow & c A d \\ A & \rightarrow & a b \mid a \end{array}$$

```
S() {
    if(inputSymbol++ == 'c')
    {
        if(A())
        {
            if(inputSymbol++ == 'd')
            {
                return true;
            }
        }
    }
    return false;
}
```

```
A() {
    isave = inputPointer();
    if(inputSymbol++ == 'a')
    {
        if(inputSymbol++ == 'b') {
            return true;
        }
    }
    inputSymbol = isave;
    if(inputSymbol++ == 'a') {
        return true;
    }
    return false;
}
```

Points to keep in mind while performing RDP with backtracking-

- 1. Try different alternatives of a nonterminal in the way it is listed in the grammar.**
- 2. If at least one alternative matches, the input string can be parsed.**

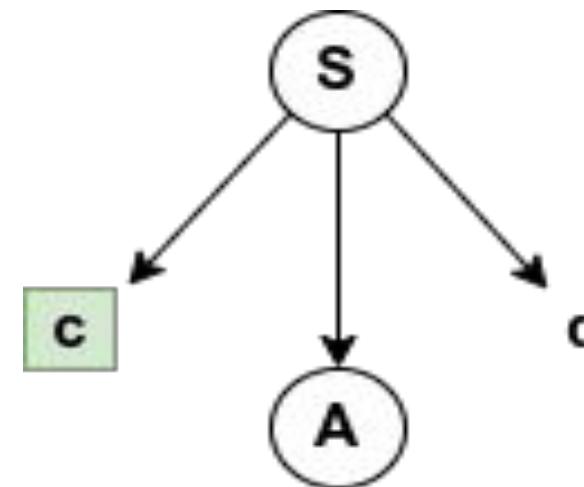
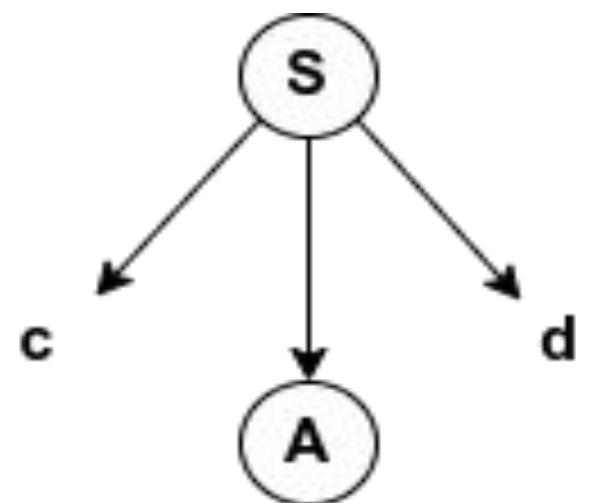
Consider the earlier grammar -

$$S \rightarrow c A d$$

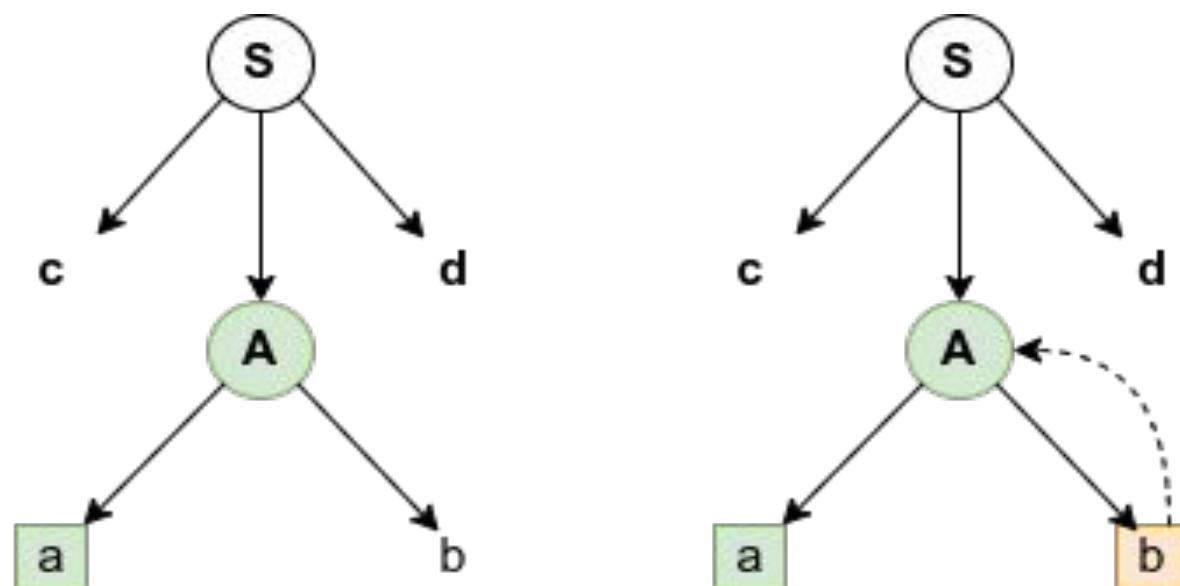
$$A \rightarrow a b \mid a$$

Let the input string be $w = \text{'cad'}$

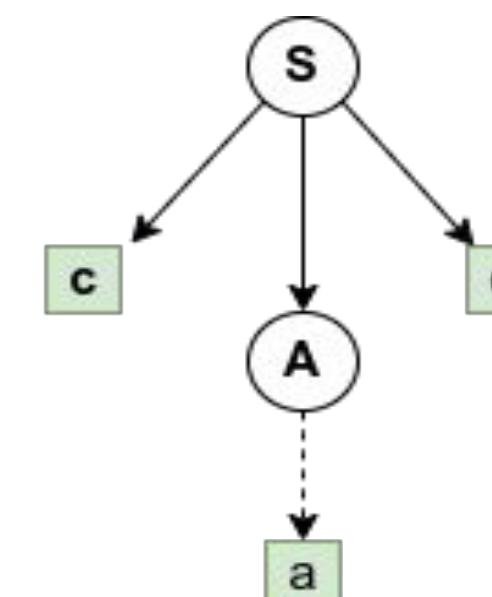
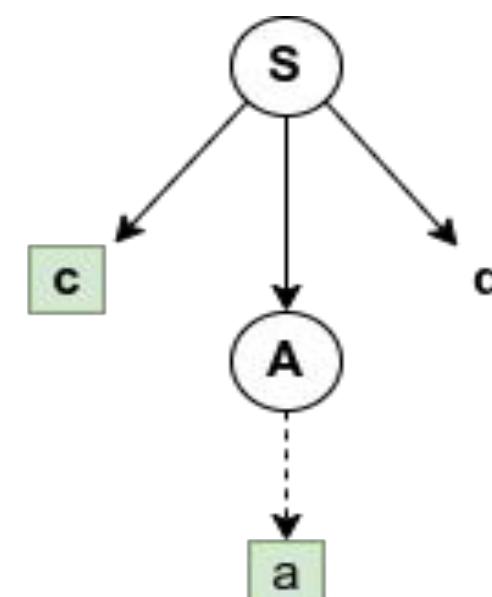
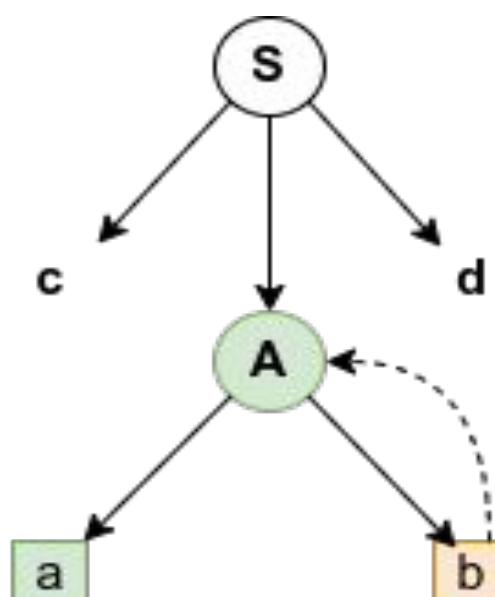
- We will start with **S**, which is the start symbol for the grammar.
- The first production of **S**, ($S \rightarrow cAd$) matches with the leftmost letter of input **c**.
- So we consider this production and continue.
- The input letter now advances to **a**.



- Now, we need to expand the non-terminal A.
- The input letter is a. The first production of A is A → ab. This matches current input letter a.
- However, since the next input letter is d and this doesn't match the current production A → ab.
- So, we backtrack to check the next production.



- The next production $A \rightarrow a$. This matches with the input letter **a**, so we accept this production and expand **A**.
- After expansion, the next input letter **d** matches the remaining part of the first production.
- We see that all the input letters are matched correctly and are also in order.
- Hence, the string is accepted by the parser.



Consider the following grammar -

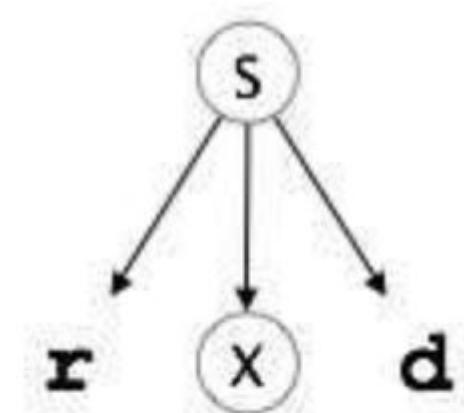
$$S \rightarrow rXd \mid rZd$$

$$X \rightarrow oa \mid ea$$

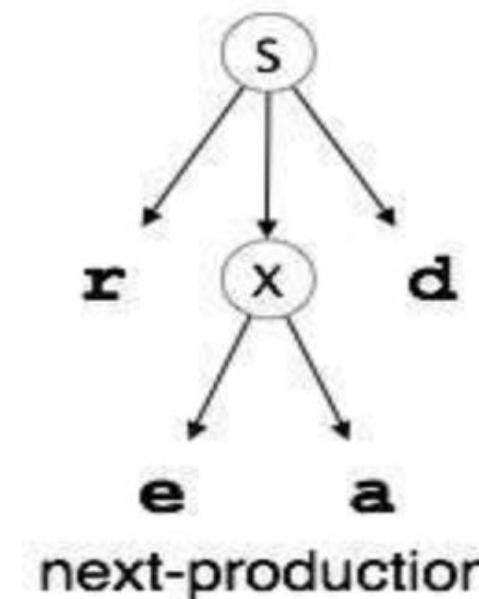
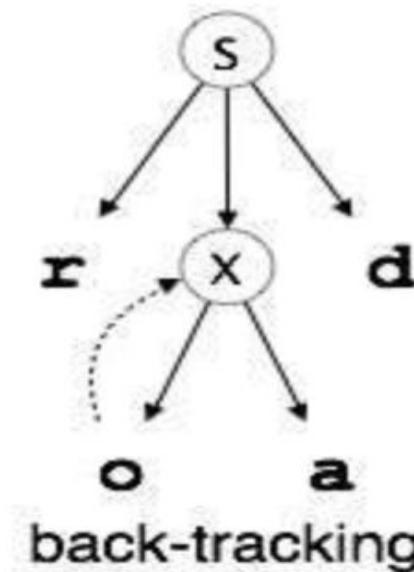
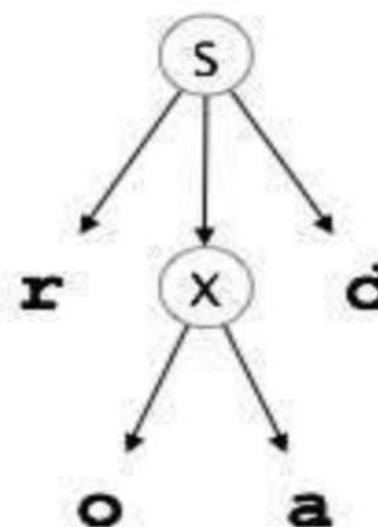
$$Z \rightarrow ai$$

Let the input string be $w = \text{'read'}$

- We will start with **S**, which is the start symbol for the grammar.
- The first production of **S**, ($S \rightarrow rXd$) matches with the left-most letter of input **r**.
- So we consider this production and continue.
- The input letter now advances to **e**



- Now, we need to expand the non-terminal X.
- The input letter is e. The first production of X is $X \rightarrow oa$. This doesn't match with the current input letter e. So, we backtrack and check the next production $X \rightarrow ea$. This matches with the input letter e, so we accept this production and expand X.
- After expansion, we see that all the input letters are matched correctly and are also in order. Hence, the string is accepted by the parser.



1. Parser program may not accept the input even if the input string belonged to the grammar because of the **order of productions.**
2. **RDP with backtracking cannot work with left recursive grammars** because it would cause the program to enter an infinite loop.
3. Reversing semantic actions during the parsing of a string using **RDP with backtracking is an overhead.**

1. Parser may not accept a valid input

- Alternative productions are tried out in the order in which they are listed in the grammar.
- This could sometimes cause the parser program to throw an error even if the input string belonged to the grammar.
- Example -
 - Consider the grammar from Example 1 -

$S \rightarrow c A d$

$A \rightarrow a b \mid a$

- Suppose the second production was changed to

$A \rightarrow a \mid a b$

- How would the input string $cabd$ be parsed?

Given -

$$S \rightarrow cAd$$

$$A \rightarrow a \mid ab$$

- Input string - cabd
- The first production of S, ($S \rightarrow cAd$) matches with the leftmost letter of input c.
- The input letter now advances to a.
- Expand the non-terminal A. The first production of A is $A \rightarrow a$.
- This matches current input letter a.

- The next input letter is **b**.
- This does not match the remaining part of the first production $S \rightarrow cAd$.
- So, we backtrack to check the next production.
- The next production $A \rightarrow ab$. This matches with the input letter **a**, so we accept this production and expand **A**.
- The next input letter **b** matches the remaining part of the production.
- The next input letter **d** matches the remaining part of the first production.
- We see that all the input letters are matched correctly and are also in order. Hence, the string is accepted by the parser.
- Notice that if the productions were unchanged, the input string would be accepted without backtracking.

2. Cannot work with Left Recursive Grammars

- Left recursive grammars would cause the program to enter an infinite loop.
- Example -
 - Consider the following grammar rule $A \rightarrow Aa \mid b$
 - Procedure $A()$ would roughly have the following pseudocode:

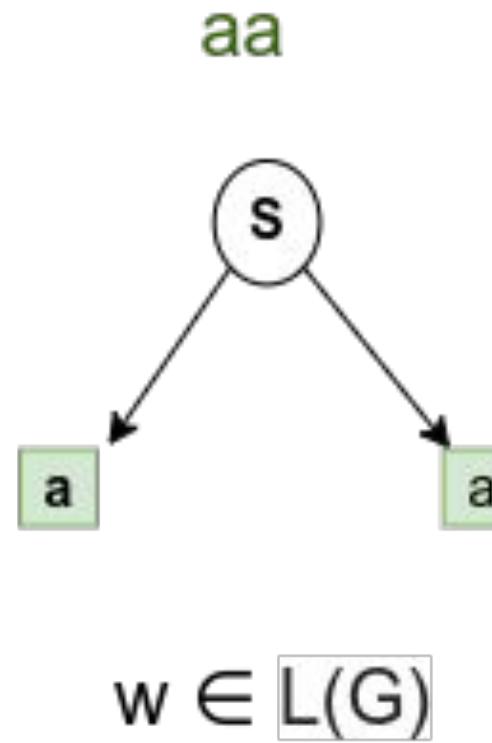
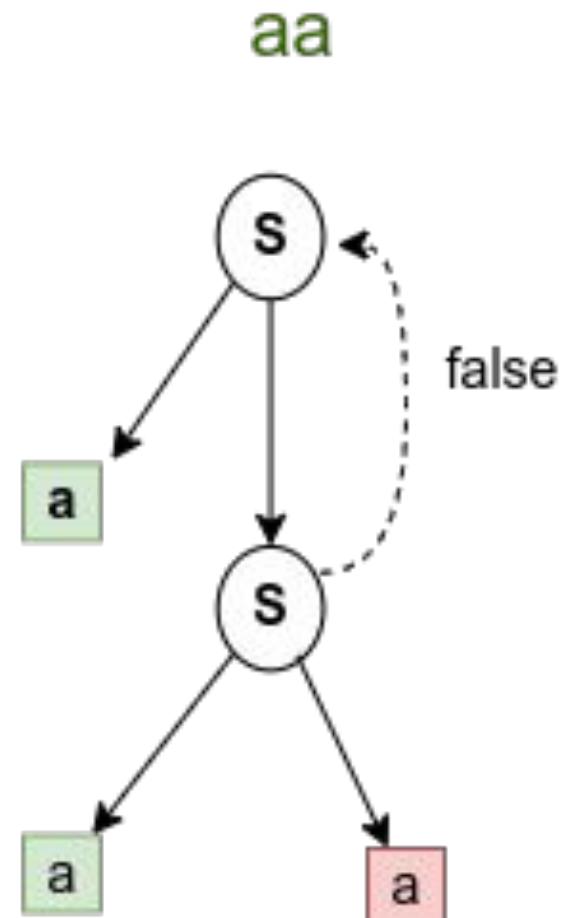
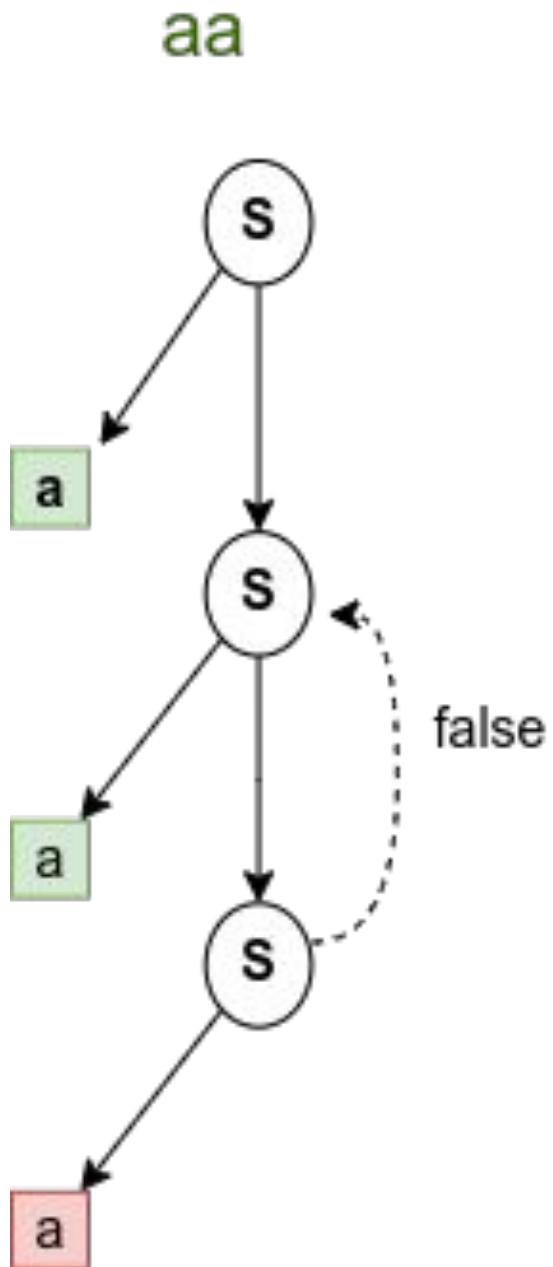
```
A() {  
    if (A()) {  
        ...  
    }  
}
```

- This causes an infinite loop.
- Thus, RDP with backtracking cannot work with left recursive grammars.

3. Overhead

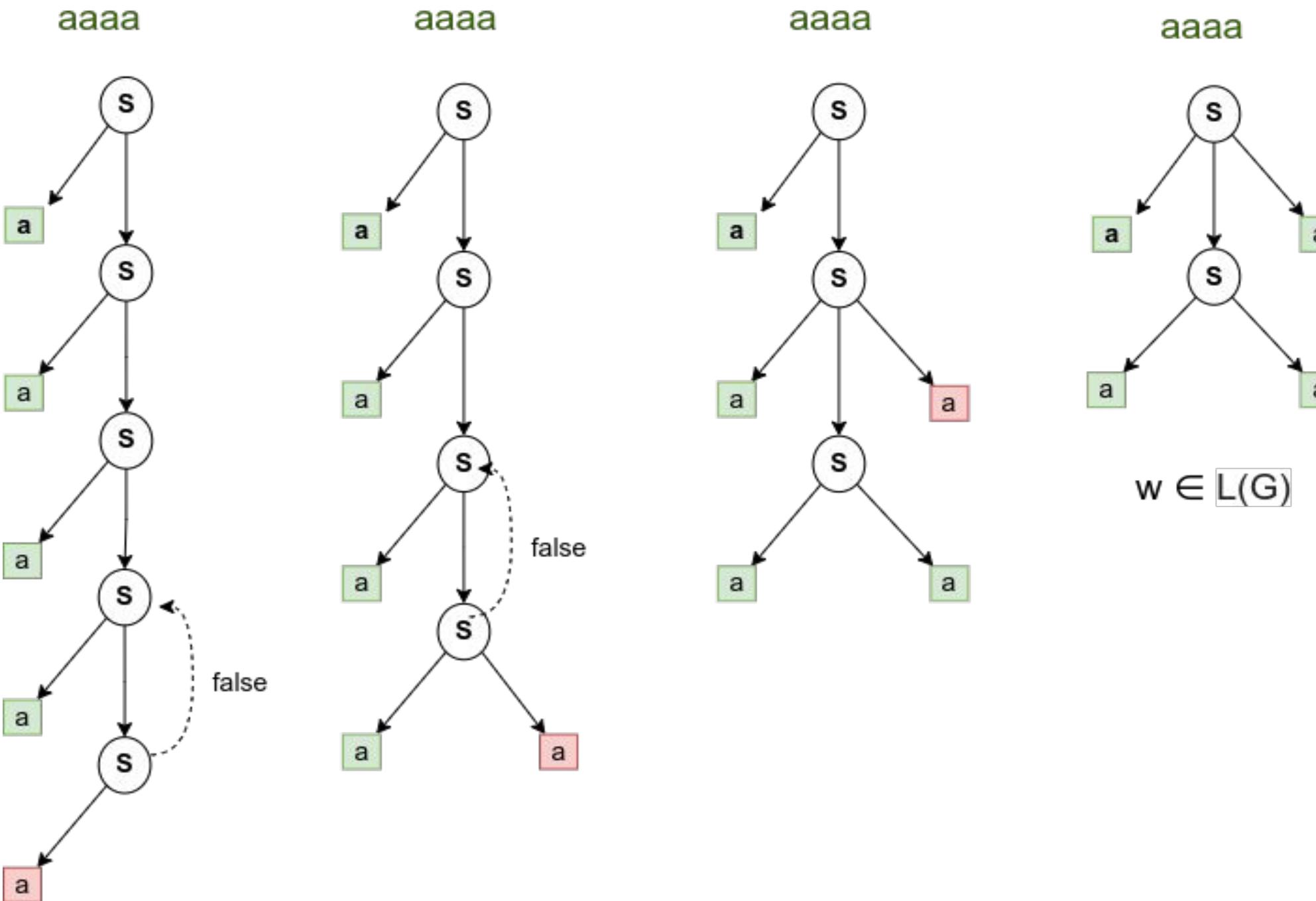
- RDP with backtracking involves a series of erroneous expansions and corresponding semantic actions.
- Reversing semantic actions during the parsing of a string causes substantial overhead.

- Consider the language defined by grammar $S \rightarrow aSa \mid aa$, which ideally accepts $L(G) = \{ a^{2n}, n \geq 1 \}$
- Show the working of RDP for the following input strings -
 - aa
 - aaaa
 - aaaaaaa
 - aaaaaaaaa
- All the strings belong to $L(G)$



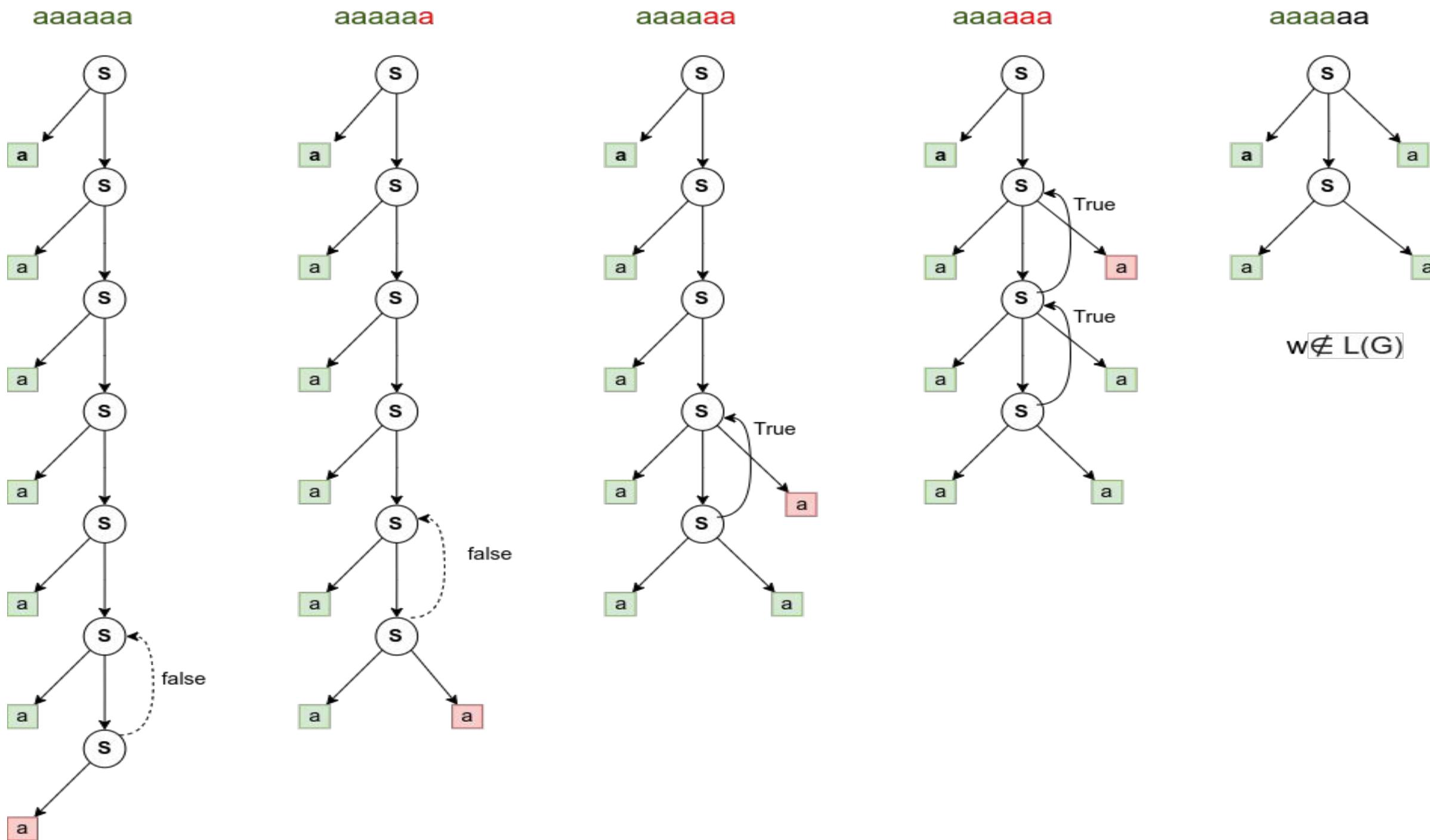
Compiler Design

Solution - aaaa

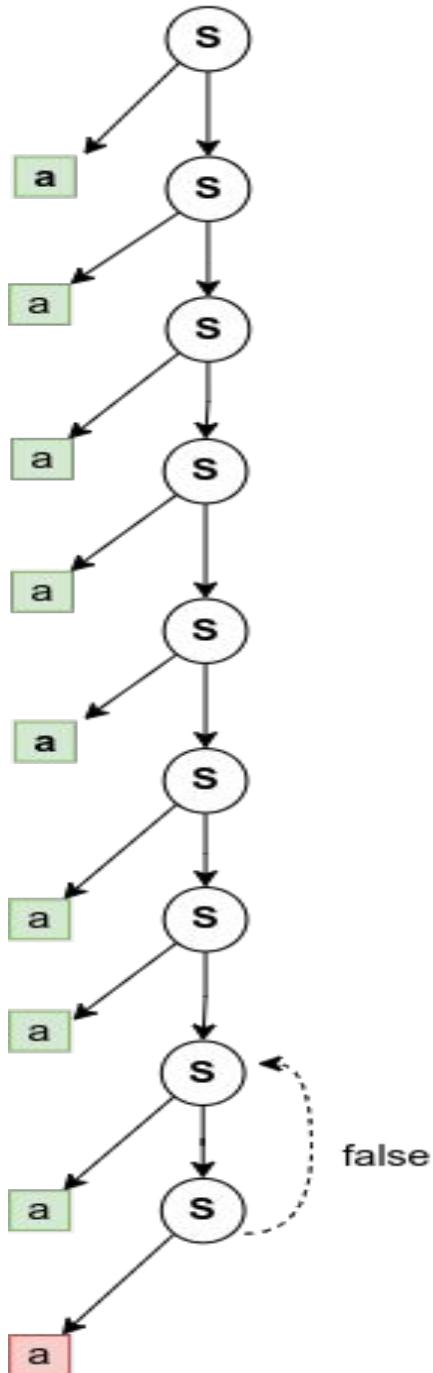


Compiler Design

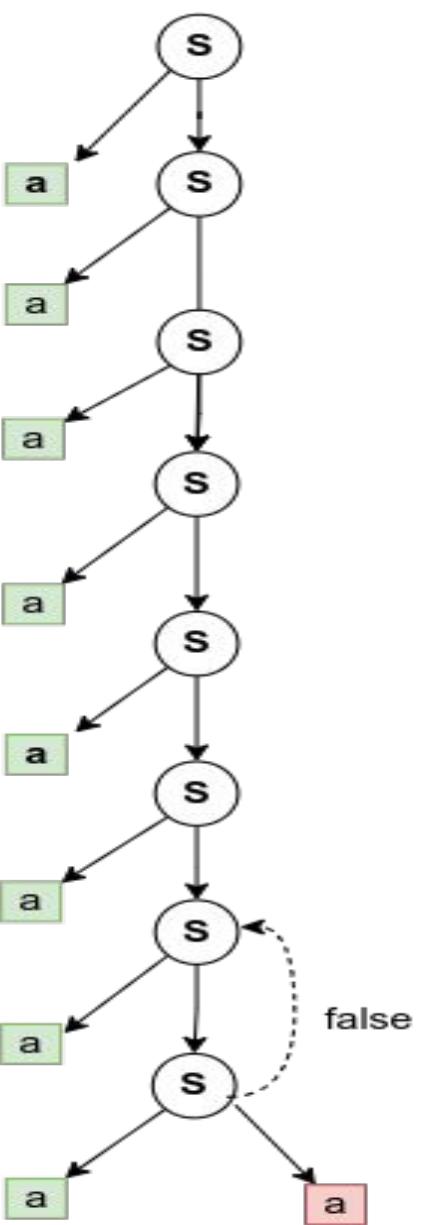
Solution - aaaaaaa



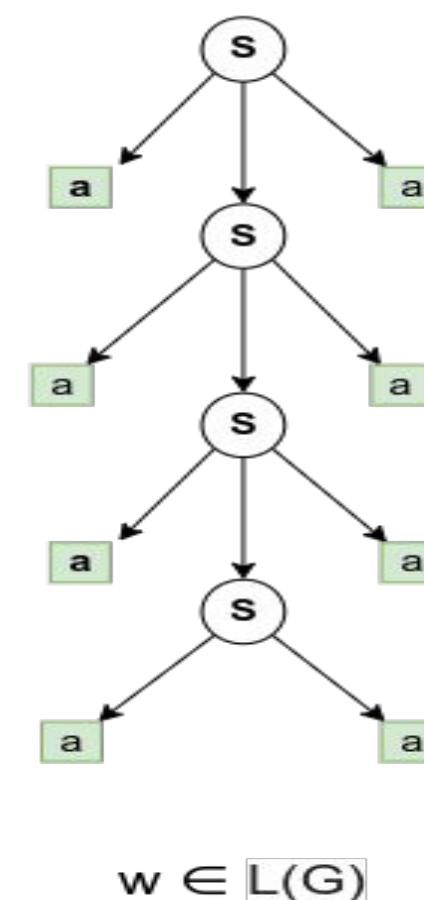
aaaaaaaaa



aaaaaaaaa



aaaaaaaaa



- The above working shows that the string aaaaaa cannot be parsed using RDP with backtracking.

Input String	RDP with Backtracking	Language
a^2	Accepts	Accepts
a^4	Accepts	Accepts
a^6	Not accepted	Accepts
a^8	Accepts	Accepts

- It is evident that only strings of the form a^{2^n} is accepted by the RDP with backtracking.



THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu



Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Kavya P K

Compiler Design

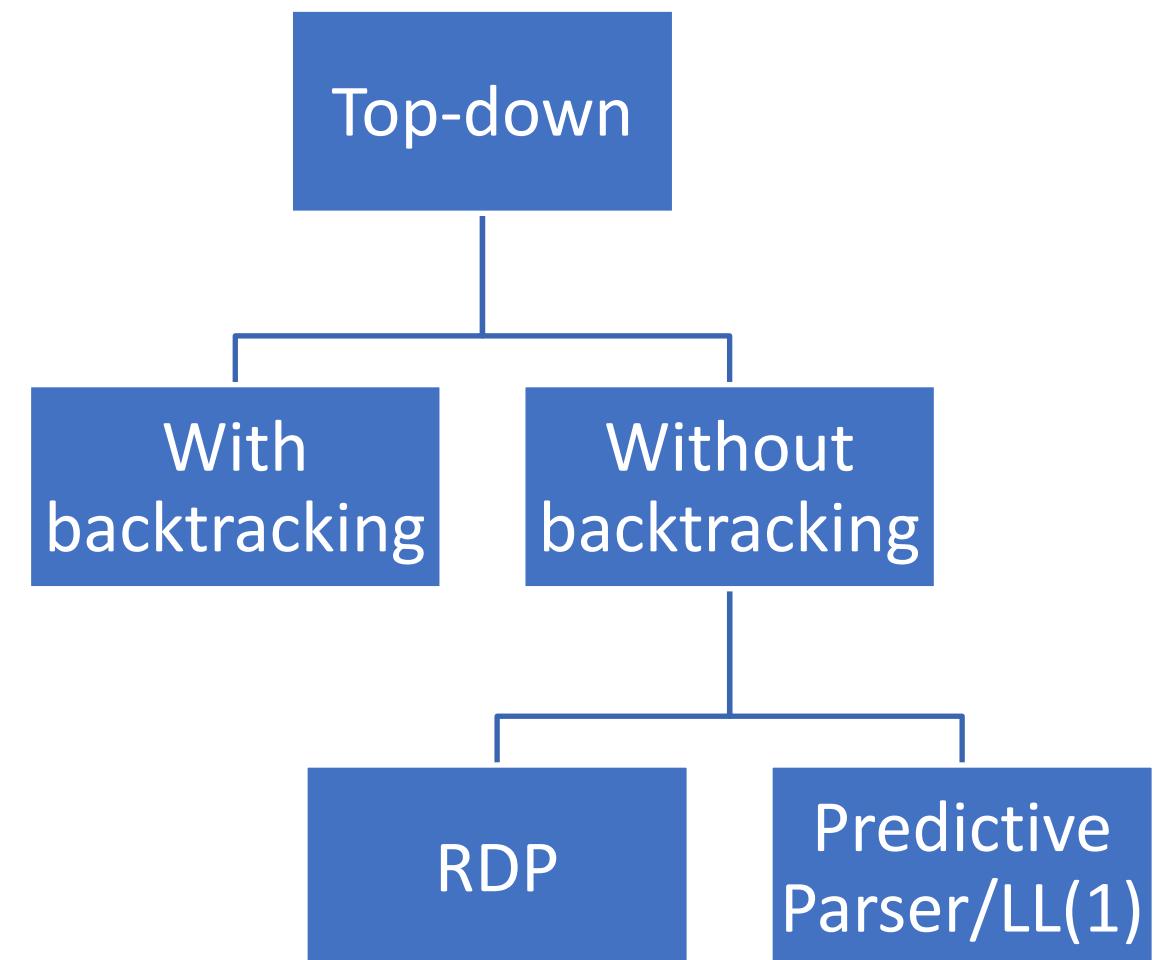
Unit 2

Recursive Descent Parsing without Backtracking

Preet Kanwal

Department of Computer Science & Engineering

- Top down parsing involves constructing a parse tree from the start symbol and attempting to transform the start symbol to the input.
- Top down parsing is based on **Left Most Derivation** of input string.
- There are two types of top-down parsing techniques:
 - With Backtracking
 - Without Backtracking
 - RDP
 - Table driven/Predictive parser



- There are two implementations of TDP without backtracking
 - RDP and table driven approach, also known as Predictive Parser or LL(1) parser.
- To deal with the drawbacks of TDP with backtracking, we introduce a few modifications to apply to grammars to make it compatible for TDP without backtracking.

- To deal with the drawbacks of TDP with backtracking, a few modifications are applied to grammars to make it compatible for TDP without backtracking.
- This involves two steps -
 - Left factoring of the grammar
 - Eliminating Left recursion
- The two operations need not be done in any particular sequence.
- Left factoring can be done before or after eliminating left recursion.
- Eliminating left recursion once is sufficient.

1. Left factoring of grammar

- Factor out the common prefix present in the grammar, i.e, scan the common part once.
- The following example shows a grammar before and after left factoring is done.

$$\begin{array}{lll} S & \rightarrow & i C t S e s \quad | \quad i C t s \quad | \quad a \\ C & \rightarrow & b \end{array}$$

On left factoring -

$$\begin{array}{lll} S & \rightarrow & i C t S X \quad | \quad a \\ C & \rightarrow & b \\ X & \rightarrow & e S \quad | \quad \lambda \end{array}$$

2. Elimination of Left Recursion

A left recursive grammar is typically of the form $A \rightarrow A\alpha$,

where $\alpha \in (V \cup T)^*$

[V = variable, T = terminal]

- Left recursion is eliminated to prevent top down parsers from entering into an infinite loop.

- Left recursion can be

- Immediate - For example -

$$A \rightarrow Aa \mid b$$

- Indirect - Recursion occurs in two or more steps

Example -

$$S \rightarrow Aab$$

$$A \rightarrow Sd \mid \lambda$$

In case of indirect left recursion, whenever there are multiple non-terminals in the grammar, specify the order of nonterminals, which is usually the order in which they are presented.

- This will convert indirect left recursion to immediate left recursion.
- Example -
 - Given -

$$S \rightarrow A a b$$

$$A \rightarrow S d \mid \lambda$$

- Here, **S** is replaced by **Aab** in the second production.

$$S \rightarrow A a b$$

$$A \rightarrow A a b d \mid \lambda$$

Algorithm to eliminate left recursion -

- **Input: Grammar with no cycles or λ productions**
- **Output: Grammar without Left Recursion**
 1. Arrange the non-terminals in some order (usually the order in which they are presented)
 2. Replace each production of the form

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_s$$

with

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

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_s A' \mid \lambda$$

Consider the following grammar -

$$A \rightarrow Bxy \mid x$$

$$B \rightarrow CD$$

$$C \rightarrow A \mid c$$

$$D \rightarrow d$$

Eliminate left recursion.

Given -

$$A \rightarrow Bxy \mid x$$

$$B \rightarrow CD$$

$$C \rightarrow A \mid c$$

$$D \rightarrow d$$

Step 1 - Replace indirect left recursion

$$A \rightarrow Bxy \mid x$$

$$B \rightarrow CD$$

$$C \rightarrow Bxy \mid x \mid c \quad \Rightarrow \quad C \rightarrow CDxy \mid c \mid x$$

$$D \rightarrow d$$

A → B x y | x

B → C D

C → C D x y | c | x

D → d

Step 2 - Eliminate immediate Left recursion

A → B x y | x

B → C D

C → x C' | c C'

C' → D x y C' | λ

D → d

Consider the following grammar -

$$S \rightarrow a A c B$$

$$A \rightarrow A b \mid b \mid b c$$

$$B \rightarrow d$$

Eliminate left recursion.

Consider the following grammar -

$$\begin{array}{lcl} S & \rightarrow & a A c B \\ A & \rightarrow & A b \mid b \mid b c \\ B & \rightarrow & d \end{array}$$

Answer -

$$\begin{array}{lcl} S & \rightarrow & a A c B \\ A & \rightarrow & b A' \mid b c A' \\ A' & \rightarrow & b A' \mid \lambda \end{array}$$

Eliminate left recursion in the following grammar -

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

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

$$F \rightarrow id \mid num \mid (E)$$

Given -

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

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

$$F \rightarrow id \mid num \mid (E)$$

Eliminate immediate left recursions for **E** and **T** productions using the algorithm.

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid \lambda$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \lambda$$

$$F \rightarrow id \mid num \mid (E)$$

- It is a top-down parser that implements a set of recursive procedures to process the input without backtracking.
- Recursive Procedures are written for each non-terminal in the grammar.
- The recursive procedures can be accessible to write and adequately effective if written in a language that performs the procedure call effectively.

Consider the earlier grammar for parsing arithmetic expressions -

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

Notice that the grammar is left factored, and left-recursion is removed.

The RDP implementation for this grammar is present at -
[RDPwithoutBacktracking.c](#)

Consider the following grammar -

$$\begin{array}{lcl} S & \rightarrow & i C t S X \mid a \\ X & \rightarrow & e S \mid b \\ C & \rightarrow & c \end{array}$$

Notice that the grammar is left factored, and left-recursion is removed.



THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu



Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Kavya P K

Compiler Design

Unit 2

Predictive Parsers (LL(1))

Preet Kanwal
Department of Computer Science & Engineering

Compiler Design

Lecture Overview



In this lecture, you will learn about -

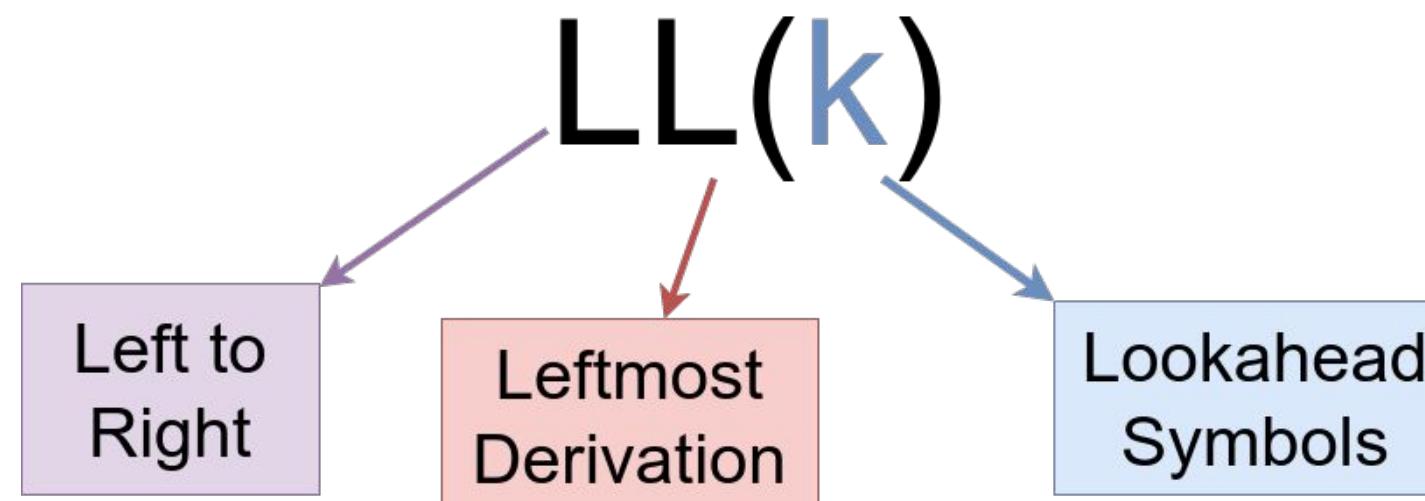
- What are Predictive Parsers?
- Model of a Table-driven Parser
- Computing First Sets
- Computing Follow Sets
- What is an LL(1) Parsing Table?
- Construction of an LL(1) Parsing Table

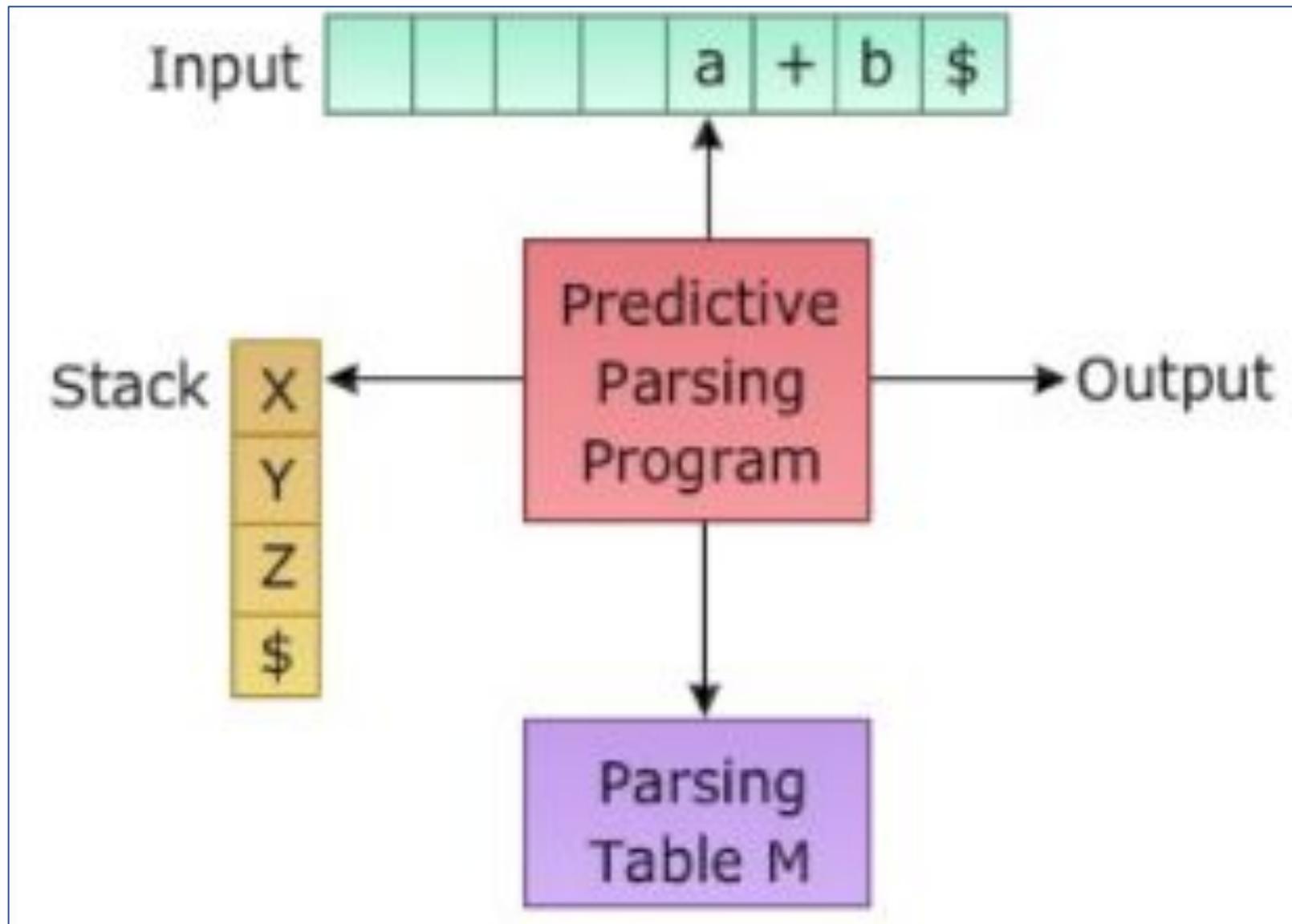
- Predictive Parser is a table driven top-down parser without backtracking.
- Class of grammars for which a table driven parser can be constructed is **LL(k)**.
- A predictive parser uses a lookahead pointer which points to the next input symbols.
- It uses a stack and a parsing table to parse the input and produce a parse tree.
- Both the stack and the input contains an end symbol **\$** to denote that the stack is empty and the input is consumed.

Compiler Design

Predictive Parsers

- Table driven parser can be constructed is for a class of grammars **LL(k)** where
 - The first L indicates that input is parsed from left to right.
 - The second L indicates that the output of the parser, which is a parse tree, is the leftmost derivation of the input string.
 - k denotes the number of symbols used for lookahead.





- The **First()** of a non-terminal is the set of symbols that appear as the first symbol in a string that derives from the non-terminal.
- Rules to compute First set:
 1. **First(t) = { t }, where t is a terminal**
 2. **If $X \rightarrow \lambda$, then add λ to $\text{First}(X)$**
 3. **If $X \rightarrow Y_1 Y_2 Y_3$**
 - **$\text{First}(X) = \text{First}(Y_1)$**
 - **If $\lambda \in \text{First}(Y_1)$, then $\text{First}(X) = \{ \text{First}(Y_1) - \lambda \} \cup \{ \text{First}(Y_2) \}$**
 - **If $\lambda \in \text{First}(Y_i)$ for all $1 \leq i \leq n$, then add λ to $\text{First}(X)$**

- Consider the following grammar:

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \lambda$$
$$T \rightarrow FT'$$
$$T \rightarrow *FT' \mid \lambda$$
$$F \rightarrow id \mid num \mid (E)$$

Find the first set of non terminal T

- Production of T: $T \rightarrow FT'$
- According to Rule 3, $\text{First}(T) = \{ \text{First}(F) \}$
- Consider all productions of F

$F \rightarrow id$

$F \rightarrow num$

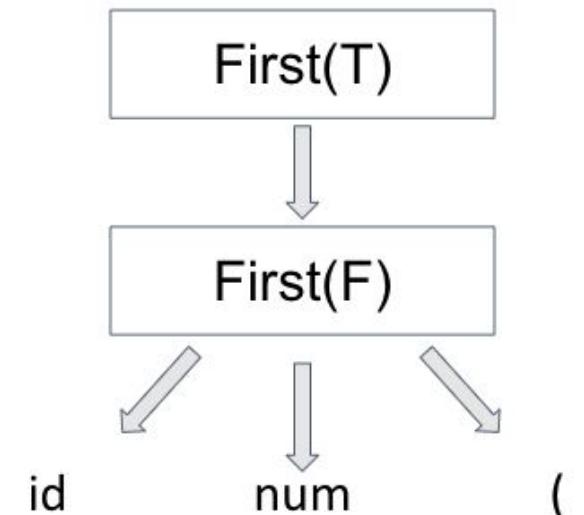
$F \rightarrow ($

- According to Rule 1, $\text{First}(F) = \{ id, num, (\}$

- Since F does not have any λ productions, we stop here.

Since $\text{First}(T) = \text{First}(F)$,

$\text{First}(T) = \{ id, num, (\}$

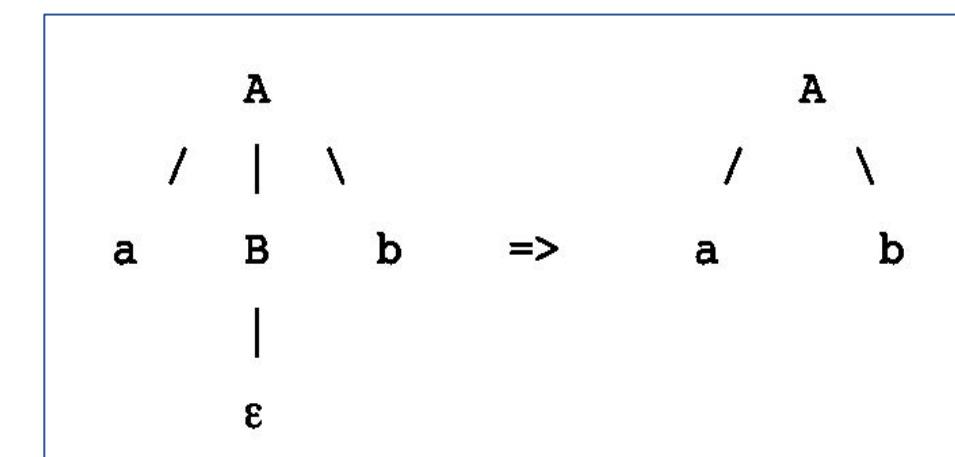
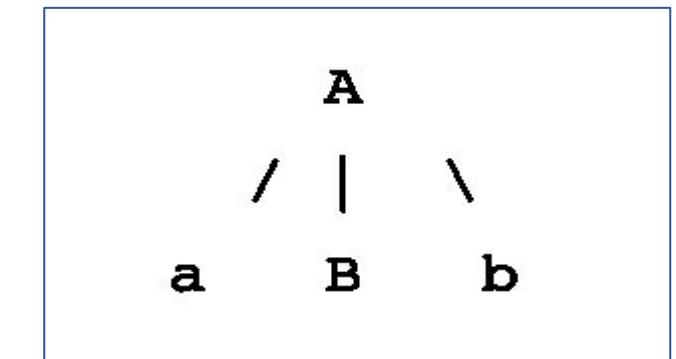


- Consider the partial parse tree on the right. Assume the following grammar:

$A \rightarrow aBb$

$B \rightarrow c \mid \lambda$

- When the input symbol is **b**, the non-terminal to derive is **B**. There is no production of B that starts with **b**, so the parser is now stuck.
- However, if the lambda production of B, $B \rightarrow \lambda$, is applied, B vanishes and the parser parses **ab** correctly.
- A non-terminal should vanish if the symbol that follows the non-terminal in the input is the same as the symbol that follows it in the production rule/in some sentential form.



- Thus, when there are nullable non-terminals in a grammar, there is a need to calculate **follow sets** of each of the nullable non-terminal so that their lambda productions can be used using parsing.
- **Follow(X)** to be the set of terminals that can appear immediately to the right of Non-Terminal X in some sentential form.

- Rules to compute Follow set:

1. $\text{Follow}(S) = \{ \$ \}$, where S is the start symbol
2. If $A \rightarrow \alpha B \beta$, where $\lambda \in \text{First}(\beta)$, then

$$\text{Follow}(B) = \{ \text{First}(\beta) - \lambda \} \cup \{ \text{Follow}(A) \}$$

3. If $A \rightarrow \alpha B$, $\text{Follow}(B) = \text{Follow}(A)$

where $\alpha, \beta \in (V \cup T)^*$

V = Variables (non terminals)

T = terminals

- Consider the following grammar:

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \lambda$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \lambda$$
$$F \rightarrow \text{num} \mid \text{id} \mid (E)$$

- Compute the Follow sets of all non-terminals in the grammar for each production.

Follow Sets - Example 1 - Solution

$E \rightarrow TE'$	$\text{Follow}(E) = \$$ $\text{Follow}(T) = \{ + \} \cup \text{Follow}(E)$ $\text{Follow}(E') = \text{Follow}(E)$
$E' \rightarrow + T E'$	$\text{Follow}(T) = \{ + \} \cup \text{Follow}(E')$ $\text{Follow}(E') = \text{Follow}(E')$
$T \rightarrow FT'$	$\text{Follow}(F) = \{ * \} \cup \text{Follow}(T)$ $\text{Follow}(T') = \text{Follow}(T)$
$T' \rightarrow * FT'$	$\text{Follow}(F) = \{ * \} \cup \text{Follow}(T')$ $\text{Follow}(T') = \text{Follow}(T')$
$F \rightarrow (E)$	$\text{Follow}(E) = \{ \} \}$

- Eliminate Left recursion (if present) and Left factor the grammar(if required).
- Calculate **First()** and **Follow()** sets for all non-terminals.
- Draw a table where the first row contains the Non-Terminals and the first column contains the Terminal Symbols.
- All the Null Productions of the Grammars will go under the elements of **Follow(LHS)**.
 - Reason - The null production must be used only when the parser knows that the character that follows the LHS in the production rule is same as the current input character.

For each production $A \rightarrow \alpha$

1. For each terminal in $\text{First}(\alpha)$, make entry $A \rightarrow \alpha$ in the table.
2. If $\text{First}(\alpha)$ contains λ , then for each terminal in $\text{Follow}(A)$ (i.e, $\text{Follow}(\text{LHS})$) make an entry $A \rightarrow \alpha$ in the table.
3. If the $\text{First}(\alpha)$ contains λ and $\text{Follow}(A)$ contains $\$$ as a terminal, then make an entry $A \rightarrow \alpha$ in the table under $\$$.

Construct the parsing table for the earlier Grammar.

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

From the earlier examples, the first and follow tables for the given grammar is as follows -

First Table				
E	E'	T	T'	F
id	+	id	*	id
num	λ	num	λ	num
(((

Follow Table				
E	E'	T	T'	F
\$	\$	+	+	*
))	\$	\$	+
))	\$
)

Construct a table with terminals on the first row and Non-terminals on the first column.

	id	+	*	num	()	\$
E							
E'							
T							
T'							
F							

Constructing LL(1) Parsing Table - Example 1

For each production in the grammar, add entries to the appropriate locations.

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

$E \rightarrow TE'$

Add the $E \rightarrow TE'$ under each element in $\text{First}(T)$

$\text{First}(T) = \{\text{id}, \text{num}$
and $\{ \}$

Constructing LL(1) Parsing Table - Example 1

For each production in the grammar, add entries to the appropriate locations.

	id	+	*	num	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$				$E' \rightarrow \lambda$	$E' \rightarrow \lambda$
T							
T'							
F							

$E' \rightarrow +TE'$

Add $E' \rightarrow +TE'$
under +

$E' \rightarrow \lambda$

So, Add $E' \rightarrow \lambda$
under all elements
in Follow(LHS), i.e,
{), \$ }

Constructing LL(1) Parsing Table - Example 1

For each production in the grammar, add entries to the appropriate locations.

	id	+	*	num	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$				$E' \rightarrow \lambda$	$E' \rightarrow \lambda$
T	$T \rightarrow FT'$			$T \rightarrow FT'$	$T \rightarrow FT'$		
T'							
F							

$T \rightarrow FT'$

Check First(F).

First(F)={id,num,()}

Add $T \rightarrow FT'$ under each element.

Constructing LL(1) Parsing Table - Example 1

For each production in the grammar, add entries to the appropriate locations.

	id	+	*	num	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$				$E' \rightarrow \lambda$	$E' \rightarrow \lambda$
T	$T \rightarrow FT'$			$T \rightarrow FT'$	$T \rightarrow FT'$		
T'		$T' \rightarrow \lambda$	$T' \rightarrow *FT'$			$T' \rightarrow \lambda$	$T' \rightarrow \lambda$
F							

$T' \rightarrow *FT'$

Add $T' \rightarrow *FT'$
under *

$T' \rightarrow \lambda$

So add $T' \rightarrow \lambda$
under all elements
of $\text{Follow}(T') =$
 $\{+,), \$\}$

Constructing LL(1) Parsing Table - Example 1

For each production in the grammar, add entries to the appropriate locations.

	id	+	*	num	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$				$E' \rightarrow \lambda$	$E' \rightarrow \lambda$
T	$T \rightarrow FT'$			$T \rightarrow FT'$	$T \rightarrow FT'$		
T'		$T' \rightarrow \lambda$	$T' \rightarrow *FT'$			$T' \rightarrow \lambda$	$T' \rightarrow \lambda$
F	$F \rightarrow id$			$F \rightarrow num$	$F \rightarrow (E)$		

$F \rightarrow num \mid id \mid (E)$

Add $F \rightarrow num$

under **num**

$F \rightarrow id$

Add $F \rightarrow id$ under
id

$F \rightarrow (E)$

Add $F \rightarrow (E)$ under
(

Constructing LL(1) Parsing Table - Example 1

Thus, this is the final LL(1) parsing table for the given grammar.

	id	+	*	num	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$				$E' \rightarrow \lambda$	$E' \rightarrow \lambda$
T	$T \rightarrow FT'$			$T \rightarrow FT'$	$T \rightarrow FT'$		
T'		$T' \rightarrow \lambda$	$T' \rightarrow *FT'$			$T' \rightarrow \lambda$	$T' \rightarrow \lambda$
F	$F \rightarrow id$			$F \rightarrow num$	$F \rightarrow (E)$		

- The LL(1) parsing table can now be used to parse an input string. This will be demonstrated in the next class.
- To check whether a grammar belongs to LL(1), check whether each box in the LL(1) parser table has atmost 1 production.
- If a grammar is not Left Factored, or Left Recursion is not removed, it will not belong to LL(1).
- The first table will never contain \$
- The follow table will never contain λ



THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu



Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Kavya P K

Compiler Design

Unit 2

Predictive Parsers (LL(1))

Preet Kanwal
Department of Computer Science & Engineering

Compiler Design

Lecture Overview



In this lecture, you will learn about -

- **LL(1) Parsing Algorithm**
- **Examples on LL(1) parsing algorithm**
- **Identify whether a grammar belongs to LL(1)**
- **LL(k) Grammars**

- Given an LL(1) Parsing table, how do we parse a given input string?
- Initially,
 - Stack contains start symbol **S**
 - Input buffer contains the input string **w**

Stack	Input Buffer	Action
S \$	w \$	

- Parsing Algorithm is applied.
- Finally, if the string is accepted, the stack and input buffer contain **\$**, indicating that they are empty.

Stack	Input Buffer	Action
\$	\$	accept

```
//Let a be the current input symbol and M be the parsing table
//Let X be the current stack top.

while(X != $) {
    if(X==a) { //If X is a terminal
        stack.pop()
        Delete 'a' from input buffer
    }
    else if(M[X,a] = X ->Y1Y2...Yk { //If X is a non terminal
        stack.pop()
        stack.push(Y1,Y2,...Yk) //Y1 is the stack top now
    }
    else
        error()
}
```

- Given the following LL(1) Parsing table M, parse the input string

“id + id * id”

		id	+	*	()	\$
E	E → TE [*]			E → TE [*]			
E [*]		E [*] → +TE [*]			E [*] → ε	E [*] → ε	
T	T → FT [*]			T → FT [*]			
T [*]		T [*] → ε	T [*] → *FT [*]		T [*] → ε	T [*] → ε	
F	F → id			F → (E)			

- Given the following LL(1) Parsing table M, parse the input string:

“id + id * id”

Initially, the stack and input buffer contain -

Stack	Input Buffer	Action
E \$	id + id * id \$	

Stack	Input Buffer	Action
E \$	id + id * id \$	$M[E,id] = E \rightarrow TE'$ pop E from stack push TE'
T E' \$	id + id * id \$	

Now, find the action for the stack top E and current input symbol id in M -

$$M[E,id] = E \rightarrow TE'$$

Based on the parsing algorithm,
pop E from stack
push TE'

Current stack top changes to T.

Stack	Input Buffer	Action
E \$	id + id * id \$	M[E,id] = E->TE' pop E from stack push TE'
T E' \$	id + id * id \$	M[T,id] = T->FT' pop T from stack push FT'
F T' E' \$	id + id * id \$	

M[T,id] = T->FT'
pop T from stack
push FT'

Current stack top changes to F.

Stack	Input Buffer	Action
E \$	id + id * id \$	M[E,id] = E->TE' pop E from stack push TE'
T E' \$	id + id * id \$	M[T,id] = T->FT' pop T from stack push FT'
F T' E' \$	id + id * id \$	M[F,id] = F->id pop F from stack push id
id T' E' \$	id + id * id \$	

M[F,id] = F->id
pop F from stack
push id

Current stack top changes to id.

LL(1) Parsing Algorithm - Example 1

Stack	Input Buffer	Action
E \$	id + id * id \$	M[E,id] = E->TE' pop E from stack push TE'
T E' \$	id + id * id \$	M[T,id] = T->FT' pop T from stack push FT'
F T' E' \$	id + id * id \$	M[F,id] = F->id pop F from stack push id
id T' E' \$	id + id * id \$	M[id,id] pop id from stack delete id from input buffer
T' E' \$	+ id * id \$	

M[id,id] - case 1, id==id - True

pop id from stack
delete id from input buffer

Current stack top changes to T'.

LL(1) Parsing Algorithm - Example 1

Stack	Input Buffer	Action
E \$	id + id * id \$	M[E,id] = E->TE' pop E from stack push TE'
T E' \$	id + id * id \$	M[T,id] = T->FT' pop T from stack push FT'
F T' E' \$	id + id * id \$	M[F,id] = F->id pop F from stack push id
id T' E' \$	id + id * id \$	M[id,id] pop id from stack delete id from input buffer
T' E' \$	+ id * id \$	M[T',+] = T -> λ pop T' from stack Current stack top changes to E'.
E' \$	+ id * id \$	

M[T',+] = T' -> λ
pop T' from stack
Current stack top changes to E'.

Stack	Input Buffer	Action
...
E' \$	+ id * id \$	$M[E',+] = E' \rightarrow +TE'$ pop E' from stack push +TE' to stack
+ T E' \$	+ id * id \$	

$M[E',+] = E' \rightarrow +TE'$
 pop E' from stack
 push +TE' to stack
 Current stack top changes to +.

Stack	Input Buffer	Action
...
E' \$	+ id * id \$	M[E',+] = E' -> +TE' pop E' from stack push +TE' to stack
+ T E' \$	+ id * id \$	M[+,+] = match pop + from stack remove + from input buffer Current stack top changes to T.
T E' \$	id * id \$	

M[+,+] = match
pop + from stack
remove + from input buffer
Current stack top changes to T.

LL(1) Parsing Algorithm - Example 1

Stack	Input Buffer	Action
...
E' \$	+ id * id \$	M[E',+] = E' -> +TE' pop E' from stack push +TE' to stack
+ T E' \$	+ id * id \$	M[+,+] = match pop + from stack remove + from input buffer
T E' \$	id * id \$	M[T,id] = T->FT' pop T from stack push FT' to stack
FT' E' \$	id * id \$	

M[T,id] = T->FT'
pop T from stack
push FT' to stack
Current stack top changes to F.

LL(1) Parsing Algorithm - Example 1

Stack	Input Buffer	Action
...
$E' \$$	$+ id * id \$$	$M[E',+] = E' \rightarrow +TE'$ pop E' from stack push $+TE'$ to stack
$+ T E' \$$	$+ id * id \$$	$M[+,+] = \text{match}$ pop $+$ from stack remove $+$ from input buffer
$T E' \$$	$id * id \$$	$M[T,id] = T \rightarrow FT'$ pop T from stack push FT' to stack
$FT' E' \$$	$id * id \$$	$M[F,id] = F \rightarrow id$ pop F from stack push id to stack
$id T' E' \$$	$id * id \$$	

$M[F,id] = F \rightarrow id$
pop F from stack
push id to stack
Current stack top changes to F .

LL(1) Parsing Algorithm - Example 1

Stack	Input Buffer	Action
...
id T' E' \$	id * id \$	M[id,id] = match pop id from stack remove id from i/p buffer
T' E' \$	* id \$	

M[id,id] = match
pop id from stack
remove id from i/p buffer
Current stack top changes to F.

LL(1) Parsing Algorithm - Example 1

Stack	Input Buffer	Action
...
id T' E' \$	id * id \$	M[id,id] = match pop id from stack remove id from i/p buffer
T' E' \$	* id \$	M[T',*] = T'-> λ pop T' from stack push *FT' to stack
* FT' E' \$	* id \$	

M[T',*] = T'-> *FT'
pop T' from stack
push *FT' to stack
Current stack top changes to E'.

LL(1) Parsing Algorithm - Example 1

Stack	Input Buffer	Action
...
id T' E' \$	id * id \$	M[id,id] = match pop id from stack remove id from i/p buffer
T' E' \$	* id \$	M[T',*] = T'-> λ pop T' from stack push *FT' to stack
* F T' \$	* id \$	M[*,*] = match
F T' \$	id \$	

M[*,*] = match
Current stack top changes to F.

LL(1) Parsing Algorithm - Example 1

Stack	Input Buffer	Action
...
id T' E' \$	id * id \$	M[id,id] = match pop id from stack remove id from i/p buffer
T' E' \$	* id \$	M[T',*] = T'->λ pop T' from stack push *FT' to stack
* F T' \$	* id \$	M[*,*] = match
F T' \$	id \$	M[F,id] = F->id pop F from stack push id to stack
id T' \$	id \$	

M[F,id] = F->id
pop F from stack
push id to stack
Current stack top changes to id.

Stack	Input Buffer	Action
...
id T' \$	id \$	M[id,id] = match
T' \$	\$	M[T',\$] = T'-> λ pop T' from stack
\$	\$	Accept

M[id,id] = match
Current stack top changes to T'.

M[T',\$] = T'-> λ
pop T' from stack

Thus, the string id+id*id is accepted by the parser.

Parse the string **(id * id)**

Construct the LL(1) Parsing table for the given grammar -

$$S \rightarrow a \mid (L)$$

$$L \rightarrow L, S \mid S$$

Then, demonstrate parsing using the input string (a,a).

Given -

$$S \rightarrow a \mid (L)$$

$$L \rightarrow L, S \mid S$$

First, remove left recursion.

- Arrange non-terminals -

$$S \rightarrow a \mid (L)$$

$$L \rightarrow L, S \mid a \mid (L)$$

- Replacing productions with left recursion -

$$S \rightarrow a \mid (L)$$

$$L \rightarrow aL' \mid (L)L'$$

$$L' \rightarrow , S L' \mid \lambda$$

Construct first and follow tables for the modified grammar -

$$S \rightarrow a \mid (L)$$

$$L \rightarrow a L' \mid (L) L'$$

$$L' \rightarrow , S L' \mid \lambda$$

First Table		
S	L	L'
a	a	,
((λ

Follow Table		
S	L	L'
\$))
,		
)		

Construct the LL(1) parsing table.

	a	()	,	\$
S	$S \rightarrow a$	$S \rightarrow (L)$			
L	$L \rightarrow aX$	$L \rightarrow (L)L'$			
L'			$L' \rightarrow \lambda$	$L' \rightarrow ,SL'$	

LL(1) Parsing Algorithm - Example 2

Stack	Input Buffer	Action
S \$	(a , a) \$	M[S,()] = S -> (L)
(L) \$	(a , a) \$	M[(,())] = match
L) \$	a , a) \$	M[L,a] = L->aX
a X) \$	a , a) \$	M[a,a] = match
X) \$, a) \$	M[X,,] = X->,SX
, S X) \$, a) \$	M[,,,] = match
S X) \$	a) \$	M[S,a] = S->a
a X) \$	a) \$	M[a,a] = match
X) \$) \$	M[X,)] = X-> λ
) \$) \$	M[),)] = match
\$	\$	Accept

Parse the input string (a,a)
Parsing successful.

- If the grammar is not left factored and/or left recursion is not eliminated, it does not belong to LL(1).
- If the LL(1) Table has more than one entry in any cell, the grammar does not belong to LL(1) class of grammars.
- Formally, to check whether a grammar belongs to LL(1) without constructing the table, the following rules are checked.
- A grammar is NOT in LL(1) if
 1. For $A \rightarrow a \mid b$, i.e, there are two or more alternatives for a Non-terminal
If $\text{First}(a) \cap \text{First}(b) \neq \Phi$, i.e, if they have something in common.
 2. $A \rightarrow a \mid \lambda$
If $\text{First}(a) \cap \text{Follow}(A) \neq \Phi$

Is the Given grammar in LL(1) ? Also, Compute first and follow sets.

$$S \rightarrow AaAb \mid BbBa$$

$$A \rightarrow \lambda$$

$$B \rightarrow \lambda$$

S → A a A b | B b B a

A → λ

B → λ

Compute First and Follow Sets -

First Table		
S	A	B
a	a	λ
b		

Follow Table		
S	A	B
\$	a	a
	b	b

S → **A a A b | B b B a**

is of the format

S → **a | b**

$$\text{First}(AaAb) = \text{First}(A) = \{ a \}$$

$$\text{First}(BbBa) = \text{First}(B) = \{ \lambda \}$$

Since $\text{First}(A) \cap \text{First}(B) = \Phi$, the given grammar belongs to LL(1)

Is the Given grammar in LL(1) ? If not, modify and re-check.

S → **i C t S** | **i C t S e S** | **a**

C → **b**

- If we construct the parse table M for the following grammar,

$$S \rightarrow i C t S \mid i C t S e S \mid a$$
$$C \rightarrow b$$

- $M[S,i]$ will have 2 productions

$$S \rightarrow i C t S$$
$$S \rightarrow i C t S e S$$

- Therefore, it is not in LL(1).
- To modify, left factor the above grammar -

$$S \rightarrow i C t S X \mid a$$
$$X \rightarrow e S \mid \lambda$$
$$C \rightarrow b$$

- Is the modified grammar in LL(1) ?

$S \rightarrow i C t S X \mid a$

$X \rightarrow e S \mid \lambda$

$C \rightarrow b$

- Notice the production

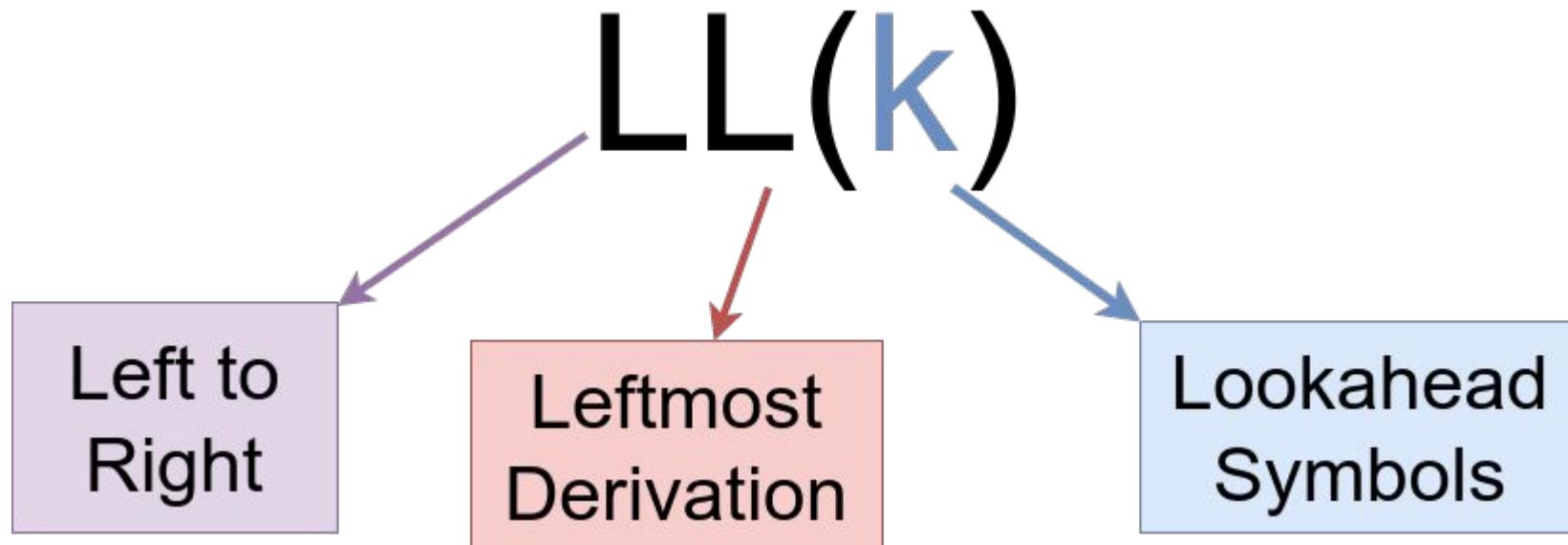
$X \rightarrow e S \mid \lambda$

is of the form

$A \rightarrow a \mid \lambda$

- $\text{First}(e) = e$
- $\text{Follow}(X) = \{ e, \$ \}$
- Thus, $\text{First}(e) \cap \text{Follow}(X) = e \neq \emptyset$
- Therefore, the modified grammar is also not in LL(1).

As mentioned previously,



- When $k = 1$, we can choose the next production rule using 1 lookahead symbol.
- That is, by reading 1 input character.
- For example,

$$S \rightarrow a \mid b$$

- For $k > 1$, we can choose the next production after reading k input characters.
- For example,

$$S \rightarrow a b \mid a c \mid a d$$

- Here, the production can be chosen only after reading 2 input characters.
- Therefore, the above grammar belongs to LL(2) class of parsers.
- $LL(k) \subset LL(k+1)$
- If the LL(k) Table has more than k entries in any cell, the grammar does not belong to LL(k) class of grammars, where $k > 1$.

Are the following grammars in LL(k)? If yes, find k.

1. $S \rightarrow iCtS \mid iCtSeS \mid a$

$C \rightarrow b$

2. $S \rightarrow a a B \mid a a C$

$B \rightarrow b$

$C \rightarrow c$

3. $E \rightarrow T + E \mid T$

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

1. $S \rightarrow iCtS \mid iCtSeS \mid a$
 $C \rightarrow b$

Answer - No. It is not left-factored.

2. $S \rightarrow aaB \mid aaC$
 $B \rightarrow b$
 $C \rightarrow c$

Answer - Yes. It belongs to LL(3).

3. $E \rightarrow T+E \mid T$
 $T \rightarrow id \mid id * T \mid (E)$

Answer - Yes. It belongs to LL(2), as the grammar needs 2 input symbols to choose the production.



THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu



Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Kavya P K

Compiler Design

Unit 2

Predictive Parsers (LL(1))

Preet Kanwal
Department of Computer Science & Engineering

Compiler Design

Lecture Overview



In this lecture, you will learn about -

- Additional examples on LL(1) parsing algorithm
- LL(k) Grammars

Construct the LL(1) Parsing table for the given grammar -

$$S \rightarrow AB$$

$$A \rightarrow a \mid \lambda$$

$$B \rightarrow b \mid \lambda$$

Then, demonstrate parsing using the input string λ .

Given -

$$S \rightarrow AB$$

$$A \rightarrow a \mid \lambda$$

$$B \rightarrow b \mid \lambda$$

No left recursion present, and grammar is left factored.

Construct first and follow tables for the grammar -

$$S \rightarrow AB$$

$$A \rightarrow a \mid \lambda$$

$$B \rightarrow b \mid \lambda$$

First Table		
S	A	B
a	a	b
b	λ	λ
λ		

Follow Table		
S	A	B
\$	b	\$
	\$	

Construct the LL(1) parsing table.

	a	b	\$
S	$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$
A	$A \rightarrow a$	$A \rightarrow \lambda$	$A \rightarrow \lambda$
B		$B \rightarrow b$	$B \rightarrow \lambda$

Stack	Input Buffer	Action
S\$	\$	$M[S,\$] = S \rightarrow AB$
AB\$	\$	$M[A,\$] = A \rightarrow \lambda$
B\$	\$	$M[B,\$] = B \rightarrow \lambda$
\$	\$	Accept

Parse the input string λ
Parsing successful.

Is the Given grammar in LL(1) ? Also, Compute first and follow sets. Parse the input string cde

S → ABCDE

A → a | λ

B → b | λ

C → c

D → d | λ

E → e | λ

$$\begin{array}{l}
 S \rightarrow ABCDE \\
 A \rightarrow a \mid \lambda \\
 B \rightarrow b \mid \lambda \\
 C \rightarrow c \\
 D \rightarrow d \mid \lambda \\
 E \rightarrow e \mid \lambda
 \end{array}$$

Compute First and Follow Sets -

First Table						
S	A	B	C	D	E	
a	a	b	c	d	e	
b	λ	λ		λ	λ	
c						

Follow Table						
S	A	B	C	D	E	
\$	b	c	d	e	\$	
	c		e	\$		
			\$			

Construct the LL(1) parsing table.

	a	b	c	d	e	\$
S	$S \rightarrow ABCDE$	$S \rightarrow ABCDE$	$S \rightarrow ABCDE$			
A	$A \rightarrow a$	$A \rightarrow \lambda$	$A \rightarrow \lambda$			
B		$B \rightarrow b$	$B \rightarrow \lambda$			
C			$C \rightarrow c$			
D				$D \rightarrow d$	$D \rightarrow \lambda$	$D \rightarrow \lambda$
E					$E \rightarrow e$	$E \rightarrow \lambda$

LL(1) Parsing Algorithm - Example 6

Stack	Input Buffer	Action
S \$	cde\$	M[S,c] = S → ABCDE
ABCDE\$	cde\$	M[A,c] = A → λ
BCDE\$	cde\$	M[B,c] = B → λ
CDE\$	cde\$	M[C,c] = C → c
cDE\$	cde\$	M[c,c] = match
DE\$	de\$	M[D,d] = D → d
dE\$	de\$	M[d,d] = match
E\$	e\$	M[E,e] = E → e
e\$	e\$	M[e,e] = match
\$	\$	Accept

Parse the input string cde
Parsing successful.

Are the following grammars in LL(k)? If yes, find k.

1. $S \rightarrow Abbx \mid Bbby$

$A \rightarrow x$

$B \rightarrow x$

2. $S \rightarrow Z$

$Z \rightarrow aMa \mid bMb \mid aRb \mid bRa$

$M \rightarrow c$

$R \rightarrow c$

Are the following grammars in LL(k)? If yes, find k.

1. $S \rightarrow Abbx \mid Bbby$

$A \rightarrow x$

$B \rightarrow x$

Answer - Yes. It belongs to LL(4).

2. $S \rightarrow Z$

$Z \rightarrow aMa \mid bMb \mid aRb \mid bRa$

$M \rightarrow c$

$R \rightarrow c$

Answer - Yes. It belongs to LL(3).



THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu



Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Kavya P K

Compiler Design

Unit 2

Predictive Parsers (LL(1))

Preet Kanwal
Department of Computer Science & Engineering

Compiler Design

Lecture Overview



In this lecture, you will learn about -

- Error recovery in LL(1) Parser
- RDP implementation without backtracking vs Predictive
Parsers

- When an action $M[X,Y]$ is blank in the Parsing Table, and it is encountered during parsing of an input string, it indicates **syntax error**.
- How can we handle errors in LL(1) parsers?
- One of the most commonly used Error recovery strategies in Parsers is **Panic Mode Recovery**.
- Recall that in Panic mode Recovery,
 - The parser will discard the input symbols until it finds a delimiter, and then restart the parsing process.
 - Such delimiters are called **synchronizing tokens**.

How is Panic Mode Recovery implemented in LL(1) Parsers?

- For each Non-terminal X in the parsing table M, add a synchronising token **sync** under the elements of **Follow(X)** if the cell is blank.
- Parsing procedure -

If $M[X,a] == \text{blank}$ // i.e, syntax error

ignore the input symbol a

If $M[X,a] == \text{sync}$

If X is the only symbol in the Stack

Ignore input symbol a

else

Pop X from stack

Implement the parsing table for the following grammar with panic mode recovery. Also, parse the string

“)id*+id”

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

Given,

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

First Table				
E	E'	T	T'	F
id	+	id	*	id
num	λ	num	λ	num
(((

Follow Table				
E	E'	T	T'	F
\$	\$	+	+	*
))	\$	\$	+
))	\$
)

The LL(1) parsing table without panic mode recovery-

	id	+	*	num	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$				$E' \rightarrow \lambda$	$E' \rightarrow \lambda$
T	$T \rightarrow FT'$			$T \rightarrow FT'$	$T \rightarrow FT'$		
T'		$T' \rightarrow \lambda$	$T' \rightarrow *FT'$			$T' \rightarrow \lambda$	$T' \rightarrow \lambda$
F	$F \rightarrow id$			$F \rightarrow num$	$F \rightarrow (E)$		

Adding sync tokens under elements of Follow(X), where X is a non-terminal.

	id	+	*	num	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	$E \rightarrow TE'$	sync	sync
E'		$E' \rightarrow +TE$,				$E' \rightarrow \lambda$	$E' \rightarrow \lambda$
T	$T \rightarrow FT'$	sync		$T \rightarrow FT'$	$T \rightarrow FT'$	sync	sync
T'		$T' \rightarrow \lambda$	$T' \rightarrow *FT'$			$T' \rightarrow \lambda$	$T' \rightarrow \lambda$
F	$F \rightarrow id$	sync	sync	$F \rightarrow num$	$F \rightarrow (E)$	sync	sync

Follow Table

E	E'	T	T'	F
\$	\$	+	+	*
))	\$	\$	+
))	\$
)

Now, using the new LL(1) Parsing table M, parse the input string

) id * + id

Initially, the stack and input buffer contain -

Stack	Input Buffer	Action
E \$) id * + id \$	

Stack	Input Buffer	Action
E\$)id*+id\$	M[E,)] = sync skip
E\$	id*+id\$	

M[E,)] = sync

Based on the new parsing algorithm,
Since E is the only symbol in the stack,
ignore the current input symbol)

LL(1) Parsing Algorithm - Example 1

Stack	Input Buffer	Action
E \$) id * + id \$	M[E,)]=sync skip
E \$	id * + id \$	M[E,id] = E->TE'
T E' \$	id * + id \$	M[T,id] = T->FT'
F T' E' \$	id * + id \$	M[F,id] = F->id
id T' E' \$	id * + id \$	M[id,id] = match
T' E' \$	* + id \$	M[T',*] = T'->*FT'
* F T' E' \$	* + id \$	M[*,*] = match
F T' E' \$	+ id \$	M[F,+]=sync Pop F
T' E' \$	+ id \$	

Continue parsing.

M[F,+] = sync

F is not the only symbol in the stack.
So, Pop F from stack.

The current stack top is T'

LL(1) Parsing Algorithm - Example 1

Stack	Input Buffer	Action
...
T' E' \$	+ id \$	M[T',+] = T' -> λ
E' \$	+ id \$	M[E',+] = +TE'
+ T E' \$	+ id \$	M[+,+] = match
T E' \$	id \$	M[T,id] = T -> FT'
F T' E' \$	id \$	M[F,id] = F->id
id T' E' \$	id \$	M[id,id] = match
T' E' \$	\$	M[T',\$] = T' -> λ
E' \$	\$	M[E',\$] = E' -> λ
\$	\$	Accept

Continue parsing.

Thus, the input string is parsed till the end even though it contained several syntax errors.

Parse the erroneous string id (+) id

RDP without Backtracking	Predictive Parsers
<p>It uses mutually recursive procedures for every non-terminal entity to parse strings.</p>	<p>It uses a lookahead pointer which points to next k input symbols. This places a constraint on the grammar.</p>
<p>It accepts all grammars.</p>	<p>It accepts only a LL(k) grammars.</p>

Thus, RDP without Backtracking is more powerful, as it accepts a larger set of grammars.



THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu



Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Kavya P K

Compiler Design

Unit 2

Predictive Parsers (LL(1))

Preet Kanwal
Department of Computer Science & Engineering

Compiler Design

Lecture Overview



In this lecture, you will learn about -

- Additional examples on Error recovery in LL(1) Parser

Implement the parsing table for the following grammar with panic mode recovery. Also, parse the string **a(a)**

$$S \rightarrow a \mid (L)$$

$$L \rightarrow a L' \mid (L) L'$$

$$L' \rightarrow , S L' \mid \lambda$$

Construct first and follow tables for the grammar -

$$S \rightarrow a \mid (L)$$

$$L \rightarrow a L' \mid (L) L'$$

$$L' \rightarrow , S L' \mid \lambda$$

First Table		
S	L	L'
a	a	,
((λ

Follow Table		
S	L	L'
\$))
,		
)		

The LL(1) parsing table without panic mode recovery is -

	a	()	,	\$
S	$S \rightarrow a$	$S \rightarrow (L)$			
L	$L \rightarrow aX$	$L \rightarrow (L)L'$			
L'			$L' \rightarrow \lambda$	$L' \rightarrow ,SL'$	

Error Recovery in LL(1) Parser - Example 2

Adding sync tokens under elements of Follow(X), where X is a non-terminal.

	a	()	,	\$
S	$S \rightarrow a$	$S \rightarrow (L)$	sync	sync	sync
L	$L \rightarrow aX$	$L \rightarrow (L)L'$	sync		
L'			$L' \rightarrow \lambda$	$L' \rightarrow ,SL'$	

Follow Table		
S	L	L'
\$))
,		
)		

Now, using the new LL(1) Parsing table M, parse the input string **a(a)**

Initially, the stack and input buffer contain -

Stack	Input Buffer	Action
S\$	a (a) \$	

Stack	Input Buffer	Action
S\$	a (a) \$	$M[S,a] = S \rightarrow a$
a\$	a (a) \$	$M[a,a] = \text{match}$
\$	(a) \$	

Parse the erroneous string $^,(,a,a)$



THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu



Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Lavitra Kshitij Madan

Compiler Design

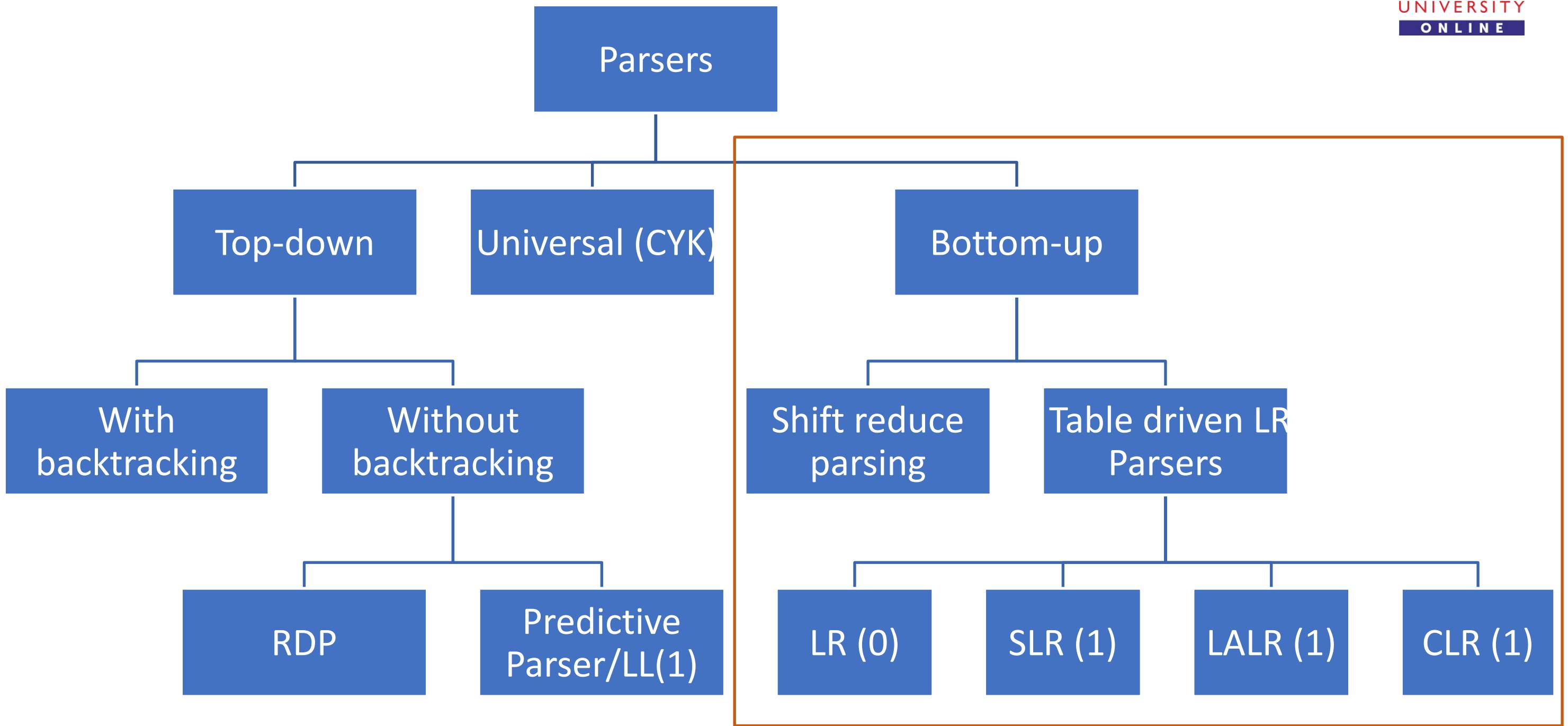
Unit 2

Bottom-up Parsers

Preet Kanwal
Department of Computer Science & Engineering

In this lecture, you will learn about:

- **Introduction to bottom-up parsing**
- **General style of bottom-up parsing: Shift/Reduce parser**
- **Conflicts during shift/reduce parsing with examples**



Difference between top-down and bottom-up parsers

Top-down parsers have two requirements. The grammar must be:

- **Left Factored**
- **Free of Left Recursion**

It read input from left-to-right and provided the leftmost derivation of the input string “w”.

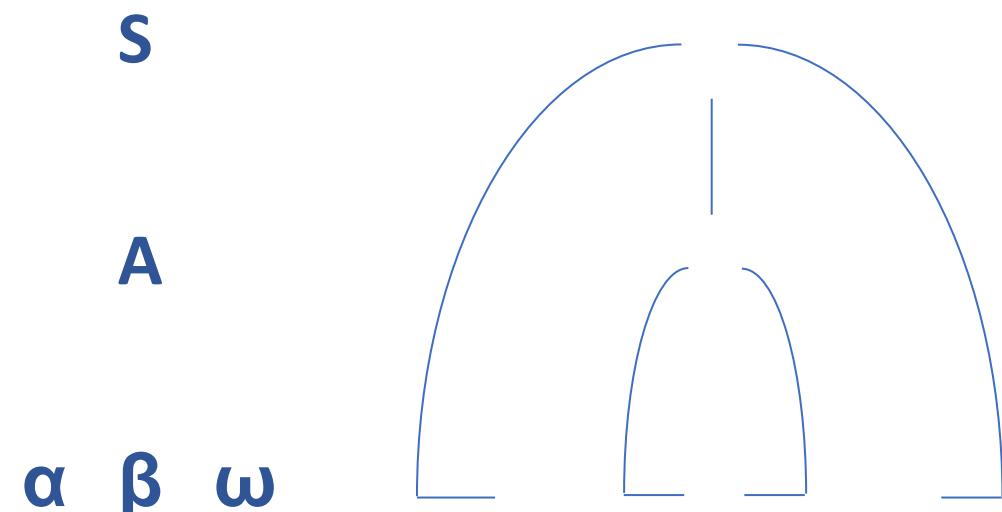
Whereas, the **bottom-up** parsers do not have such requirements and also read input from left-to-right but provide the **rightmost derivation in reverse**.

Compiler Design

Some definitions

Handle : It is that substring on the stack that matches the body (RHS) of a production rule.

Handle pruning : A left-to-right, bottom-up parse works by iteratively searching for a handle, then reducing the handle to the left hand side of the production. The process is repeated until we get the Start Symbol.



For the grammar:
 $S \rightarrow \alpha A \omega$
 $A \rightarrow \beta$

“ β ” is first reduced to “ A ” and then, “ $\alpha A \omega$ ” is reduced to “ S ”

Bottom-up parsing is the process of reducing a string “w” to the start symbol “S”.

Consider the following grammar:

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

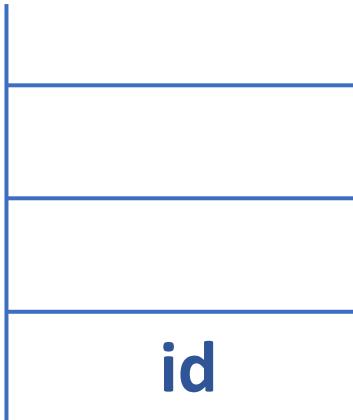
$$T \rightarrow T * F \mid F$$

$$F \rightarrow id$$

- Let the string ‘w’ be “id * id”
- The string will be parsed from left-to-right.
- Bottom-up parsers use stacks.

Compiler Design

Reduction



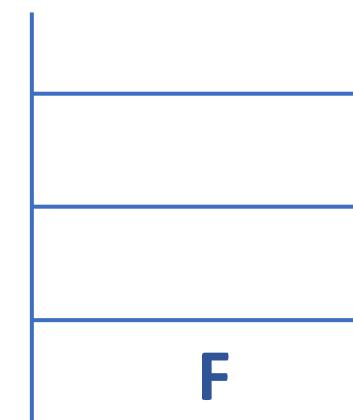
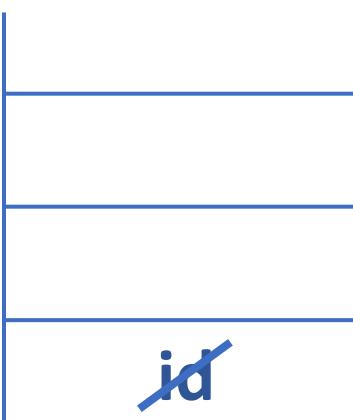
The substring (at Top-of-Stack or TOS) that matches the right-hand-side of a production rule is replaced by the left-hand-side of that production rule. This process is called “reduction”.

Notice that the TOS is “id” and it matches the RHS of the third production rule. We can reduce it to “F”.

Grammar:

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

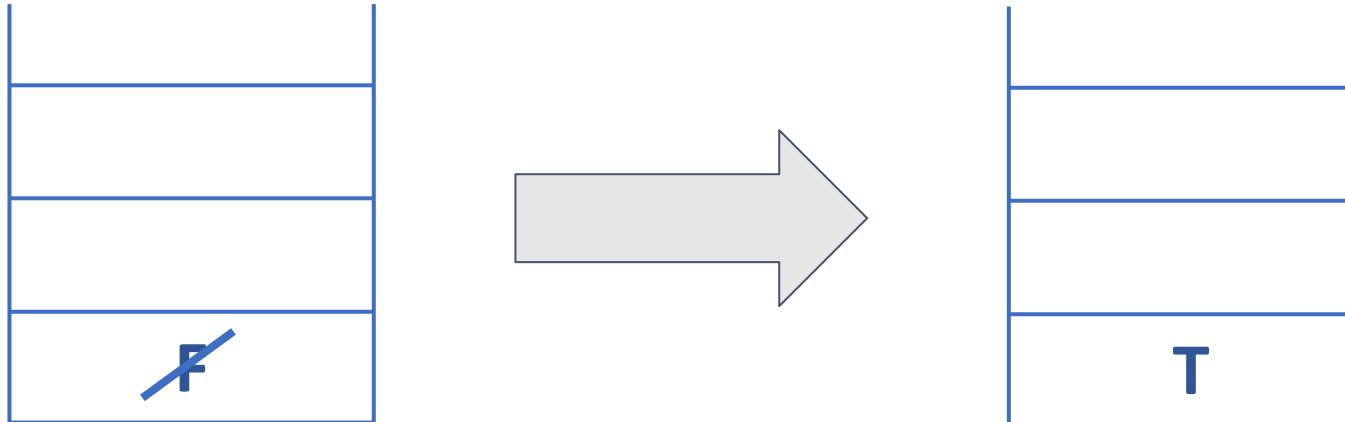
w = “id * id”



...

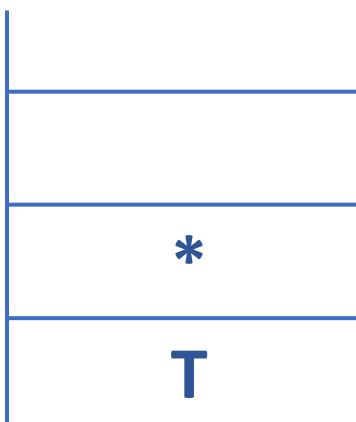
Compiler Design

Shifting



Similarly, we can reduce “F” to “T” using the second production rule.

Now, we should not reduce “T” to “E” using the first production rule because there is no production that has “E *” on its RHS. But, what we can do is that we can shift the next symbol “*” from the input onto the stack.

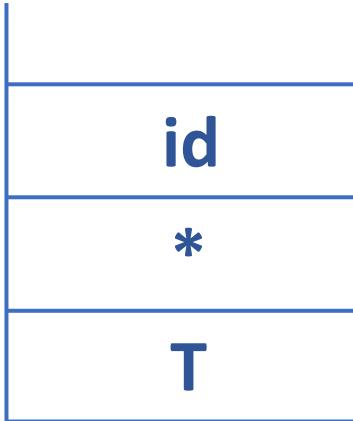


This step has a “conflict” since we had the choice to either shift or reduce. Conflicts may arise in ambiguous grammar. The less powerful compilers do not know what to do when conflicts arise

Grammar:

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

w = “id * id”



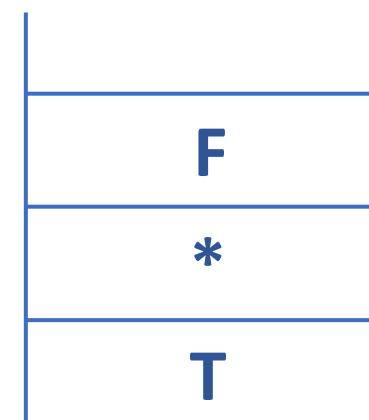
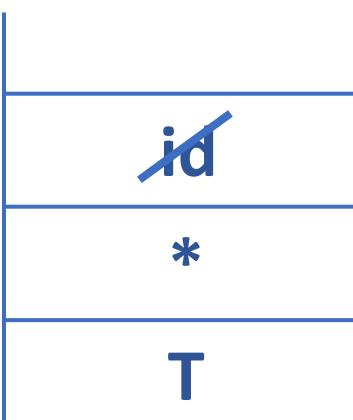
Since we do not have a handle at TOS, we can shift another symbol onto the stack.

The TOS now has “id” which can be reduced to “F” using the third production rule.

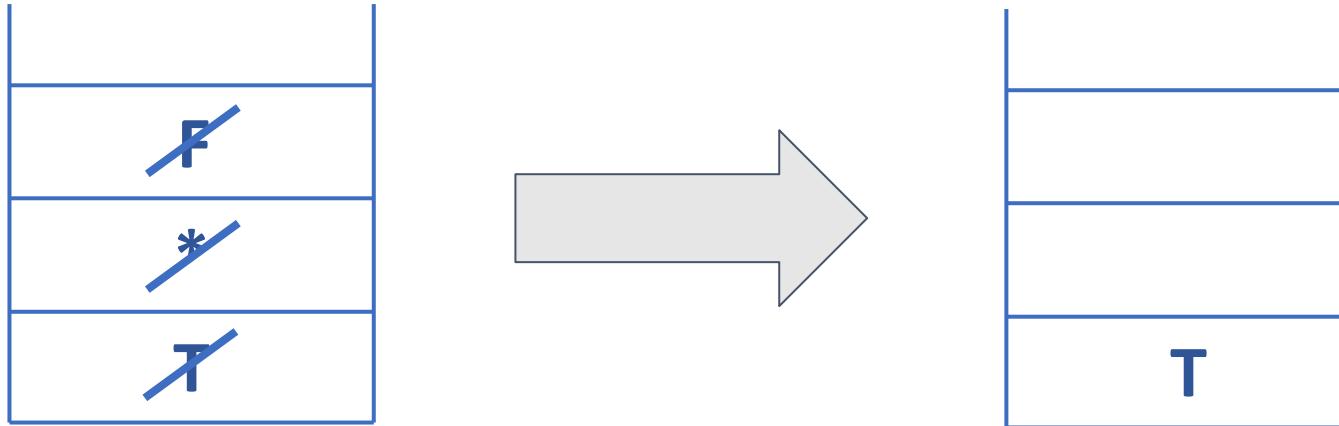
Grammar:

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

w = “id * id”



Bottom-up parsing example (continued)



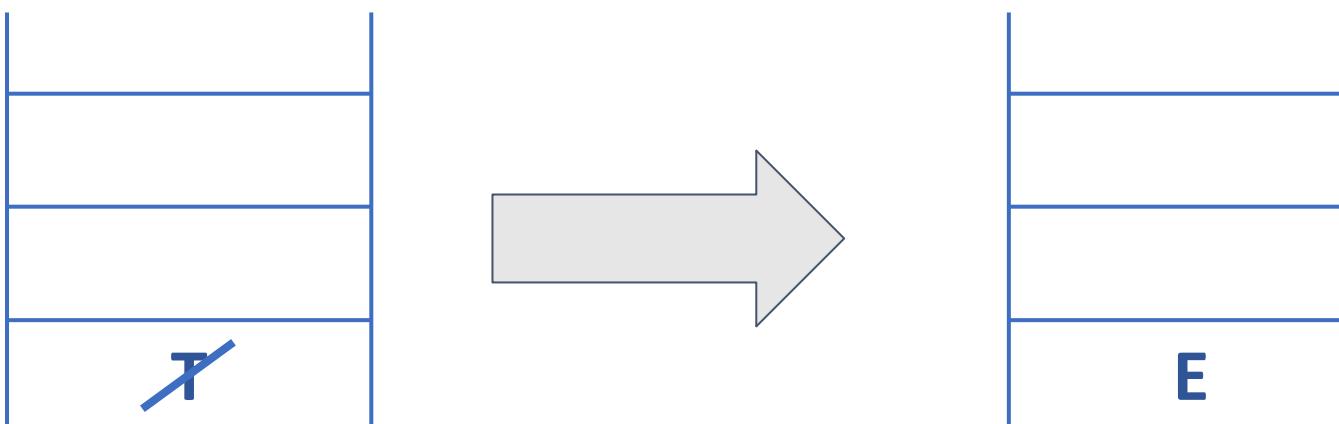
Now, “T * F” perfectly matches the RHS of the second production rule and so it can be reduced to just “T”.

Finally, the TOS now has only “T” which is the RHS of the first production rule and can be reduced to “E” using the same. We have reached the start symbol “E” and so, the string ‘w’ was successfully parsed using the given grammar.

Grammar:

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

w = “id * id”



Summary of the example problem:

id * id

F * id

T * id

T * F

T

E

Rightmost derivation in reverse order:

$E \Rightarrow T$

$\Rightarrow T * F$

$\Rightarrow T * id$

$\Rightarrow F * id$

$\Rightarrow id * id$

Grammar:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow id$

$w = "id * id"$

The output of a Bottom-up parser, if $w \in L(G)$ is the rightmost derivation of the string but in reverse order.

Shift/Reduce Parsing

Bottom-up parsing is the process of reducing a string “w” to the start symbol “S”.

We will now use a table to solve the problem step-by-step.

The table consists of three columns:

- Stack (status)
- Input buffer (status)
- Action (Shift, reduce, error, accept)

Note: Whatever occurs on the stack is known as viable prefix.
The parser will never let you go past your handle. This will be covered in detail in the following slides.

Solution explained using a table (continued)

Consider the same grammar and input example.

The initial configuration is as follows:

Stack	I/p buffer	Action
\$	id * id \$	

It consists of an empty stack and a full input buffer.

We first shift “id” onto the stack.

Stack	I/p buffer	Action
\$	id * id \$	shift id
\$ id	* id \$	

Grammar:
 $E \rightarrow E + T \mid T$
 $T \rightarrow T * F \mid F$
 $F \rightarrow id$

w = “id * id”

Solution explained using a table (continued)

At TOS, “id” can be reduced to “F”.

Stack	I/p buffer	Action
\$	id * id \$	shift id
\$ id	* id \$	Reduce using (iii)
\$ F	* id \$	

“F” is further reduced to “T”.

Stack	I/p buffer	Action
\$	id * id \$	shift id
\$ id	* id \$	Reduce using (iii)
\$ F	* id \$	Reduce using (ii)
\$ T	* id \$	

Grammar:

- (i) $E \rightarrow E + T \mid T$
- (ii) $T \rightarrow T * F \mid F$
- (iii) $F \rightarrow id$

w = “id * id”

Solution explained using a table (continued)

Since shifting “*” was preferred over reducing “T” to “E”, “*” is now shifted onto the stack.

Stack	I/p buffer	Action
\$	id * id \$	shift id
\$ id	* id \$	Reduce using (iii)
\$ F	* id \$	Reduce using (ii)
\$ T	* id \$	Shift *
\$ T *	id \$	

Grammar:

- (i) $E \rightarrow E + T \mid T$
- (ii) $T \rightarrow T * F \mid F$
- (iii) $F \rightarrow id$

w = “id * id”

Solution explained using a table (continued)

Since TOS does not form the RHS of any rule, we now shift the last symbol “id” onto the stack.

Stack	I/p buffer	Action
\$	id * id \$	Shift id
\$ id	* id \$	Reduce using (iii)
\$ F	* id \$	Reduce using (ii)
\$ T	* id \$	Shift *
\$ T *	id \$	Shift id
\$ T * id	\$	

Grammar:

- (i) $E \rightarrow E + T \mid T$
- (ii) $T \rightarrow T * F \mid F$
- (iii) $F \rightarrow id$

w = “id * id”

Using the third production rule, “id” is reduced to “F”.

Stack	I/p buffer	Action
\$	id * id \$	Shift id
\$ id	* id \$	Reduce using (iii)
\$ F	* id \$	Reduce using (ii)
\$ T	* id \$	Shift *
\$ T *	id \$	Shift id
\$ T * id	\$	Reduce using (iii)
\$ T * F	\$	

Grammar:

- (i) $E \rightarrow E + T \mid T$
- (ii) $T \rightarrow T * F \mid F$
- (iii) $F \rightarrow id$

w = “id * id”

Solution explained using a table (continued)

Using the second production rule, “T * F” is reduced to “T”.

Stack	I/p buffer	Action
\$	id * id \$	Shift id
\$ id	* id \$	Reduce using (iii)
\$ F	* id \$	Reduce using (ii)
\$ T	* id \$	Shift *
\$ T *	id \$	Shift id
\$ T * id	\$	Reduce using (iii)
\$ T * F	\$	Reduce using (ii)
\$ T	\$	

Grammar:

- (i) $E \rightarrow E + T \mid T$
- (ii) $T \rightarrow T * F \mid F$
- (iii) $F \rightarrow id$

w = “id * id”

Solution explained using a table (continued)

Finally, using the first production rule, “T” is reduced to “E”.

Stack	I/p buffer	Action
\$	id * id \$	Shift id
\$ id	* id \$	Reduce using (iii)
\$ F	* id \$	Reduce using (ii)
\$ T	* id \$	Shift *
\$ T *	id \$	Shift id
\$ T * id	\$	Reduce using (iii)
\$ T * F	\$	Reduce using (ii)
\$ T	\$	Reduce using (i)
\$ E	\$	

Grammar:

- (i) $E \rightarrow E + T \mid T$
- (ii) $T \rightarrow T * F \mid F$
- (iii) $F \rightarrow id$

w = “id * id”

Solution explained using a table (continued)

We have reached the start symbol and can accept the string.

Stack	I/p buffer	Action
\$	id * id \$	Shift id
\$ id	* id \$	Reduce using (iii)
\$ F	* id \$	Reduce using (ii)
\$ T	* id \$	Shift *
\$ T *	id \$	Shift id
\$ T * id	\$	Reduce using (iii)
\$ T * F	\$	Reduce using (ii)
\$ T	\$	Reduce using (i)
\$ E	\$	Accept

Grammar:

- (i) $E \rightarrow E + T \mid T$
- (ii) $T \rightarrow T * F \mid F$
- (iii) $F \rightarrow id$

w = "id * id"

Conflicts during shift/reduce Parsing

In the example discussed earlier, we had encountered a situation where we could either reduce using a production rule or shift a new symbol onto the stack. It was an example of a **conflict**.

There are two types of conflicts:

1. Shift-reduce conflict (s/r)
2. Reduce-reduce conflict (r/r)

A **shift-reduce conflict** occurs when there are two production rules such that the top-of-stack exactly matches the right-hand-side of one rule and is also a prefix of the right-hand-side of the other rule.

In this situation, we can either reduce the handle to the left-hand-side of the first production rule or we can read another symbol from the input buffer and push it onto the stack.

Example: The dangling-else problem.

Dangling-else problem (Shift-reduce conflict)

Consider the following grammar:

S → if expr then stmt
→ if expr then stmt else stmt
→ other

If the status of the stack is:

\$... if expr then stmt

And, the status of the input buffer is:

else ... \$

We can either reduce the TOS to “S” or can push “else” onto the stack and try to match the second production rule.

Dangling-else problem (Shift-reduce conflict)

Consider the following grammar:

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

If the status of the stack is:

\$... if expr then stmt

And, the status of the input buffer is:

else ... \$

The yacc tool resolves this conflict by choosing the shift operation over the reduce operation and gives a warning!

We can either reduce the TOS to “S” or can push “else” onto the stack and try to match the second production rule.

Pseudo-code in C of an example of this problem:

```
1. int a = 10, b = 20, c = 30;
2. if (a>b)
3.     if (a>c)
4.         printf("%d", a);
5.     else if (c>=a)
6.         printf("%d", c);
7.     else if (b>c)
8.         printf("%d", b);
9. else
10.    printf("%d", c);
```

The else if on line 7 will be considered as a part of the inner if statement because the compiler shifts it rather than reducing lines 2 to 6 together.

A **reduce-reduce conflict** occurs when there are two production rules such that the top-of-stack exactly matches the right-hand-side of both the rules.

In this situation, we can reduce the handle to the left-hand-side of any of the two production rules.

The compiler will not know which of the two production rules to choose for reduction in such an ambiguous case.

The yacc tool resolves this conflict by simply choosing the first rule and gives a warning!

Example of reduce-reduce conflict

Consider the following grammar:

stmt	$\rightarrow id \ (parameter_list)$
parameter_list	$\rightarrow parameter_list, \ parameter \mid parameter$
parameter	$\rightarrow id$
expr	$\rightarrow id \mid id(expr_list)$
Expr_list	$\rightarrow expr \mid expr_list, \ expr$

If the status of the stack is:

\$... id (**id**

And, the status of the input buffer is:

, id) ... \$

Func (5)
or
Func (7)

How to
differentiate
between a
procedure and
an expression
(array)

We can reduce the TOS ("id") to "expr" or to "parameter".
Both are valid!



THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu



Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Philip Joseph

Compiler Design

Unit 2

Bottom-Up Parsing

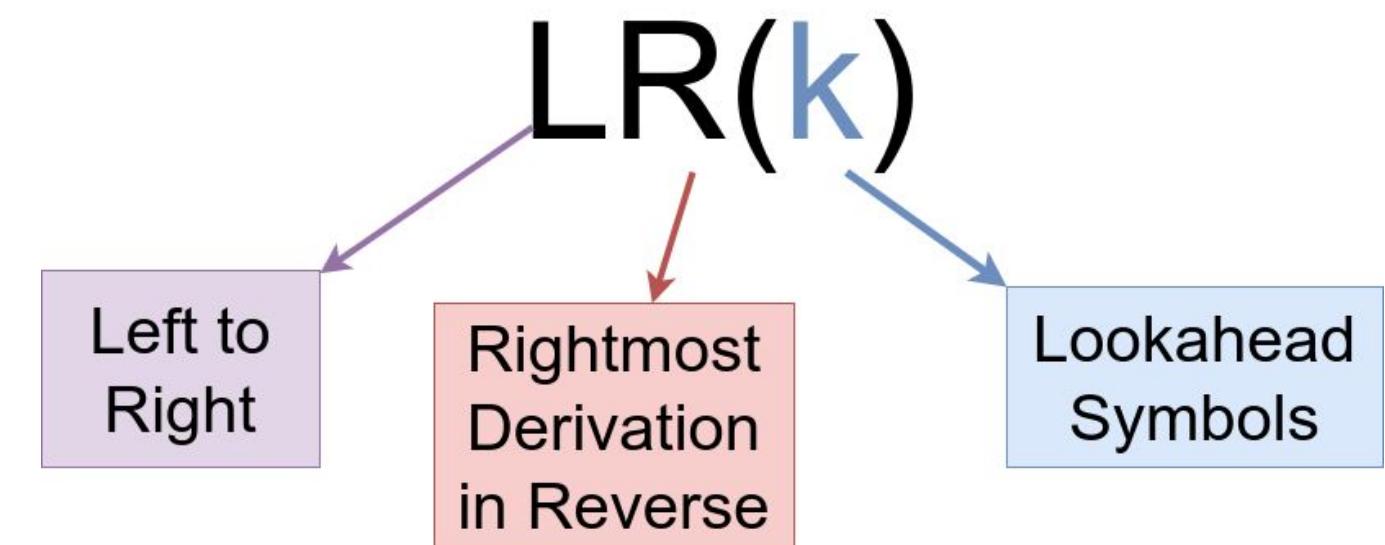
Preet Kanwal
Department of Computer Science & Engineering

In this lecture, we will learn about:

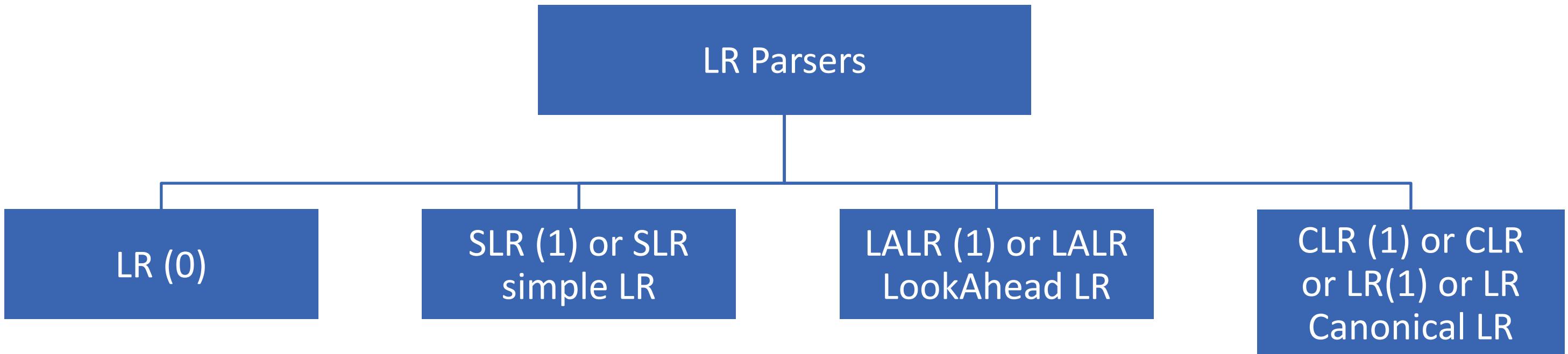
- Types of LR Parsers
- LR(0) Item
- Model of an LR Parser
- Construction of LR(0) Parser with example

- A class of grammar for which we can build shift-reduce parser (BUPs) is called **LR grammars**.
- LR parsing in most general cases is non-backtracking and shift-reduce parsing.
- **LL(1)-Grammar ⊂ LR(1)-Grammar**

- In general, it is $LR(k)$ and is a set of table-driven parsers, where
 - The first **L** indicates that input is parsed from left to right.
 - The **R** indicates that the output of a bottom-up parser is the **Rightmost derivation of the input string in Reverse**.
 - **k** denotes the number of symbols used for lookahead.
 - ‘k’ is always greater than or equal to 1.
 - The default value of ‘k’ is 1.



There are 4 types of LR parsers.



The **power** of a parser is how quickly it can catch errors.

The power of these parsers increase as we move from left-to-right in the order given above.

The number of states required by the 4 types of parsers is-

$$(n_{LR(0)} = n_{SLR} = n_{LALR}) \leq n_{CLR}$$

Compiler Design

LR Automata



- All the four types of LR parsers use DFA (Deterministic Finite Automaton)
- The DFA of LR(0) and SLR(1) parsers is known as LR(0) automata. This is because it makes use of LR(0) items.
- The DFA of LALR and CLR parsers is known as LR(1) automata because it uses LR(1) items.

Compiler Design

Items

Searching for handle:

When using a shift/reduce parser, we must decide whether to shift or reduce at each point.

- We only want to reduce when we know we have a handle.
- Question: How can we tell that we might be looking at a handle?
- An item is a production with a dot in right-hand-side of the production. Example : A \rightarrow XY.Z
- “.” indicates where we are in this production.
- When . is at the end of the production, it is called a final item or a kernel item and indicates that the handle is on TOS and we can perform reduction.

Compiler Design

Items

For the example used in the previous slide, the possible items are:

$$A \rightarrow .XYZ$$

$$A \rightarrow X.YZ$$

$$A \rightarrow XY.Z$$

$$A \rightarrow XYZ.$$

“ $A \rightarrow X.YZ$ ” means that X is seen (possibly on stack) and Y, Z are expected to be seen next.

“ $A \rightarrow X Y Z .$ ” means that XYZ is seen. The entire right-hand-side of the production is on the stack and so it can be reduced.

Compiler Design

Items



At any instant in time, the contents of the left side of the parser can be described using the following process:

- Trace out, from the start symbol, the series of productions that have not yet been completed and where we are in each production.
- For each production, in order, output all of the symbols up to the point where we change from one production to the next.

Compiler Design

Items



Idea: At each point, track

- Which production we are in, and
- Where we are in that production.

At each point, we can do one of two things:

- Match the next symbol of the candidate left-hand side with the next symbol in the current production, or
- If the next symbol of the candidate left-hand side is a nonterminal, nondeterministically guess which production to try next.

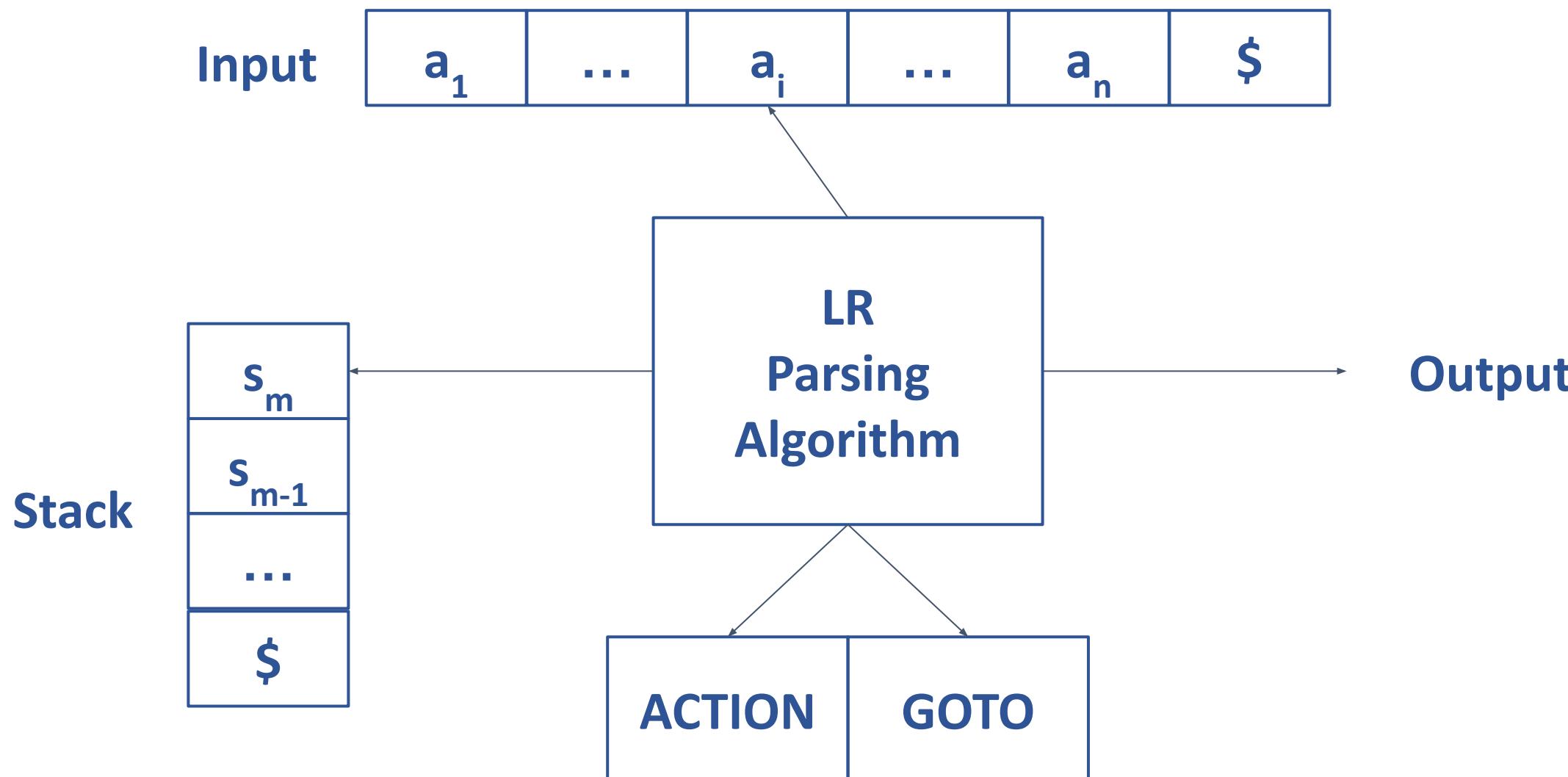
Compiler Design

Items



An Important Result:

- There are only finitely many productions, and within those productions only finitely many positions.
- At any point in time, we only need to track where we are in one production.
- There are only finitely many options we can take at any one point.
- We can use a finite automaton as our recognizer.



The “ACTION” and “GOTO” parts make up the **LR Parsing Table**.

The structure of the LR Parsing Table and the LR Parsing Program is the same for all the LR Parsers.

But, different parsers differ in the content of the table.

More about the “ACTION” part:

- It is like a guide about when to shift or when to reduce.
- Any conflicts (shift/reduce or reduce/reduce) will arise here.

Steps for constructing the LR automaton

1. Augment the grammar
2. Number your productions in the original grammar
3. Follow the LR(0) Parsing Algorithm
4. Create the parsing table

Let us try to understand the process by solving a problem step-by-step.

Examine whether the following grammar is in LR(0) or not.

$$S \rightarrow A A$$

$$A \rightarrow a A \mid b$$

We have to make an LR(0) automata (DFA)

There are two functions that will help us create the DFA:

- Closure
- Goto

Possible items
are:

$$\begin{aligned} S &\rightarrow . A A \\ S &\rightarrow A . A \\ S &\rightarrow A A . \end{aligned}$$

Augment the grammar

Augmenting the grammar involves introduction of a new start symbol. This is done to make sure that only the start symbol is on the stack when the input buffer has \$ to accept the string

$$S' \rightarrow S$$

$$S \rightarrow A A$$

$$A \rightarrow a A$$

$$A \rightarrow b$$

Number your productions in the original grammar

	S'	\rightarrow	S
1.	S	\rightarrow	A A
2.	A	\rightarrow	a A
3.	A	\rightarrow	b

We will start with the initial item $\Rightarrow S' \rightarrow . S$

Follow the LR(0) Parsing Algorithm

1. Start with the initial item.
2. If there is a non-terminal immediately after the “.” in any of the items, add all the productions from that non-terminal.
3. For every newly added item, repeat step 2.
4. If no more items can be added, it becomes a closure/state.
5. Use the goto function to define a transition for each symbol after the dot.
6. Repeat the process until no new states can be added to the automaton.

Follow the LR(0) Parsing Algorithm

Step 1: Starting

Starting with the initial item.

$$S' \rightarrow . S$$

Step 2: Adding productions of NTs after the dot

Since “S” follows the dot, we put the first production.

$$S' \rightarrow . S$$

$$S \rightarrow . A A$$

Production rules:

$$\begin{array}{lll} S' & \rightarrow & S \\ 1. S & \rightarrow & A A \\ 2. A & \rightarrow & a A \\ 3. A & \rightarrow & b \end{array}$$

Follow the LR(0) Parsing Algorithm

Step 3: Repeating step 2 until no more items

Since “A” follows the dot, we put productions 2 and 3.

$$\begin{array}{ll} S' & \rightarrow \cdot S \\ S & \rightarrow \cdot A A \\ A & \rightarrow a A \\ A & \rightarrow b \end{array}$$

Step 4: Making closure

Since no more items can be added to the state, we can now make it a closure. It is done by adding the productions in a box and naming it.

State I0	
S'	$\rightarrow \cdot S$
S	$\rightarrow \cdot A A$
A	$\rightarrow \cdot a A$
A	$\rightarrow \cdot b$

Production rules:

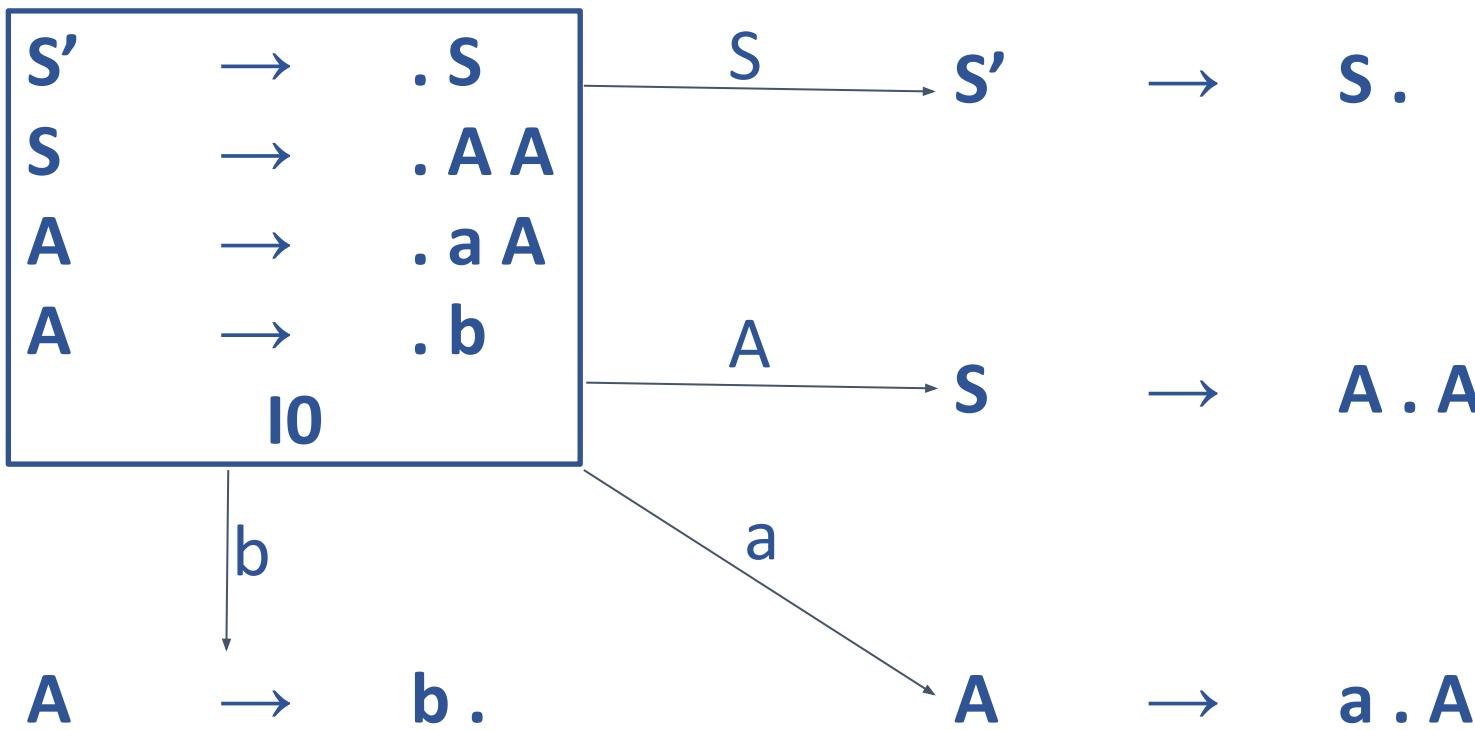
$$\begin{array}{ll} S' & \rightarrow S \\ 1. S & \rightarrow A A \\ 2. A & \rightarrow a A \\ 3. A & \rightarrow b \end{array}$$

Follow the LR(0) Parsing Algorithm

Step 5: Transitions using the goto function

Goto (I, X) \Rightarrow Define transition from state 'I' on every symbol 'X' after the dot.

We will have transitions on "S", "A", "a" and "b".



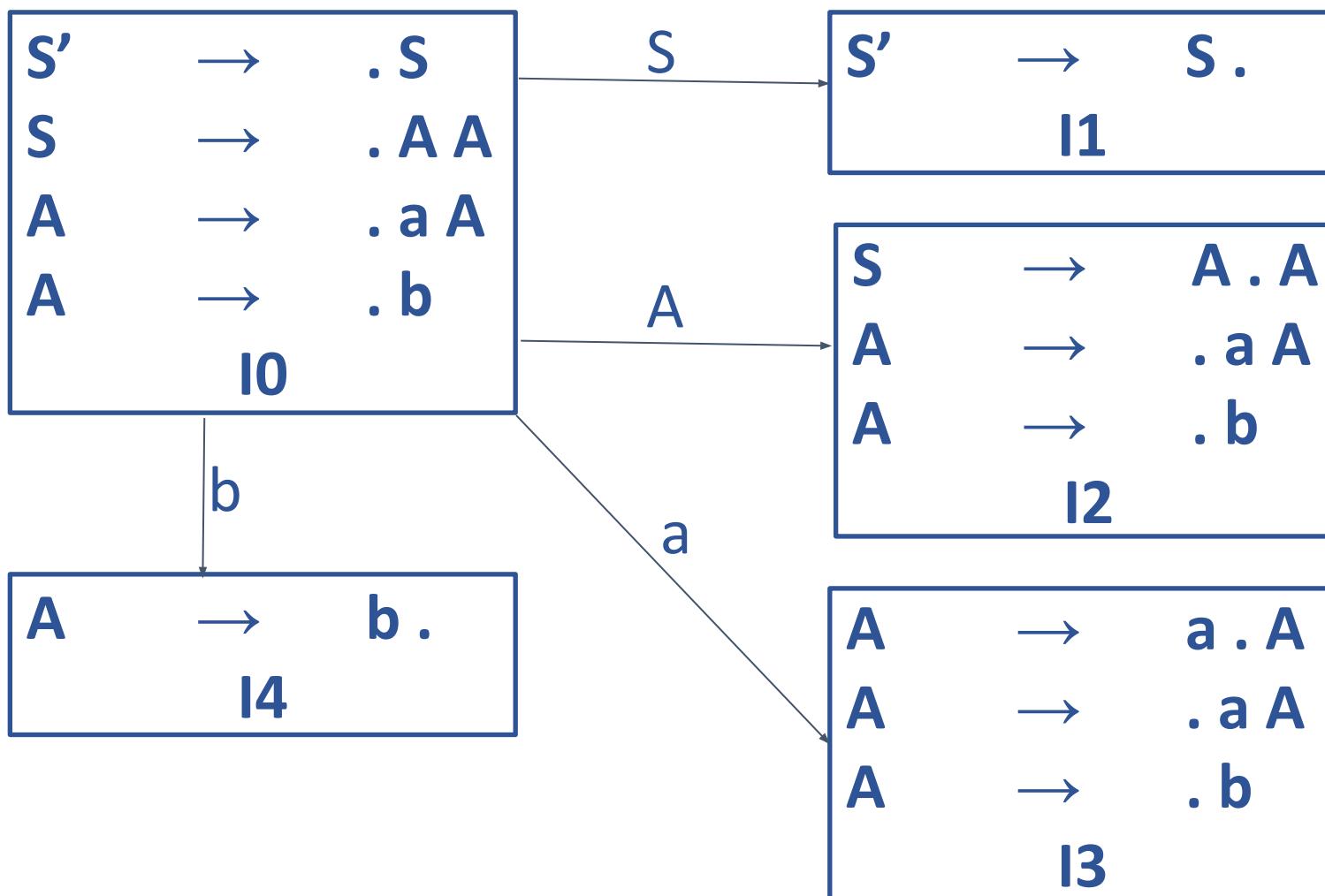
Production rules:

S'	\rightarrow	S
1. S	\rightarrow	AA
2. A	\rightarrow	aA
3. A	\rightarrow	b

Follow the LR(0) Parsing Algorithm

Step 6: Repeat steps 2-5 until no more states can be added

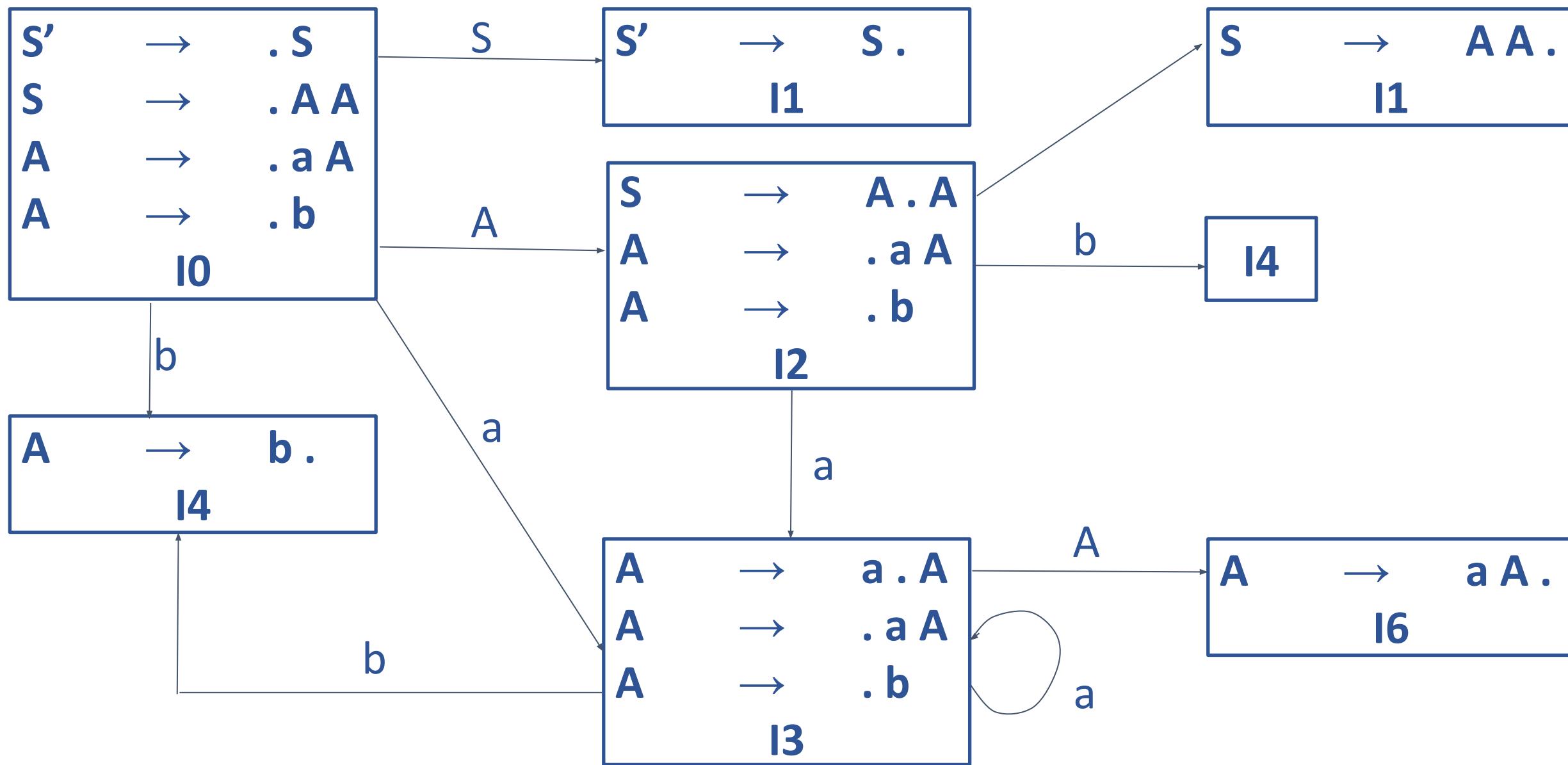
$S' \rightarrow S.$ and $A \rightarrow b.$ are already completed and closure can be used on them. The other two transitions require steps 2-5.



Production rules:

$S' \rightarrow S$	$S \rightarrow AA$
1. $S \rightarrow AA$	2. $A \rightarrow aA$
2. $A \rightarrow aA$	3. $A \rightarrow b$

Follow the LR(0) Parsing Algorithm (Step 6)



- The canonical collection of LR(0) items is represented by $C = \{ I_0, I_1, I_2, I_3, I_4, I_5, I_6 \}$
- Notice the transition from state I_2 to state I_4 . It can be made to directly point to the box that actually contains all the items that correspond to the state I_4 , just like how the transition from state I_2 to I_3 and from state I_3 to I_4 is, but it will make the automata look very cluttered.
- $S' \rightarrow S .$
The aforementioned item is known as the **special final item**. It indicates that the start symbol is on the stack and thus the given grammar can be accepted by the parser.

The LR(0) parsing table consists of three parts-

- State
- Action
- Goto

The state numbers from the automaton diagram are used to fill the 'state' column of the parsing table.

The 'action' column has sub-columns for each terminal used in the grammar and also for “\$” symbol.

The 'goto' column has sub-columns for each non-terminal used in the grammar.

State	Action			Goto	
	a	b	\$	S	A
0					

We will start filling the LR Parsing Table now.

For the state I0, transition over “S” leads to state I1 and so, we fill the ‘S’ sub-column in the ‘goto’ column as “1”.

Similarly, the sub-column ‘A’ gets a value “2”.

Transition on a symbol is called **shift**. It is denoted by “ s_x ”, where X is the next state.

When a state has a final item, we must **reduce it**. It is denoted by “ r_x ”, where X is the next state.

State	Action			Goto	
	a	b	\$	S	A
0	s_3	s_4		1	2

State 1 contains the special final item. “Accept” should be put under the ‘\$’ sub-column in the ‘Action’ column.

State	Action			Goto	
	a	b	\$	S	A
0	s_3	s_4		1	2
1			Accept		

Similarly filling for states I2 and I3.

State	Action			Goto	
	a	b	\$	s	A
0	s_3	s_4		1	2
1			Accept		
2	s_3	s_4			5
3	s_3	s_4			6

States I4, I5 and I6 have a final item each and so, we must reduce them. In LR(0) we place “ r_x ” in the entire action column for that state and this is a disadvantage of this parser(as it does not perform any lookahead). The “X” subscript takes the value of the rule number that the production rule corresponds to.

Remember that we had numbered all the productions as part of the second step of the solution.

State	Action			Goto	
	a	b	\$	s	A
0	s_3	s_4		1	2
1			Accept		
2	s_3	s_4			5
3	s_3	s_4			6
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

Compiler Design

Conclusion

Note that if we have multiple entries in a cell of the parsing table, the grammar is not in LR(0).

In our example grammar, since we do not have multiple entries, we say that the given grammar:

$$S \rightarrow AA$$

$$A \rightarrow aA \mid b$$

is in LR(0).

This concludes the solution of the example problem.

In the next class, we will see how to parse a string using the table we have developed!



THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu



Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Lavitra Kshitij Madan

Compiler Design

Unit 2

Table-driven BUP

Preet Kanwal
Department of Computer Science & Engineering

In this lecture, we will learn about:

- LR(0) parsers (example problems)
- Disadvantages of LR(0) parsers

Recall the example from our previous lecture.

We had successfully created an LR(0) automaton and LR(0) parsing table for the following grammar:

$$S \rightarrow AA$$

$$A \rightarrow aA \mid b$$

Here is the parsing table for reference.

State	Action			Goto	
	a	b	\$	S	A
0	s_3	s_4		1	2
1			Accept		
2	s_3	s_4			5
3	s_3	s_4			6
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

Let us now see the moves made by LR(0) parser on an example string “abb”.

The initial configuration is as follows:

Stack	I/p buffer	Action
\$ 0	w \$	

Start state **Input String**

We stop parsing when, for an action, we get the output as “Accept” according to the parsing table.

Look at the top of the stack and the next symbol in the input buffer to be read. Find the corresponding cell in the parsing table and perform that action.

Stack	I/p buffer	Action
\$ 0	abb \$	A[0, a] = s ₃ Push 'a' on stack Push 3 on stack
\$ 0a3	bb \$	

Now continue similarly.

Compiler Design

Parsing a string using LR(0)

Stack	I/p buffer	Action
\$ 0	abb \$	$A[0, a] = s_3$ Push 'a' on stack Push 3 on stack
\$ 0a3	bb \$	$A[3, b] = s_4$ Push 'b' on stack Push 4 on stack
\$ 0a3b4	b \$	

Stack	I/p buffer	Action
\$ 0	abb \$	$A[0, a] = s_3$ Push 'a' on stack Push 3 on stack
\$ 0a3	bb \$	$A[3, b] = s_4$ Push 'b' on stack Push 4 on stack
\$ 0a3b4	b \$	$A[4, b] = r_3$ {Reduce previous 'b' using 3rd rule} Pop 4 from stack Pop 'b' from stack Push 'A' on stack {TOS must be a state number and so, use Goto function} Goto[3, A] = 6 Push 6 on stack
\$ 0a3A6	b \$	

If size of RHS of reducing production is x, then pop 2x symbols off the stack.

Hence, in our case,

$$A \rightarrow b$$

Size of RHS= $|b| = 1$

Therefore, we pop 2 symbols off the stack and push the LHS of the production "A"

Compiler Design

Parsing a string using LR(0)

Stack	I/p buffer	Action
:	:	:
\$ 0a3A6	b \$	A[6, b] = r_2 ($A \rightarrow a A$) Pop 6, 'A', 3, 'a' from stack Push 'A' on stack Push Goto[0,A] = 2
\$ 0A2	b \$	

Compiler Design

Parsing a string using LR(0)

Stack	I/p buffer	Action
:	:	:
\$ 0a3A6	b \$	$A[6, b] = r_2 (A \rightarrow a A)$ Pop 6, 'A', 3, 'a' from stack Push 'A' on stack Push Goto[0,A] = 2
\$ 0A2	b \$	$A[2, b] = s_4$ Push 'b' on stack Push 4 on stack
\$ 0A2b4	\$	

Compiler Design

Parsing a string using LR(0)

Stack	I/p buffer	Action
:	:	:
\$ 0a3A6	b \$	$A[6, b] = r_2 (A \rightarrow a A)$ Pop 6, 'A', 3, 'a' from stack Push 'A' on stack Push Goto[0,A] = 2
\$ 0A2	b \$	$A[2, b] = s_4$ Push 'b' on stack Push 4 on stack
\$ 0A2b4	\$	$A[4, \$] = r_3 (A \rightarrow b)$ Pop 4, 'b' from stack Push 'A' on stack Push Goto[2, A] = 5
\$ 0A2A5	\$	

Compiler Design

Parsing a string using LR(0)

Stack	I/p buffer	Action
:	:	:
\$ 0A2b4	\$	A[4, \$] = $r_3 (A \rightarrow b)$ Pop 4, 'b' from stack Push 'A' on stack Push Goto[2, A] = 5
\$ 0A2A5	\$	A[5, \$] = $r_1 (S \rightarrow A A)$ Pop 5, 'A', 2, 'A' from stack Push 'S' on stack Push Goto[0, S] = 1
\$ 0S	\$	

Compiler Design

Parsing a string using LR(0)

Stack	I/p buffer	Action
:	:	:
\$ 0A2b4	\$	$A[4, \$] = r_3 (A \rightarrow b)$ Pop 4, 'b' from stack Push 'A' on stack Push Goto[2, A] = 5
\$ 0A2A5	\$	$A[5, \$] = r_1 (S \rightarrow A A)$ Pop 5, 'A', 2, 'A' from stack Push 'S' on stack Push Goto[0, S] = 1
\$ 0S1	\$	$A[1, \$] = \text{Accept}$

Parsing is successful.

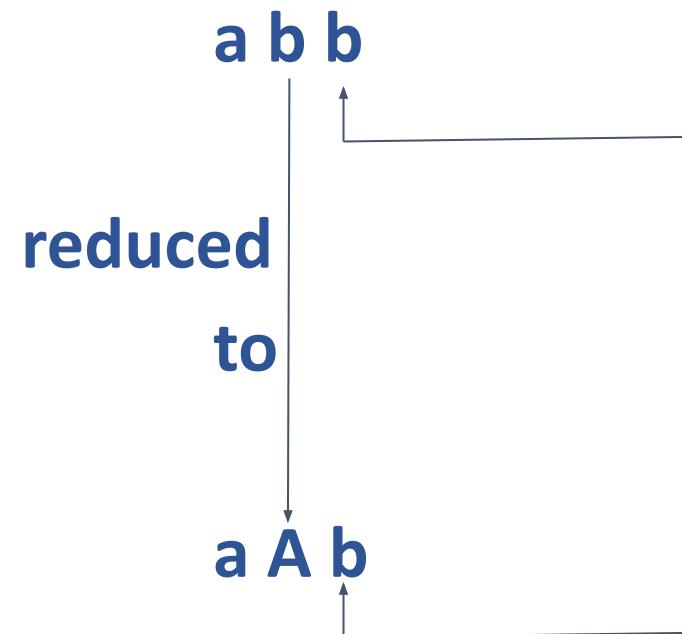
Compiler Design

Is something wrong?

Note that LR(0) does not care about next symbol. It does not use lookahead.

Whenever there is a final item, we put the reduce move in the entire row corresponding to the state that contains the final item.

Considering the string,



Being at this position, we decide to reduce the previous 'b'.

Hence, we replace 'b' by 'A' using $A \rightarrow b$

Irrespective of the symbol we have here, we always reduce previous 'b' to 'A'.

This behaviour can be erroneous

Is the following grammar in LR(0)?

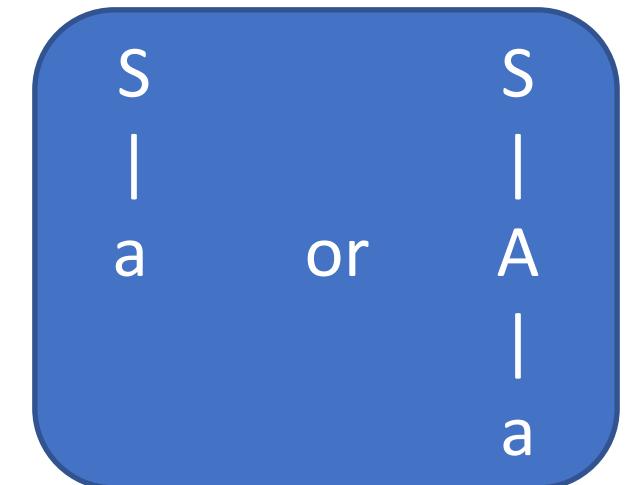
$$S \rightarrow A \mid a$$

$$A \rightarrow a$$

- In such questions, make a quick check if the grammar is ambiguous. Giving the DFA and the table is not necessary.
- Just check if any string can be derived in more than one ways.
- In this example, “a” can be derived directly from ‘S’ or indirectly from ‘S’ via ‘A’.
- Once proved that the grammar is ambiguous, conclude that the grammar is not in LR(0).

Note: No parser can work with ambiguous grammar.

However, let us take this example to easily understand when r/r conflicts can occur.



Is the following grammar in LR(0)?

$$S \rightarrow A \mid a$$

$$A \rightarrow a$$

Solution:

Augmenting and numbering the grammar-

$$S' \rightarrow S$$

1. $S \rightarrow A$

2. $S \rightarrow a$

3. $A \rightarrow a$

Step 1: Starting

Starting with the initial item.

$$S' \rightarrow . S$$

Step 2: Adding productions of NTs after the dot

$$S' \rightarrow . S$$

$$S \rightarrow . A$$

$$S \rightarrow . a$$

Step 3: Repeating step 2 until no more items

$$S' \rightarrow . S$$

$$S \rightarrow . A$$

$$S \rightarrow . a$$

$$A \rightarrow . a$$

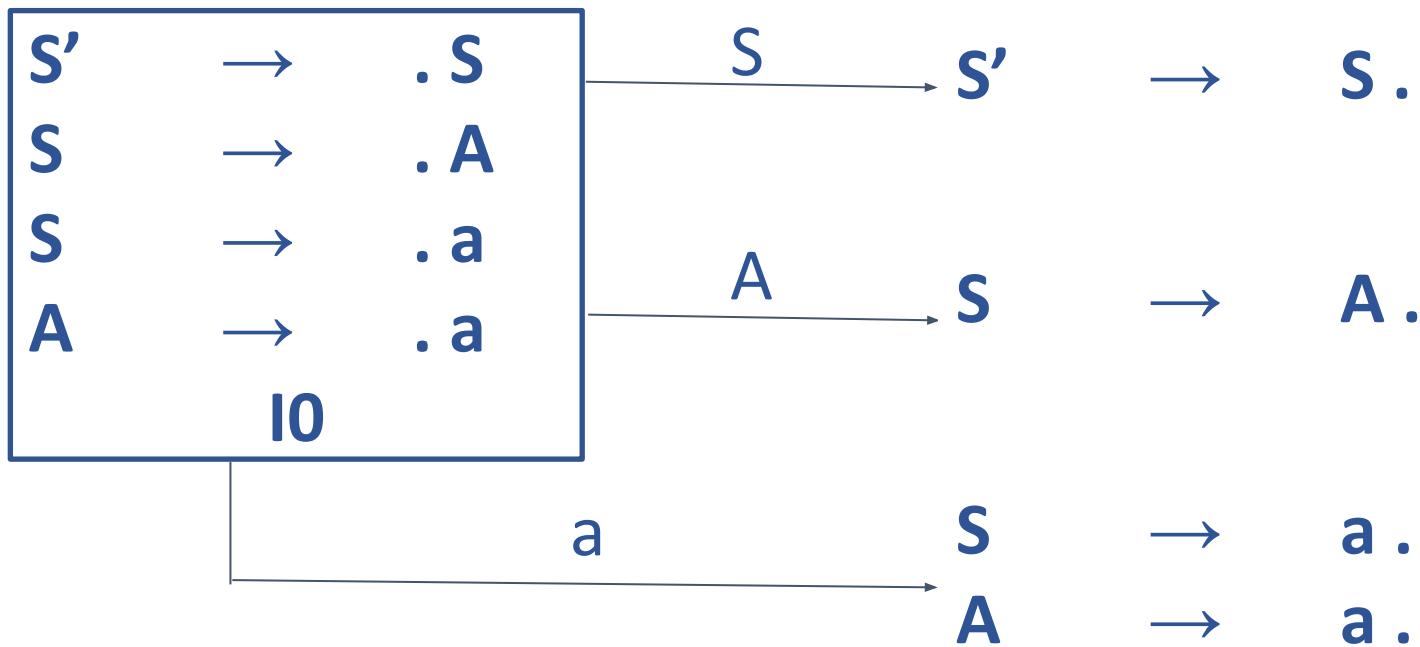
Production rules:

$$\begin{array}{l} S' \rightarrow S \\ 1. S \rightarrow A \\ 2. S \rightarrow a \\ 3. A \rightarrow a \end{array}$$

Step 4: Making closure

S'	\rightarrow	. S
S	\rightarrow	. A
S	\rightarrow	. a
A	\rightarrow	. a
		 IO

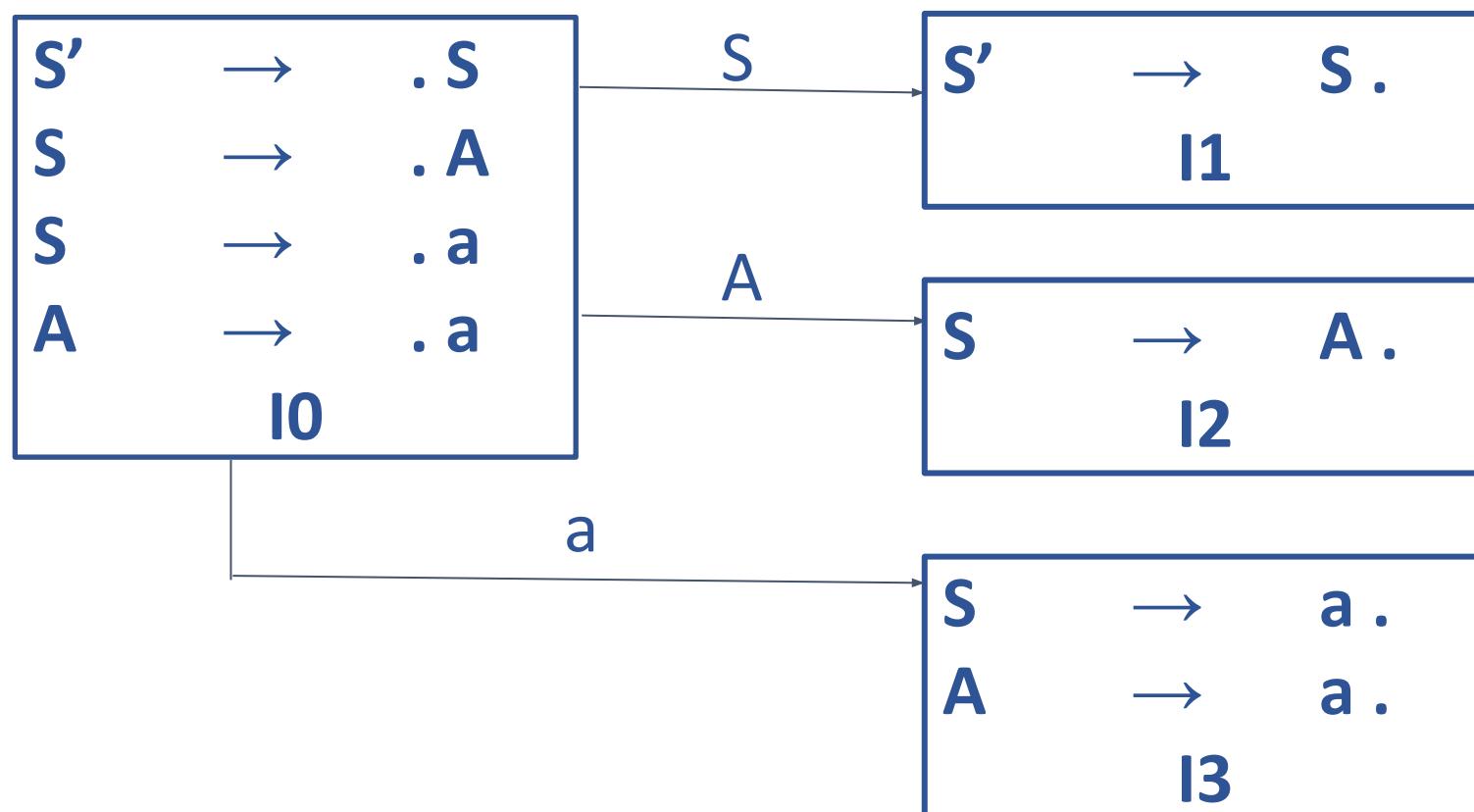
Step 5: Transitions using the goto function



Production rules:

S'	\rightarrow	S
1. S	\rightarrow	A
2. S	\rightarrow	a
3. A	\rightarrow	a

Step 6: Repeat steps 2-5 until no more states can be added



Production rules:

$S' \rightarrow S$	$1.$	$S \rightarrow A$	
$2.$	$S \rightarrow a$	$3.$	$A \rightarrow a$

State	Action		Goto	
	a	\$	s	A
0	s_3		1	2

State	Action		Goto	
	a	\$	s	A
0	s_3		1	2
1		Accept		

State	Action		Goto	
	a	\$	S	A
0	s_3		1	2
1		Accept		
2	r_1	r_1		
3	r_2 / r_3	r_2 / r_3		

The grammar is not in LR(0) family since there exist 2 r/r (reduce/reduce) conflicts.

Recall that:
A reduce-reduce conflict occurs when there are two production rules such that the top-of-stack exactly matches the right-hand-side of both the rules.

Is the following grammar in LR(0)?

$$S \rightarrow A a A b \mid B b B a$$

$$A \rightarrow \lambda$$

$$B \rightarrow \lambda$$

Is the following grammar in LR(0)?

$$S \rightarrow AaAb \mid BbBa$$

$$A \rightarrow \lambda$$

$$B \rightarrow \lambda$$

Solution:

Augmenting and numbering the grammar-

- | | |
|----------------------------|--|
| $S' \rightarrow S$ | |
| 1. $S \rightarrow AaAb$ | |
| 2. $S \rightarrow BbBa$ | |
| 3. $A \rightarrow \lambda$ | |
| 4. $B \rightarrow \lambda$ | |

Is the grammar
ambiguous?

No?

Construct LR(0)
automata and
check.

Steps 1-4:

S'	\rightarrow	.	S
S	\rightarrow	.	$A a A b$
S	\rightarrow	.	$B b B a$
A	\rightarrow	.	
B	\rightarrow	.	

IO

STOP!

There is a conflict here since there are two final items in the state (reduce/reduce conflict).

Production rules:

S'	\rightarrow	S
1. S	\rightarrow	$A a A b$
2. S	\rightarrow	$B b B a$
3. A	\rightarrow	λ
4. B	\rightarrow	λ

Conclusion:

Since there are two final items, this grammar does not belong to LR(0).

A snapshot of the table may be given.

State	Action		Goto		
	a	\$	S	A	B
0	r_3/r_4	r_3/r_4			

There are only two strings that can be generated using this grammar: “ab” and “ba”.

Even though the grammar is not ambiguous, there is a problem. It is because the problem is in LR(0). The LR(0) parser is not looking ahead into what symbol is on the input buffer.

Stack

\$

I/p Buffer

a b \$

The first symbol in the input buffer is “a”. So, we must reduce $A \rightarrow \lambda$.

If the first symbol was “b”, then we must reduce $B \rightarrow \lambda$.

So, the problem is with the parser. It takes decisions without checking properly. The parser is weak. If we look-ahead, it will work properly - CLR / LR(1).

Production rules:

	$S' \rightarrow S$
1.	$S \rightarrow AaAb$
2.	$S \rightarrow BbBa$
3.	$A \rightarrow \lambda$
4.	$B \rightarrow \lambda$

Is the following grammar in LR(0)?

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

$$T \rightarrow id$$

Is the following grammar in LR(0)?

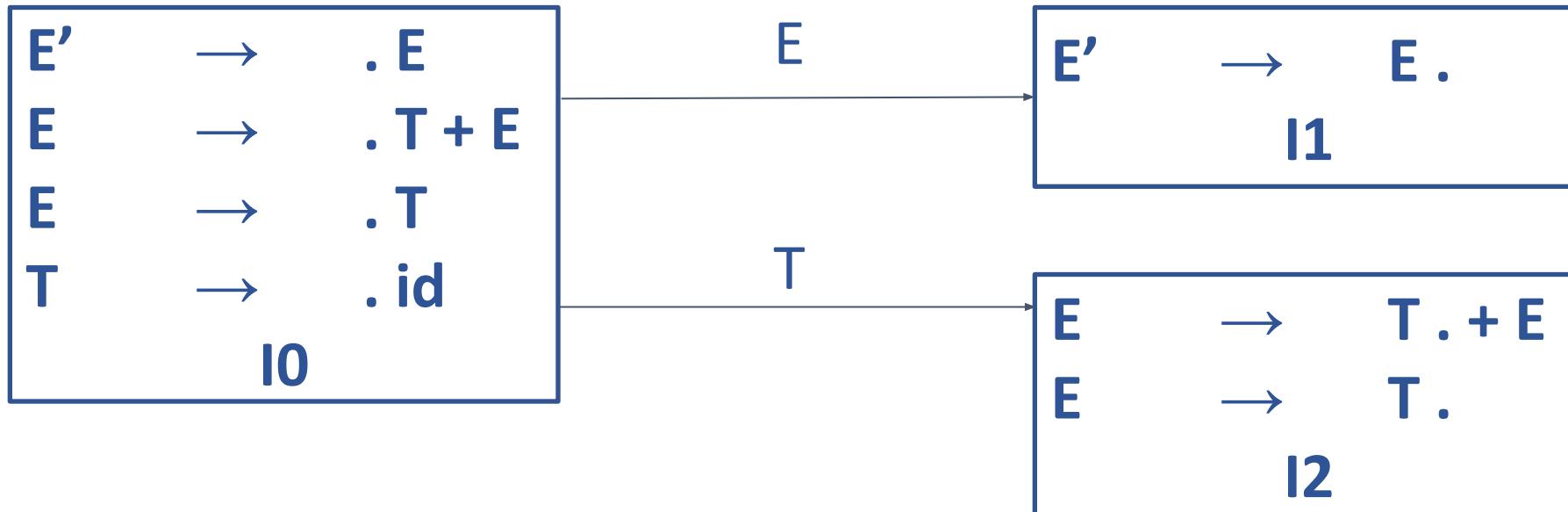
$$E \rightarrow T + E \mid T$$

$$T \rightarrow id$$

Solution:

Augmenting and numbering the grammar-

- | | |
|----|-----------------------|
| 1. | $E' \rightarrow E$ |
| 2. | $E \rightarrow T + E$ |
| 3. | $E \rightarrow T$ |
| | $T \rightarrow id$ |



Notice that, in the state I_2 , there is a shift/reduce conflict. The state can transition over “+” to a new state, based on the first item. Or, a reduction can happen using rule 2, since the second item in the state is a final item.

Production rules:

$E' \rightarrow E$	$E \rightarrow T + E$
1. $E \rightarrow T$	$E \rightarrow T$
2. $T \rightarrow id$	

A snapshot of the table is given below:

State	Action			Goto	
	id	+	\$	E	T
2	r ₂	s ₄ /r ₂	r ₂		

Conclusion:

Since there is a shift/reduce conflict on “+” in the state I2,
the grammar is not in LR(0).

But, is it really a problem? This grammar can generate:

id

id + id

id + id + id ...

- When we are working with just “id”, the next symbol upon shift is ‘\$’ and if we just look-ahead to check if it is ‘\$', we can simply reduce it.
- If there was a “+” as the next symbol on the buffer, then it can be shifted.
- LR(1) works better in such cases.

- LR(0) is the simplest technique in the LR family. Although that makes it the easiest to learn, these parsers are too weak to be of practical use for anything but a very limited set of grammars.
- The fundamental limitation of LR(0) is the zero, meaning no lookahead tokens are used. It is a stifling constraint to have to make decisions using only what has already been read, without even glancing at what comes next in the input.
- If we could peek at the next token and use that as part of the decision making, we will find that it allows for a much larger class of grammars to be parsed.



THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu



Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Lavitra Kshitij Madan

Compiler Design

Unit 2

Table-driven BUP

Preet Kanwal
Department of Computer Science & Engineering

In this lecture, we will learn about:

- SLR Parsers
- Follow set
- Comparison between LR(0) and SLR
- Conflicts in LR(0) and SLR parsers
- Viable prefix and its importance

- Considering SLR(1), where the S stands for simple. SLR(1) parsers use the same LR(0) automata and items and have the same table structure and parser operation, so everything you have already learned about LR(0) applies here.
- The difference comes in assigning table actions, where we are going to use one token of lookahead to help arbitrate among the conflicts.
- If we think back to the kind of conflicts we encountered in LR(0) parsing, it was the reduce actions that cause us grief. A state in an LR(0) parser can have at most one reduce action and cannot have both shift and reduce instructions.

- Since a reduce is indicated for any final item, this dictates that each final item must be in a state by itself.
- But let's revisit the assumption that if the item is complete, the parser must choose to reduce. Is that always appropriate? If we peeked at the next upcoming token, it may tell us something that invalidates that reduction.
- Basically, it says that we should not blindly put the 'reduce' move in the entire action part of the parsing table, but perform some sort of lookahead and see what is next in the buffer and then take the decision!

How does it look-ahead?

It places the reduce move only in the set **follow(LHS)**.

The set **follow(X)** is defined as the set of all terminals that follow X in the production rules.

Consider the grammar from the previous example (Example 18.3 of previous lecture):

$$\begin{array}{lll} E' & \rightarrow & E \\ 1. \quad E & \rightarrow & T + E \\ 2. \quad E & \rightarrow & T \\ 3. \quad T & \rightarrow & id \end{array}$$

Recall that there was a shift/reduce conflict in state I2:

$E \rightarrow T . + E$
$E \rightarrow T .$
I2

Follow(E) is only '\$' since it is the start symbol.

The reduce move is placed only under those terminals in the table that are a part of this follow set.

A snapshot of the SLR table is:

State	Action			Goto	
	id	+	\$	E	T
2		s_4	r_2		

Production rules:

$E' \rightarrow E$	
1. $E \rightarrow T + E$	
2. $E \rightarrow T$	
3. $T \rightarrow id$	

There is no shift-reduce conflict now for that particular state.

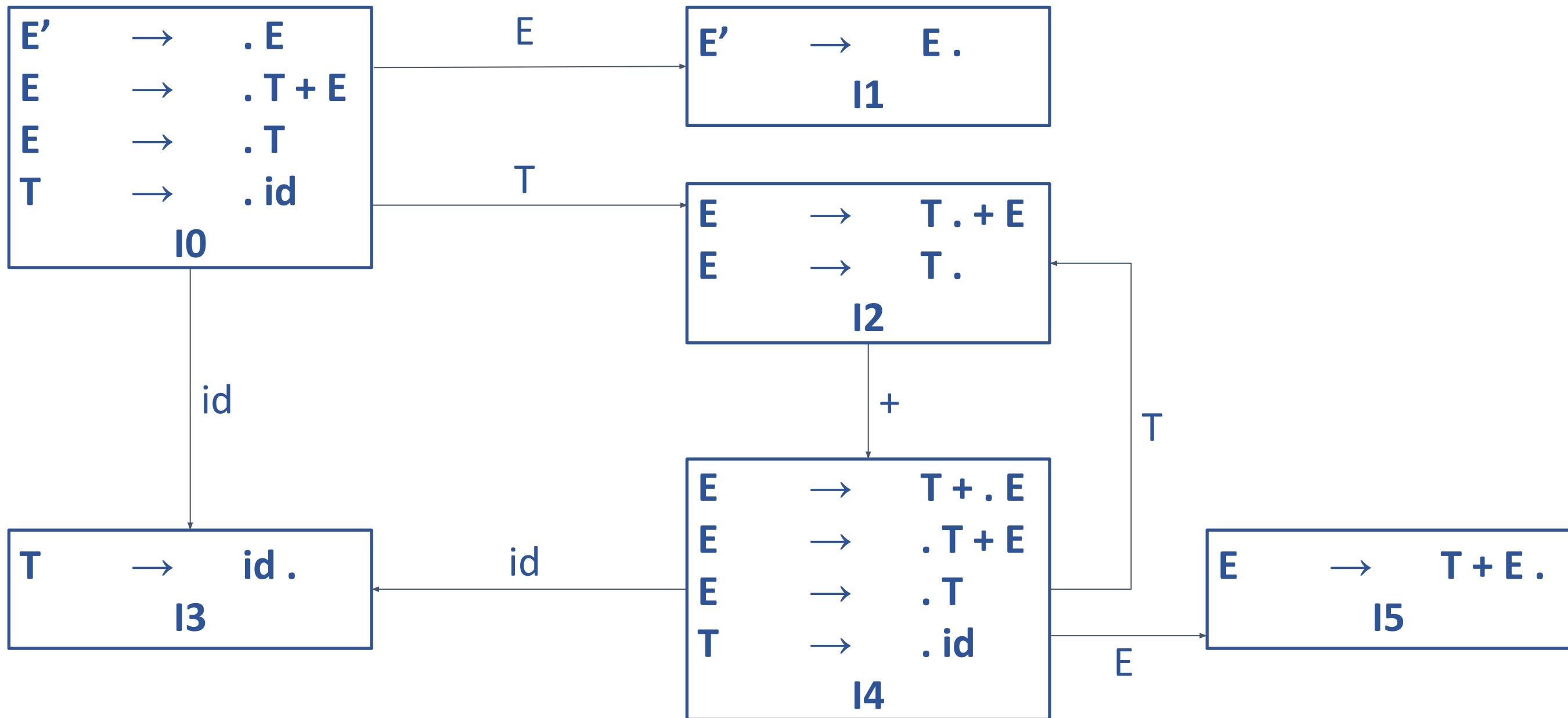
Let us continue building the solution to check if this grammar belongs to SLR or not

Continuing similarly, $\text{follow}(T) = \{+, \$\}$ because “+” follows ‘T’ in the second production rule and “\$” follows ‘T’ in the third rule.

Production rules:

$E' \rightarrow E$	$E \rightarrow T + E$
1. $E \rightarrow T + E$	$T \rightarrow id$
2. $E \rightarrow T$	

Let us finish building the LR(0) automata.



Compiler Design

Example (continued)

Let us now build the SLR parsing table.

State	Action			Goto	
	id	+	\$	E	T
0	s_3			1	2
1			Accept		
2		s_4	r_2		
3		r_3	r_3		
4	s_3			5	2
5			r_1		

Production rules:

- $E' \rightarrow E$
- 1. $E \rightarrow T + E$
- 2. $E \rightarrow T$
- 3. $T \rightarrow id$

A blank in the parsing table corresponds to error.

SLR(1) detects errors faster than LR(0) as SLR(1) parsing table has more blanks than corresponding LR(0) parsing table.

The simple improvement that SLR(1) makes on the basic LR(0) parser is to reduce only if the next input token is a member of the follow set of the non-terminal being reduced.

When filling in the table, we don't assume a reduce on all inputs as we did in LR(0), we selectively choose the reduction only when the next input symbols in a member of the follow set.

Let us now parse the string:

w = “id + id”

using the parsing table we have developed as part of the example.

The initial configuration is as follows:

Stack	I/p buffer	Action
\$ 0	id + id \$	

Let us start parsing the string step-by-step.

Compiler Design

Parsing a string

Stack	I/p buffer	Action
\$ 0	id + id \$	A[0, id] = s_3 Push 'id', 3 onto stack
\$ 0 id 3	+ id \$	

Compiler Design

Parsing a string

Stack	I/p buffer	Action
\$ 0	id + id \$	$A[0, \text{id}] = s_3$ Push 'id', 3 onto stack
\$ 0 id 3	+ id \$	$A[3, +] = r_3$ ($T \rightarrow \text{id}$) Pop 3, 'id' from stack Push 'T' onto stack Push Goto[0, T] = 2
\$ 0 T 2	+ id \$	

Compiler Design

Parsing a string

Stack	I/p buffer	Action
\$ 0	id + id \$	$A[0, \text{id}] = s_3$ Push 'id', 3 onto stack
\$ 0 id 3	+ id \$	$A[3, +] = r_3 (T \rightarrow \text{id})$ Pop 3, 'id' from stack Push 'T' onto stack Push Goto[0, T] = 2
\$ 0 T 2	+ id \$	$A[2, +] = s_4$ Push '+', 4 onto stack
\$ 0 T 2 + 4	id \$	

Stack	I/p buffer	Action
\$ 0	id + id \$	$A[0, \text{id}] = s_3$ Push 'id', 3 onto stack
\$ 0 id 3	+ id \$	$A[3, +] = r_3 (T \rightarrow \text{id})$ Pop 3, 'id' from stack Push 'T' onto stack Push Goto[0, T] = 2
\$ 0 T 2	+ id \$	$A[2, +] = s_4$ Push '+', 4 onto stack
\$ 0 T 2 + 4	id \$	$A[4, \text{id}] = s_3$ Push 'id', 3 onto stack
\$ 0 T 2 + 4 id 3	\$	

Compiler Design

Parsing a string



Stack	I/p buffer	Action
:	:	:
\$ 0 T 2 + 4 id 3	\$	A[3, \$] = r ₃ (T → id) Pop 3, 'id' from stack Push 'T' onto stack Push Goto[4, T] = 2
\$ 0 T 2 + 4 T 2	\$	

Stack	I/p buffer	Action
⋮ ⋮	⋮	⋮
\$ 0 T 2 + 4 id 3	\$	$A[3, \$] = r_3 (T \rightarrow id)$ Pop 3, 'id' from stack Push 'T' onto stack Push Goto[4, T] = 2
\$ 0 T 2 + 4 T 2	\$	$A[2, \$] = r_2 (E \rightarrow T)$ Pop 2, 'T' from stack Push 'E' onto stack Push Goto[4, E] = 5
\$ 0 T 2 + 4 E 5	\$	

Compiler Design

Parsing a string

Stack	I/p buffer	Action
:	:	:
\$ 0 T 2 + 4 E 5	\$	A[5, \$] = r ₁ (E → T + E) Pop 5, 'E', 4, '+', 2, 'T' Push 'E' onto stack Push Goto[0, E] = 1
\$ 0 E 1	\$	

Compiler Design

Parsing a string

Stack	I/p buffer	Action
⋮	⋮	⋮
\$ 0 T 2 + 4 E 5		\$ A[5, \$] = r ₁ (E → T + E) Pop 5, 'E', 4, '+', 2, 'T' Push 'E' onto stack Push Goto[0, E] = 1
\$ 0 E 1	\$	A[1, \$] = Accept

Similarities between LR(0) and SLR parsers:

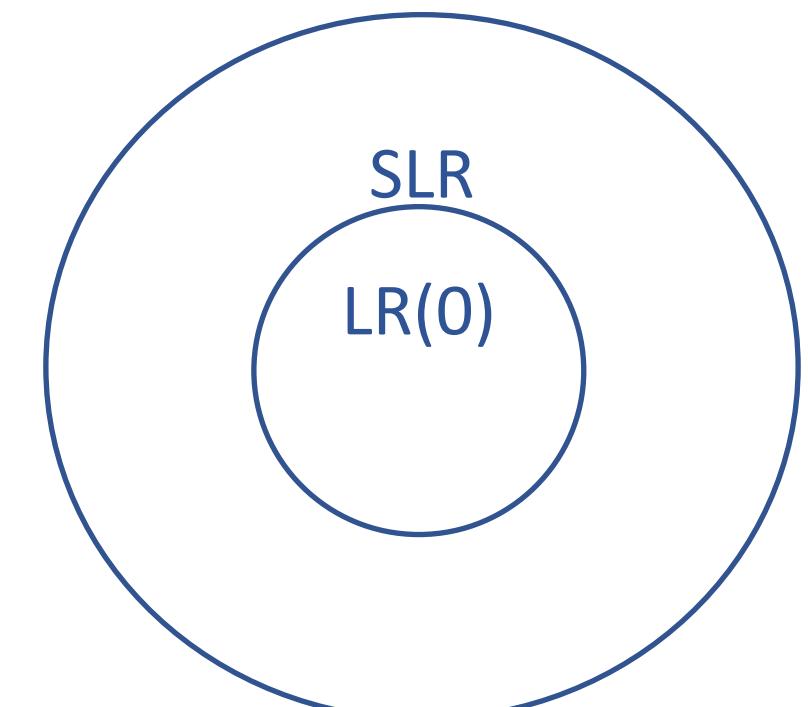
- Both use LR(0) automata
- Same way to fill the Goto part for the shift move in the parsing table.

The only difference between the two:

- Where to place the reduce move

The addition of just one token of lookahead and use of the follow set greatly expands the class of grammars that can be parsed without conflict.

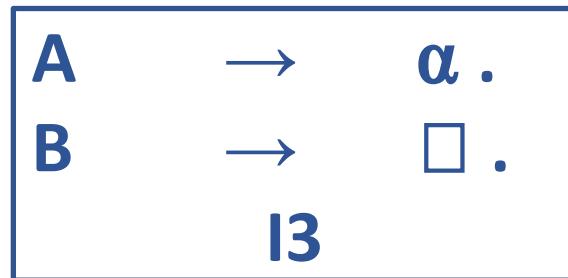
If the grammar is in LR(0), it is definitely in SLR and if it is in SLR, it may or may not be in LR(0).



Conflicts in LR(0) and SLR(1)

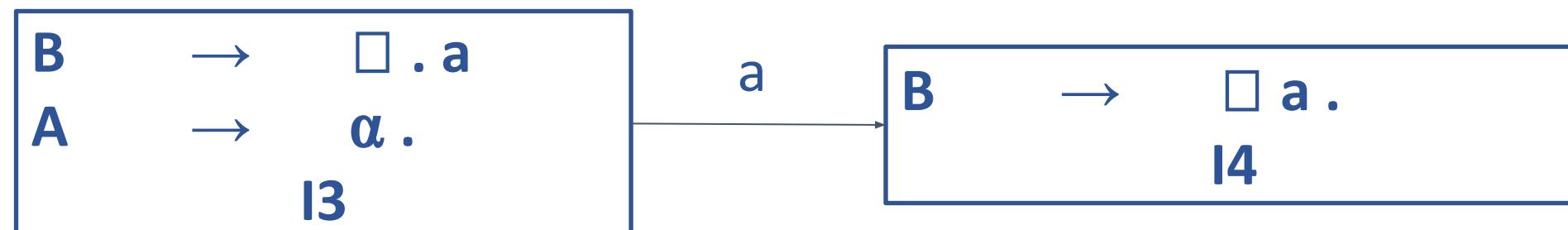
There exist a r/r conflict in LR(0) if and only if there are two or more final items in a state.

Example:



A shift/reduce (s/r) conflict occurs in LR(0) when there is a final item and a shift on a terminal.

Example:



Compiler Design

Conflicts in SLR Parsers



There exist a r/r conflict in SLR if and only if there are two or more final items in a state and the intersection of the follow sets of the left-hand-sides of both the items is not empty.

Example:

A	→	$\alpha .$
B	→	$\square .$
I3		

If f (There is something in common),
then there exists a reduce-reduce conflict.

Compiler Design

Conflicts in SLR Parsers

A shift/reduce (s/r) conflict occurs in SLR when there is a final item and a shift on a terminal, and the follow of the left-hand-side of the final item contains that terminal which may be shifted next.

Example:



If $\text{follow}(A)$ contains 'a', then there is a shift-reduce conflict.

Viable Prefix

Recall the shift-reduce parser example

The string was accepted by the grammar.

Stack	I/p buffer	Action
\$	id * id \$	Shift id
\$ id	* id \$	Reduce using (iii)
\$ F	* id \$	Reduce using (ii)
\$ T	* id \$	Shift *
\$ T *	id \$	Shift id
\$ T * id	\$	Reduce using (iii)
\$ T * F	\$	Reduce using (ii)
\$ T	\$	Reduce using (i)
\$ E	\$	Accept

Grammar:

- (i) $E \rightarrow E + T \mid T$
- (ii) $T \rightarrow T * F \mid F$
- (iii) $F \rightarrow id$

w = "id * id"

Compiler Design

Viable Prefix

We observe that at any point of time, the stack contents must be a prefix of a right sentential form. However, not all prefixes of a right sentential form can appear on the stack.

For example, consider the rightmost derivation:

Rightmost derivation of $\text{id}^* \text{id}$	Set of prefixes of a right sentential form	Viable Prefixes
$E \rightarrow T$	T	T
$\rightarrow T * F$	T, T *, T * F	T, T *, T * F
$\rightarrow T * \text{id}$	T, T *, T * id	T, T *, T * id
$\rightarrow F * \text{id}$	F, F *, F * id	F
$\rightarrow \text{id} * \text{id}$	id, id *, id * id	id

Compiler Design

Viable Prefix



Here, “**id ***” is a prefix of a right sentential form. But it can never appear on the stack!

This is because we will always reduce by $F \rightarrow id$ before shifting “*”.

The prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called **viable prefixes**. Its a building block for recognizing handles.

By definition, a **viable prefix** is a prefix of a right sentential form that does not continue past the right end of the rightmost handle of that sentential form. It's a **viable prefix** because it is a prefix of the handle.

We will see the importance of viable prefixes in the upcoming lectures when we learn about more powerful parsers.

The entire parsing algorithm is based on the idea that the LR(0) automaton can recognize viable prefixes and reduce them appropriately.

Recognizing Viable Prefixes Idea: To recognize viable prefixes, we must

- Recognize a sequence of partial RHSs of productions, where
- Each sequence can eventually reduce to part of the missing suffix of its predecessor.



THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu



Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Philip Joseph

Compiler Design

Unit 2

Table-driven BUP

Preet Kanwal
Department of Computer Science & Engineering

Compiler Design

Lecture Overview



In this lecture, we will practice -

- Questions related to different Bottom up parsing techniques covered so far.

Provide a production with the shortest RHS that will introduce s/r conflict in SLR parser for the following grammar -

$$\begin{array}{lll} S & \rightarrow & A b \\ A & \rightarrow & c b A d \\ A & \rightarrow & c A d \\ A & \rightarrow & \lambda \end{array}$$

Add a production

$$A \rightarrow b \mid d$$

to the given grammar -

$$S \rightarrow A b$$

$$A \rightarrow c b A d$$

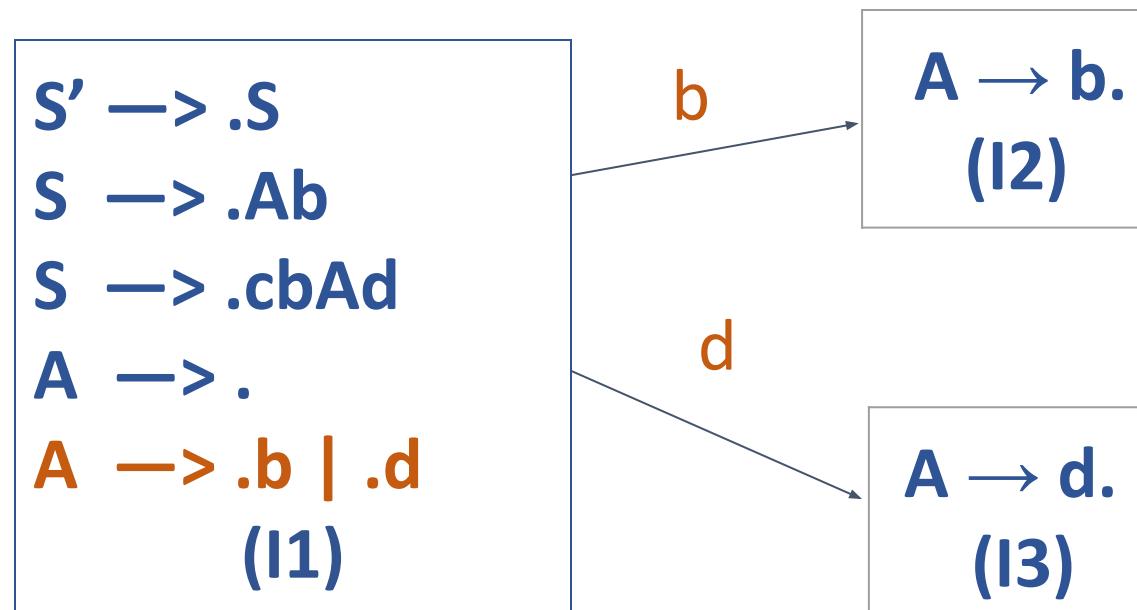
$$A \rightarrow c A d$$

$$A \rightarrow \lambda$$

$$A \rightarrow b \mid d$$

To ensure an s/r conflict, the terminal shift from I1 should be '**b**' and '**d**' to satisfy the condition for s/r conflict in SLR parser because I0 has a final state $A \rightarrow .$ and **Follow(A) = { b , d }** should contain that terminal '**b**' and '**d**' which may be shifted next.

If **follow(A)** contains '**b**' or '**d**', then there is a shift-reduce conflict.



Identify whether the given grammar is LL(1), LR(0), or SLR(1)?

$$X \rightarrow Y z \mid a$$

$$Y \rightarrow b z \mid \lambda$$

$$Z \rightarrow \lambda$$

- To check if a grammar is LL(1), one option is to construct the LL(1) parsing table and check for any conflicts.
- Build the FIRST and FOLLOW sets for each of the nonterminals. Here, we get -
 - $\text{FIRST}(X) = \{ a, b, z \}$
 - $\text{FIRST}(Y) = \{ b, \lambda \}$
 - $\text{FIRST}(Z) = \{ \lambda \}$
- FOLLOW sets are given by
 - $\text{FOLLOW}(X) = \{ \$ \}$
 - $\text{FOLLOW}(Y) = \{ z \}$
 - $\text{FOLLOW}(Z) = \{ z \}$

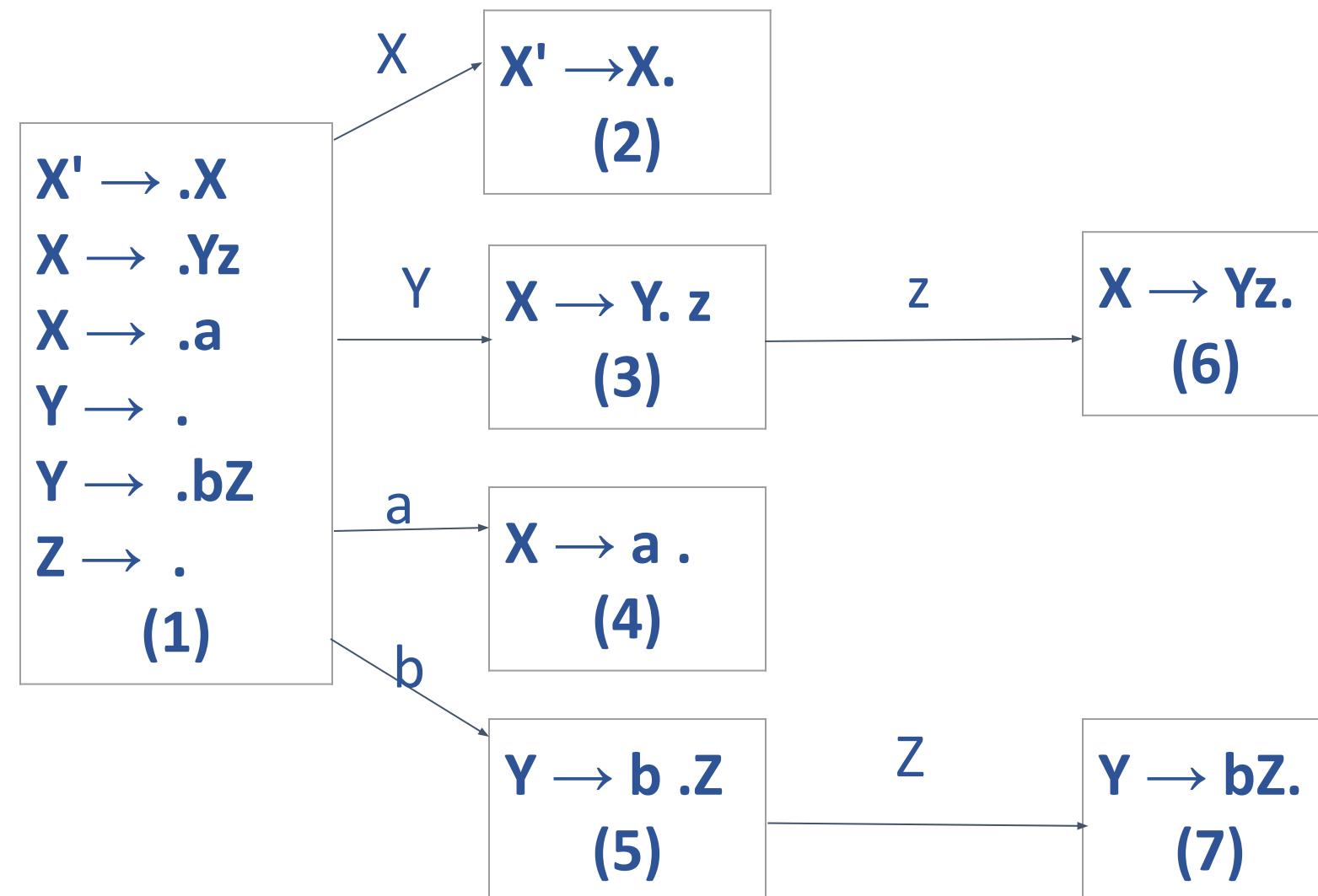
Build the following LL(1) parsing table -

	a	b	z	\$
X	$X \rightarrow a$	$X \rightarrow Yz$	$X \rightarrow Yz$	
Y		$Y \rightarrow bz$	$Y \rightarrow \lambda$	
Z			$Z \rightarrow \lambda$	

Since we can build this parsing table with no conflicts, the grammar is in LL(1).

To check if a grammar is LR(0) or SLR(1), we begin by building up all of the LR(0) sets for the grammar.

In this case, assuming that X is the start symbol, we get the following -



- We can see that the grammar is not in LR(0) because there is a shift-reduce conflict in state (1).
- This is because, since there is a shift item $X \rightarrow .a$ and final item $Y \rightarrow .$, we can't tell whether to shift the 'a' or reduce the empty string.
- More generally, no grammar with ϵ -productions is LR(0).
- However, there remains the possibility that grammar might be SLR(1).

- To check if the grammar is in SLR(1), we augment each reduction with the lookahead set for the particular nonterminals.
- The shift/reduce conflict in state (1) has been eliminated in SLR(1) because we only reduce when the lookahead is ‘z’, which doesn't conflict with any of the other items.
- The reduce/reduce conflict in state (1) is not eliminated in SLR(1) because $\text{follow}(Y) \cap \text{follow}(Z) \neq \emptyset$
- Thus this grammar is not in SLR(1).



THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu



Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Philip Joseph

Compiler Design

Unit 2

Table-driven BUP

Preet Kanwal
Department of Computer Science & Engineering

Compiler Design

Lecture Overview



In this lecture, we will practice -

- Questions related to LR(0) and SLR Parsers.

Let us try to solve the following problem step-by-step.

Construct LR(0) automata and LR(0) & SLR parsing tables for the following grammar.

$$S \rightarrow d A \mid a B$$

$$A \rightarrow b A \mid c$$

$$B \rightarrow b B \mid c$$

Recall the steps for constructing the LR automaton:

1. Augment the grammar
2. Number your productions in the original grammar
3. Follow the LR(0) Parsing Algorithm

Augmenting and numbering the grammar

S' → **S**

1. **S** → **d A**

2. **S** → **a B**

3. **A** → **b A**

4. **A** → **c**

5. **B** → **b B**

6. **B** → **c**

Recall the LR(0) Parsing Algorithm

1. Start with the initial item.
2. If there is a non-terminal after the “.” in any of the productions, add all the productions from that non-terminal, i.e., where that non-terminal is on the left-hand-side.
3. For every newly added item, repeat step 2.
4. If no more items can be added, it becomes a closure/state.
5. Use the goto function to define a transition for each symbol after the dot.
6. Repeat the process until no more new states can be added to the automaton.

Step 1: Starting

Starting with the initial item.

$$S' \rightarrow . S$$

Step 2: Adding productions of NTs after the dot

Since "S" follows the dot, we put the first production.

$$S' \rightarrow . S$$

$$S \rightarrow . d A$$

$$S \rightarrow . a B$$

Step 3: Repeating step 2 until no more items

There are no productions that have a non-terminal immediately following a dot.

Production rules:

$$\begin{array}{lll} S' & \rightarrow & S \\ 1. S & \rightarrow & d A \\ 2. S & \rightarrow & a B \\ 3. A & \rightarrow & b A \\ 4. A & \rightarrow & c \\ 5. B & \rightarrow & b B \\ 6. B & \rightarrow & c \end{array}$$

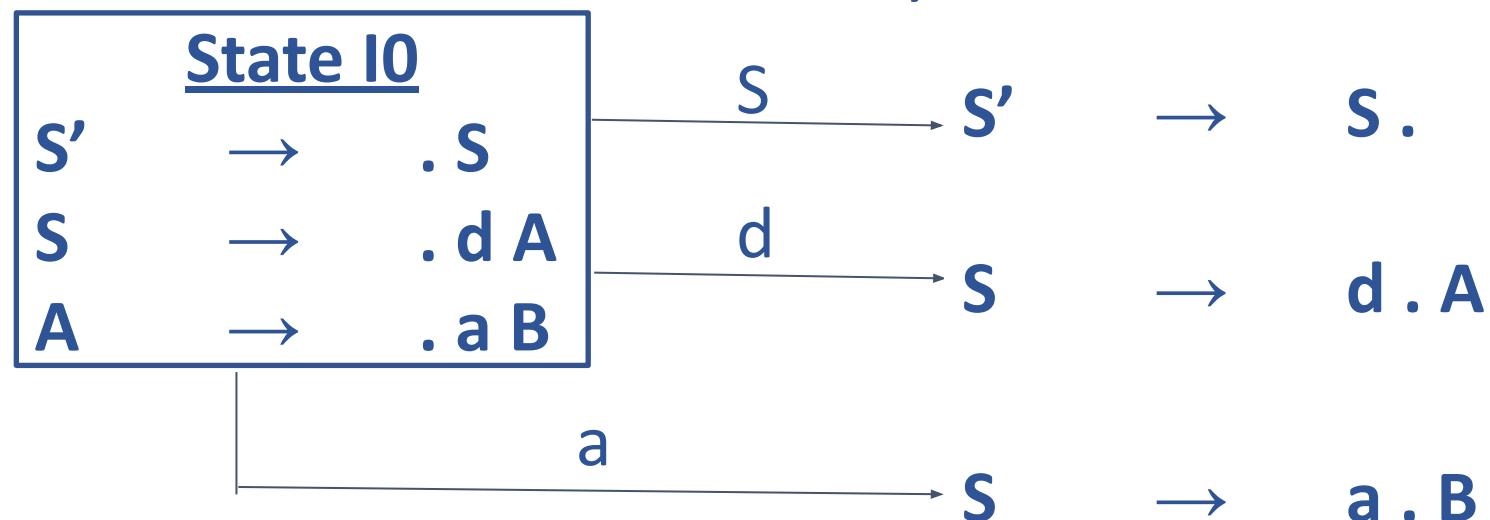
Step 4: Making closure

Since no more items can be added to the state, we can now make it a closure. It is done by adding the productions in a box and naming it.

State I0	
S'	→ . S
S	→ . d A
A	→ . a B

Step 5: Transitions using the goto function

We will have transitions on “S”, “d” and “a”.



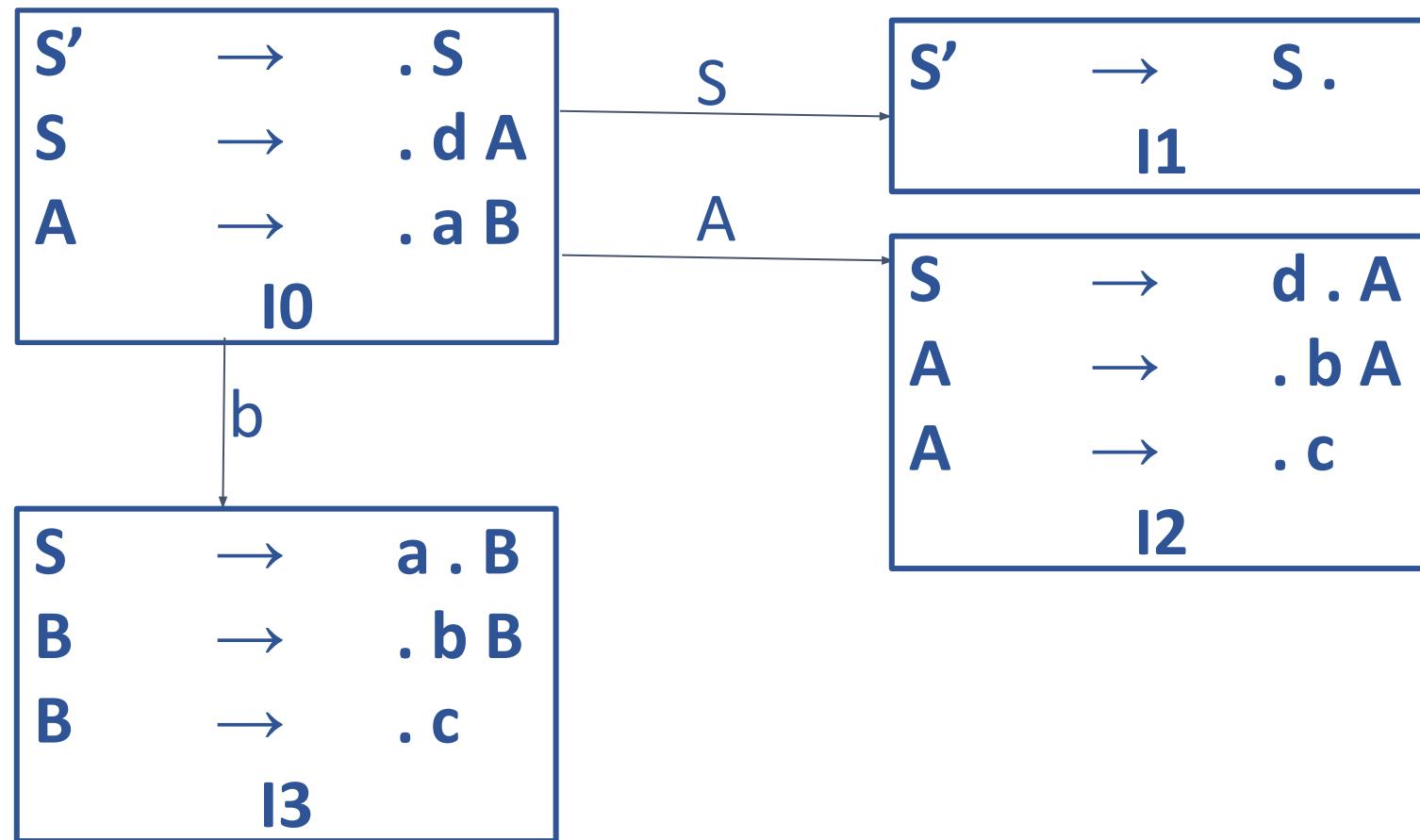
Production rules:

S'	→	S
1. S	→	d A
2. S	→	a B
3. A	→	b A
4. A	→	c
5. B	→	b B
6. B	→	c

Step 6: Repeat steps 2-5 until no more states can be added

$S' \rightarrow S.$ is already completed and closure can be used on it.

The other two transitions require steps 2-5.



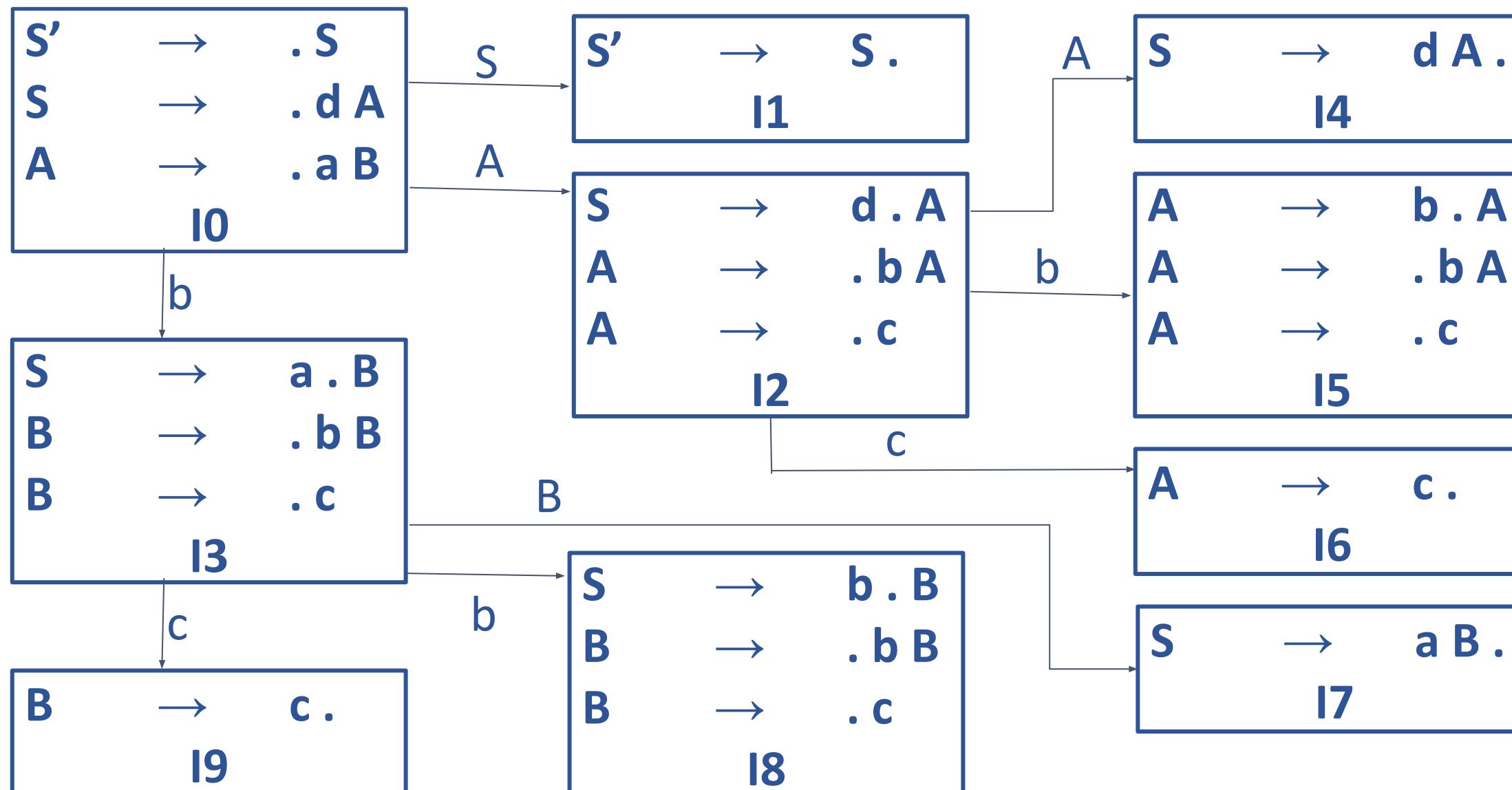
Production rules:

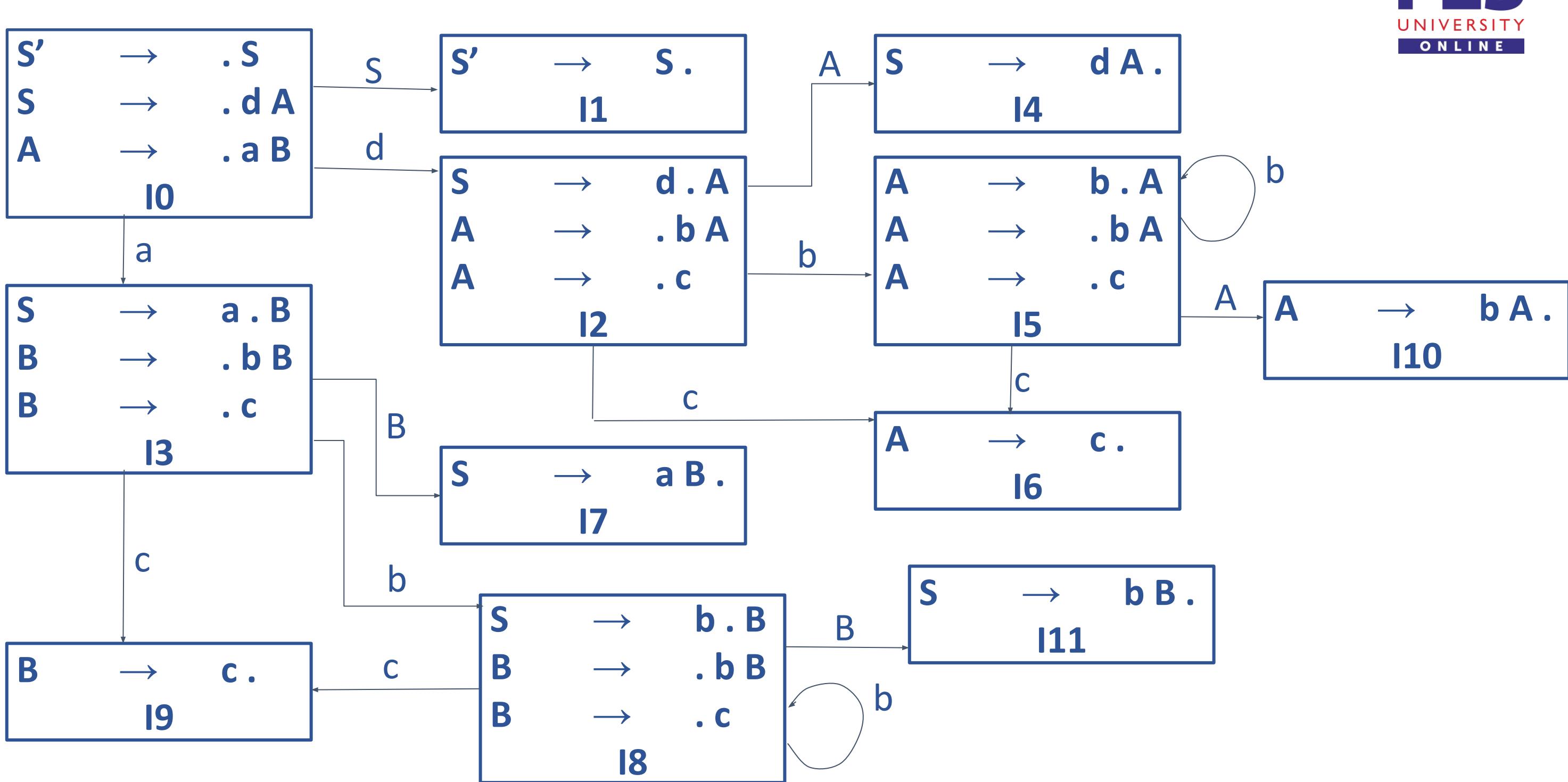
1.	$S' \rightarrow S$
2.	$S \rightarrow d A$
3.	$S \rightarrow a B$
4.	$A \rightarrow b A$
5.	$A \rightarrow c$
6.	$B \rightarrow b B$
7.	$B \rightarrow c$

Step 6: Repeat steps 2-5 until no more states can be added

$S' \rightarrow S.$ is already completed and closure can be used on it.

The other two transitions require steps 2-5.





State	Action					Goto		
	a	b	c	d	\$	S	A	B
0								

We will start filling the LR(0) Parsing Table now.

For the state I0, transition over “S” leads to state I1 and so, we fill the ‘S’ sub-column in the ‘goto’ column as “1”.

Similarly, transition over “d” and “a” lead to states I2 and I3 respectively. We mark them as s_2 and s_3 in the respective sub-columns of the ‘action’ column.

State	Action					Goto		
	a	b	c	d	\$	S	A	B
0	s_3			s_2		1		

State 1 contains the special final item. “Accept” should be put under the ‘\$’ sub-column in the ‘Action’ column.

State	Action					Goto		
	a	b	c	d	\$	S	A	B
0	s_3			s_2		1		
1					Accept			

Similarly filling for states I2, I3, I5 and I8.

State	Action					Goto		
	a	b	c	d	\$	S	A	B
0	s_3				s_2	1		
1					Accept			
2		s_5	s_6				4	
3		s_8	s_9					7
5		s_5	s_6				10	
8		s_8	s_9					11

States I4, I6 I7, I9, I10 and I11 have a final item each and so, we must reduce them. To reduce, we must place “ r_x ” in the entire action column for that state, where “X” takes the value of the production rule number.

State	Action					Goto		
	a	b	c	d	\$	S	A	B
0	s_3				s_2		1	
1						Accept		
2		s_5	s_6				4	
3		s_8	s_9					7
4	r_1	r_1	r_1	r_1	r_1			
5		s_5	s_6				10	
6	r_4	r_4	r_4	r_4	r_4			
:	:	:	:	:	:	:	:	:

Table continued on next slide

State	Action					Goto		
	a	b	c	d	\$	S	A	B
:	:	:	:	:	:	:	:	:
7	r_2	r_2	r_2	r_2	r_2			
8		s_8	s_9					11
9	r_6	r_6	r_6	r_6	r_6			
10	r_3	r_3	r_3	r_3	r_3			
11	r_5	r_5	r_5	r_5	r_5			

Note: These states are also called Canonical LR(0) collection.

$C = \{I0, I1, I2, \dots, I11\}$

State	Action					Goto		
	a	b	c	d	\$	S	A	B
0								

Let us also fill the SLR(1) Parsing Table now.

Let us build the follow table first.

$\text{follow}(S) = \$$

$\text{follow}(A) = \text{follow}(S) = \$$

$\text{follow}(B) = \text{follow}(S) = \$$

Production rules:

- | | | |
|------------------------|---------------|-------|
| $S' \rightarrow S$ | \rightarrow | S |
| 1. $S \rightarrow d A$ | \rightarrow | $d A$ |
| 2. $S \rightarrow a B$ | \rightarrow | $a B$ |
| 3. $A \rightarrow b A$ | \rightarrow | $b A$ |
| 4. $A \rightarrow c$ | \rightarrow | c |
| 5. $B \rightarrow b B$ | \rightarrow | $b B$ |
| 6. $B \rightarrow c$ | \rightarrow | c |

State	Action					Goto		
	a	b	c	d	\$	S	A	B
0	s_3				s_2		1	
1						Accept		
2		s_5	s_6				4	
3		s_8	s_9					7
4						r_1		
5		s_5	s_6					10
6						r_4		
:	:	:	:	:	:	:	:	:

Table continued on next slide

State	Action					Goto		
	a	b	c	d	\$	S	A	B
:	:	:	:	:	:	:	:	:
7					r_2			
8		s_8	s_9					11
9					r_6			
10					r_3			
11					r_5			

Let us now parse the string:

w = “abc”

using the LR(0) and SLR parsing tables we have developed as part of the example.

The initial configuration is as follows:

Stack	I/p buffer	Action
\$ 0	a b c \$	

Let us start parsing the string step-by-step.

Stack	I/p buffer	Action
\$ 0	a b c \$	$A[0, a] = s_3$ Push 'a', 3 onto stack
\$ 0 a 3	b c \$	$A[3, b] = s_8$ Push 'b', 8 onto stack
\$ 0 a 3 b 8	c \$	$A[8, c] = s_9$ Push 'c', 9 onto stack
\$ 0 a 3 b 8 c 9	\$	$A[9, \$] = r_6 (B \rightarrow c)$ Pop 9, 'c' from stack Push 'B' onto stack Push Goto[8, B] = 11
:	:	:

Production rules:

- $S' \rightarrow S$
- 1. $S \rightarrow d A$
- 2. $S \rightarrow a B$
- 3. $A \rightarrow b A$
- 4. $A \rightarrow c$
- 5. $B \rightarrow b B$
- 6. $B \rightarrow c$

Table continued on next slide

Stack	I/p buffer	Action
:	:	:
\$ 0 a 3 b 8 B 11	\$	$A[11, \$] = r_5 (B \rightarrow b B)$ Pop 11, 'B', 8, 'b' Push 'B' onto stack Push Goto[3, B] = 7
\$ 0 a 3 B 7	\$	$A[7, \$] = r_2 (S \rightarrow a B)$ Pop 7, 'B', 3, 'a' Push 'S' onto stack Push Goto[0, S] = 1
\$ 0 S 1	\$	$A[1, \$] = \text{Accept}$

Production rules:

- | | |
|------------------------|-----------------------|
| $S' \rightarrow S$ | $S \rightarrow d A$ |
| 1. $S \rightarrow a B$ | $a B \rightarrow b A$ |
| 2. $S \rightarrow b A$ | $b A \rightarrow c$ |
| 3. $A \rightarrow c$ | $b B \rightarrow c$ |
| 4. $A \rightarrow c$ | |
| 5. $B \rightarrow b B$ | |
| 6. $B \rightarrow c$ | |

The string has been successfully parsed. When the same string is parsed using SLR, we will get the same table.

Let us try to solve another problem.

Construct LR(0) automata and LR(0) & SLR parsing tables for the following grammar.

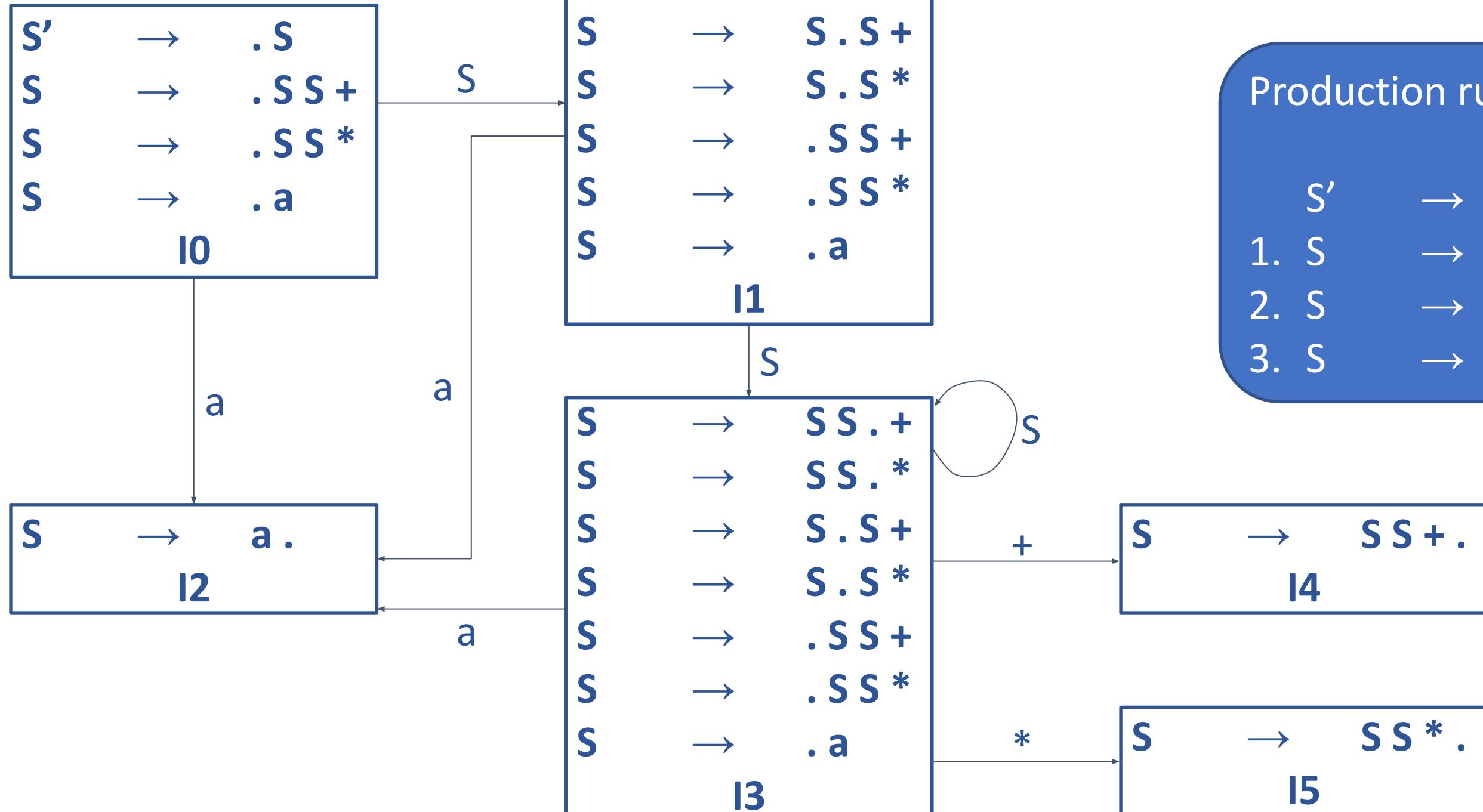
$$S \rightarrow SS+ \mid SS^* \mid a$$

Augmenting and numbering the grammar

$$\begin{array}{lll} S' & \rightarrow & S \\ 1. \ S & \rightarrow & S \ S \ + \\ 2. \ S & \rightarrow & S \ S \ * \\ 3. \ S & \rightarrow & a \end{array}$$

$\text{follow}(S) = \{\$, +, *, a\}$

Making the LR(0) automaton



Production rules:

1. $S' \rightarrow S$
2. $S \rightarrow S S^+$
3. $S \rightarrow S S^*$
4. $S \rightarrow a$

Making the LR(0) parsing table

State	Action				Goto
	+	*	a	\$	
0			s_2		1
1			s_2	Accept	3
2	r_3	r_3	r_3	r_3	
3	s_4	s_5	s_2		3
4	r_1	r_1	r_1	r_1	
5	r_2	r_2	r_2	r_2	

Production rules:

- 1. $S' \rightarrow S$
- 2. $S \rightarrow SS +$
- 3. $S \rightarrow SS *$
- 3. $S \rightarrow a$

Note that since the follow of S includes all terminals and '\$', the SLR parsing table will be exactly the same as LR(0) parsing table. The grammar belongs to both, LR(0) and SLR, since there are no conflicts!

Let us now parse the string:

w = "a a + a a *"

using the common parsing table we have developed as part of the example.

The initial configuration is as follows:

Stack	I/p buffer	Action
\$ 0	a a + a a * \$	

Let us start parsing the string step-by-step.

Parsing a string - Example 20.2

Stack	I/p buffer	Action
\$ 0	a a + a a * \$	A[0, a] = s_2 Push 'a', 2 onto stack
\$ 0 a 2	a + a a * \$	A[2, a] = r_3 ($S \rightarrow a$) Pop 2, 'a' from stack Push 'S' onto stack Push Goto[0, S] = 1
\$ 0 S 1	a + a a * \$	A[1, a] = s_2 Push 'a', 2 onto stack
\$ 0 S 1 a 2	+ a a * \$	A[2, +] = r_3 ($S \rightarrow a$) Pop 2, 'a' from stack Push 'S' onto stack Push Goto[1, S] = 3
:	:	:

Production rules:

- 1. $S' \rightarrow S$
- 2. $S \rightarrow S S +$
- 3. $S \rightarrow S S *$
- 3. $S \rightarrow a$

Parsing a string - Example 20.2

Stack	I/p buffer	Action
\$ 0 S 1 S 3	+ a a * \$	A[3, +] = s_4 Push '+', 4 onto stack
\$ 0 S 1 S 3 + 4	a a * \$	A[4, a] = r_1 ($S \rightarrow SS+$) Pop 4, '+', 3, 'S', 1, 'S' Push 'S' onto stack Push Goto[0, S] = 1
\$ 0 S 1	a a * \$	A[1, a] = s_2 Push 'a', 2 onto stack
\$ 0 S 1 a 2	a * \$	A[2, a] = r_3 ($S \rightarrow a$) Pop 2, 'a' from stack Push 'S' onto stack Push Goto[1, S] = 3
:	:	:

Production rules:

- 1. $S' \rightarrow S$
- 2. $S \rightarrow SS+$
- 3. $S \rightarrow SS^*$
- 3. $S \rightarrow a$

Compiler Design

Parsing a string - Example 20.2

Stack	I/p buffer	Action
\$ 0 S 1 S 3	a * \$	A[3, a] = s ₂ Push 'a', 2 onto stack
\$ 0 S 1 S 3 a 2	* \$	A[2, *] = r ₃ (S → a) Pop 2, 'a' from stack Push 'S' onto stack Push Goto[3, S] = 3
\$ 0 S 1 S 3 S 3	* \$	A[3, *] = s ₅ Push '*', 5
\$ 0 S 1 S 3 S 3 * 5	\$	A[5, \$] = r ₂ (S → SS *) Pop 5, '*', 3, 'S', 3, 'S' Push 'S' onto stack Push Goto[1, S] = 3
\$ 0 S 1 S 3	\$	A[3, \$] = Reject

Production rules:

- 1. $S' \rightarrow S$
- 2. $S \rightarrow SS +$
- 3. $S \rightarrow SS *$
- 3. $S \rightarrow a$

Construct the LR(0) and SLR parsing tables for the following grammar:

$$S \rightarrow L = R \mid R$$

$$L \rightarrow * R \mid id$$

$$R \rightarrow L$$

Augmenting and numbering the grammar

$S' \rightarrow S$

1. $S \rightarrow L = R$

2. $S \rightarrow R$

3. $L \rightarrow * R$

4. $L \rightarrow id$

5. $R \rightarrow L$

$follow(S) = \{\$\}$

$follow(L) = \{=, \$\}$

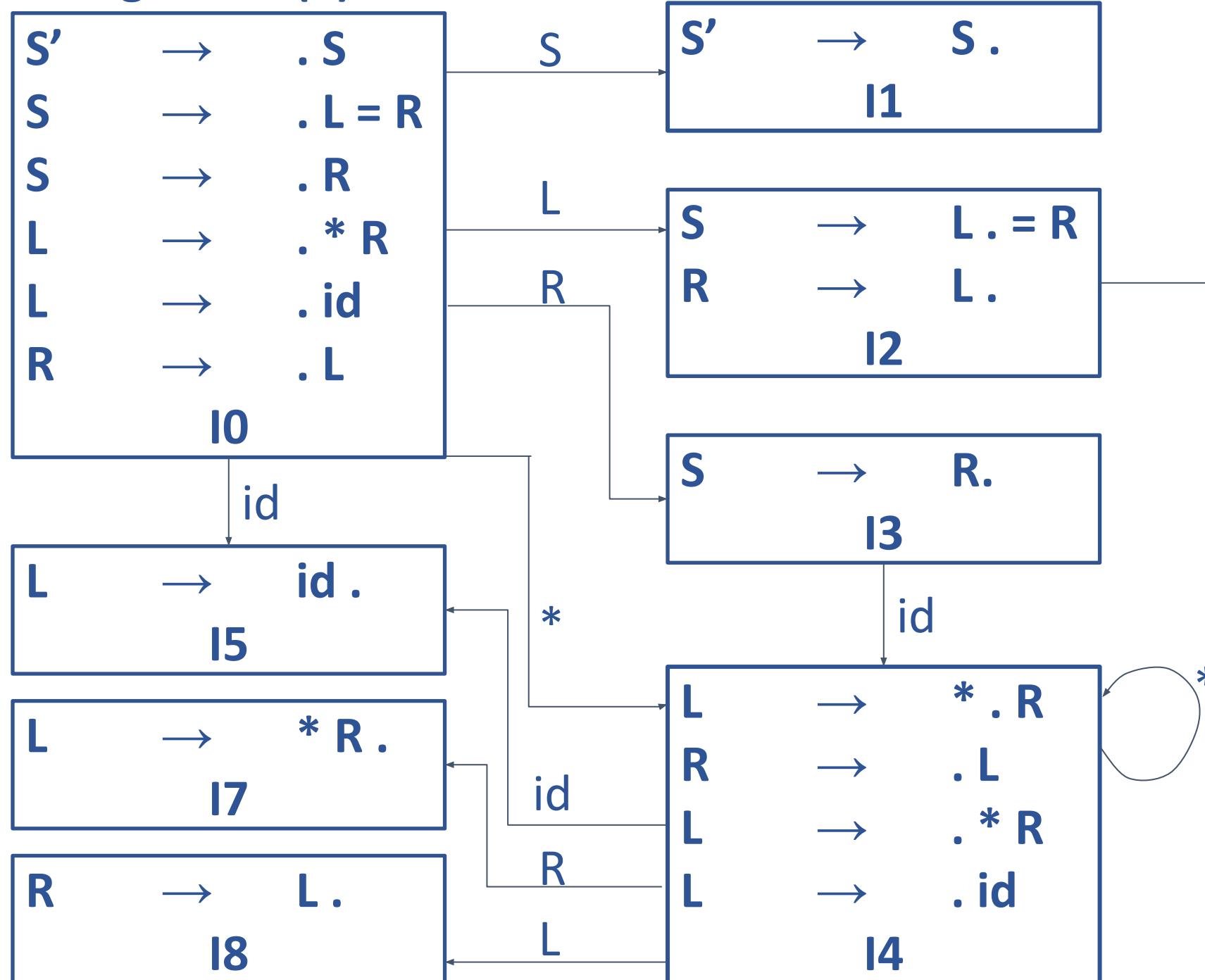
$follow(R) = \{\$, =\}$

S	L	R
\$	=	\$
	\$	=

Example 20.3 (continued)



Making the LR(0) automaton



Production rules:

$S' \rightarrow S$	$S \rightarrow L = R$
1. $S \rightarrow L = R$	$R \rightarrow .$
2. $S \rightarrow R$	$* R \rightarrow .$
3. $L \rightarrow *$	$R \rightarrow . * R$
4. $L \rightarrow id$	$R \rightarrow .id$
5. $R \rightarrow L$	$L \rightarrow .$

Compiler Design

Example 20.3 (continued)

Note that there are shift-reduce conflicts in states I2 and I3.

State	ACTION				GOTO		
	*	=	id	\$	S	L	R
0	s_4		s_5		1	2	3
1				Accept			
2	r_5	s_6 / r_5	r_5	r_5			

Production rules:

1. $S' \rightarrow S$
2. $S \rightarrow L = R$
3. $S \rightarrow R$
4. $L \rightarrow * R$
5. $L \rightarrow id$
6. $R \rightarrow L$

Let us, however, continue making the LR(0) parsing table

Compiler Design

Example 20.3 (continued)

State	ACTION				GOTO		
	*	=	id	\$	S	L	R
0	s_4		s_5		1	2	3
1				Accept			
2	r_5	s_6 / r_5	r_5	r_5			
3	r_2	r_2	s_4 / r_2	r_2			
4	s_4		s_5			8	7
5	r_4	r_4	r_4	r_4			
6	s_4		s_5			8	9
7	r_3	r_3	r_3	r_3			
8	r_5	r_5	r_5	r_5			
9	r_1	r_1	r_1	r_1			

Production rules:

- $S' \rightarrow S$
- 1. $S \rightarrow L = R$
- 2. $S \rightarrow R$
- 3. $L \rightarrow * R$
- 4. $L \rightarrow id$
- 5. $R \rightarrow L$

Example 20.3 (continued)

Let us now make the SLR(1) parsing table

State	ACTION				GOTO		
	*	=	id	\$	S	L	R
0	s_4		s_5		1	2	3
1				Accept			
2		s_6 / r_5		r_5			

Production rules:

- $S' \rightarrow S$
- 1. $S \rightarrow L = R$
- 2. $S \rightarrow R$
- 3. $L \rightarrow * R$
- 4. $L \rightarrow id$
- 5. $R \rightarrow L$

s/r conflict occurs in because follow(R) is same as the shifted terminal '='

S	L	R
\$	=	\$
	\$	=

Compiler Design

Example 20.3 (continued)

State	ACTION				GOTO		
	*	=	id	\$	S	L	R
0	s_4			s_5	1	2	3
1					Accept		
2			s_6 / r_5		r_5		
3					r_2		
4	s_4			s_5		8	7
5			r_4		r_4		
6	s_4			s_5		8	9
7			r_3		r_3		
8			r_5		r_5		
9					r_1		

Production rules:

- $S' \rightarrow S$
- 1. $S \rightarrow L = R$
- 2. $S \rightarrow R$
- 3. $L \rightarrow * R$
- 4. $L \rightarrow id$
- 5. $R \rightarrow L$

S	L	R
\$	=	\$
	\$	=

Even though the SLR(1) parser removed one of the two s/r conflicts that were seen in LR(0) parser, the other s/r conflict could not be removed. SLR(1) is not very powerful.

This means that we need more powerful parsers: those which will look into particular productions/derivations and not generalise things by simply looking into the follow table.

In the next lecture, we will look into parsers such as CLR(1)/LR(1) and LALR(1).

Construct the LR(0) and SLR parsing tables for the following grammar:

$$S \rightarrow AaAb \mid BbBa$$

$$A \rightarrow \lambda$$

$$B \rightarrow \lambda$$

Augmenting and numbering the grammar

$$\begin{array}{ll} S' & \rightarrow S \\ 1. \ S & \rightarrow AaAb \\ 2. \ S & \rightarrow BbBa \\ 3. \ A & \rightarrow \lambda \\ 4. \ B & \rightarrow \lambda \end{array}$$

follow(S) = { \$ }
follow(A) = { a, b, \$ }
follow(B) = { \$, a, b }

S	A	B
\$	a	\$
	\$	a
	b	b

Making the LR(0) automaton

S'	\rightarrow	. S
S	\rightarrow	. $AaAb$
S	\rightarrow	. $BbBa$
A	\rightarrow	.
B	\rightarrow	.
		I0

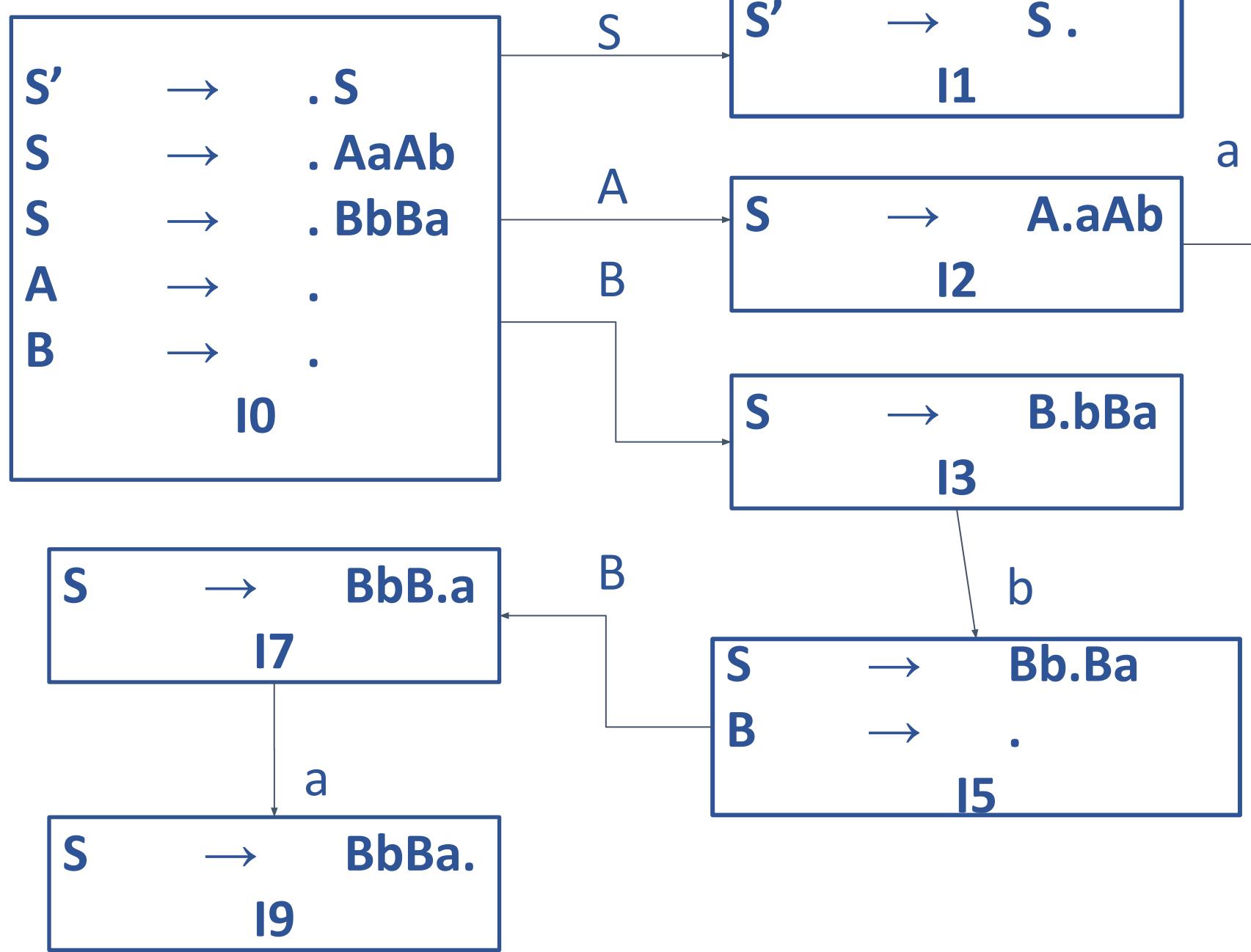
There are 2 final items in this itemset which leads to a reduce-reduce conflict.

Hence, the given grammar is not in LR(0).

Compiler Design

Example 20.4 (continued)

Making the LR(0) automaton



Production rules:

1. $S' \rightarrow S$
2. $S \rightarrow AaAb$
3. $S \rightarrow BbBa$
4. $A \rightarrow \lambda$
5. $B \rightarrow \lambda$

Example 20.4 (continued)

Let us now make the SLR(1) parsing table

State	ACTION			GOTO		
	a	b	\$	S	A	B
0	r_3/r_4	r_3/r_4	r_3/r_4	1	2	3
1			Accept			

Reduce-reduce conflict occurs as
follow(A)=follow(B)

Production rules:

- $S' \rightarrow S$
- 1. $S \rightarrow AaAb$
- 2. $S \rightarrow BbBa$
- 3. $A \rightarrow \lambda$
- 4. $B \rightarrow \lambda$

S	A	B
\$	a	\$
	\$	a
	b	b

Compiler Design

Example 20.4 (continued)

State	ACTION			GOTO		
	a	b	\$	S	A	B
0	r_3/r_4	r_3/r_4	r_3/r_4	1	2	3
1			Accept			
2	s_4					
3		s_5				
4	r_3	r_3	r_3		6	
5	r_4	r_4	r_4			7
6		s_8				
7	s_9	r_3	r_3			
8			r_1			
9			r_2			

Production rules:

$$S' \rightarrow S$$

$$1. S \rightarrow AaAb$$

$$2. S \rightarrow BbBa$$

$$3. A \rightarrow \lambda$$

$$4. B \rightarrow \lambda$$

S	A	B
\$	a	\$
	\$	a
	b	b

Construct the LR(0) and SLR parsing tables for the following grammar:

$$S \rightarrow Aa \mid bAc \mid dc \mid bd$$

$$A \rightarrow d$$

Augmenting and numbering the grammar

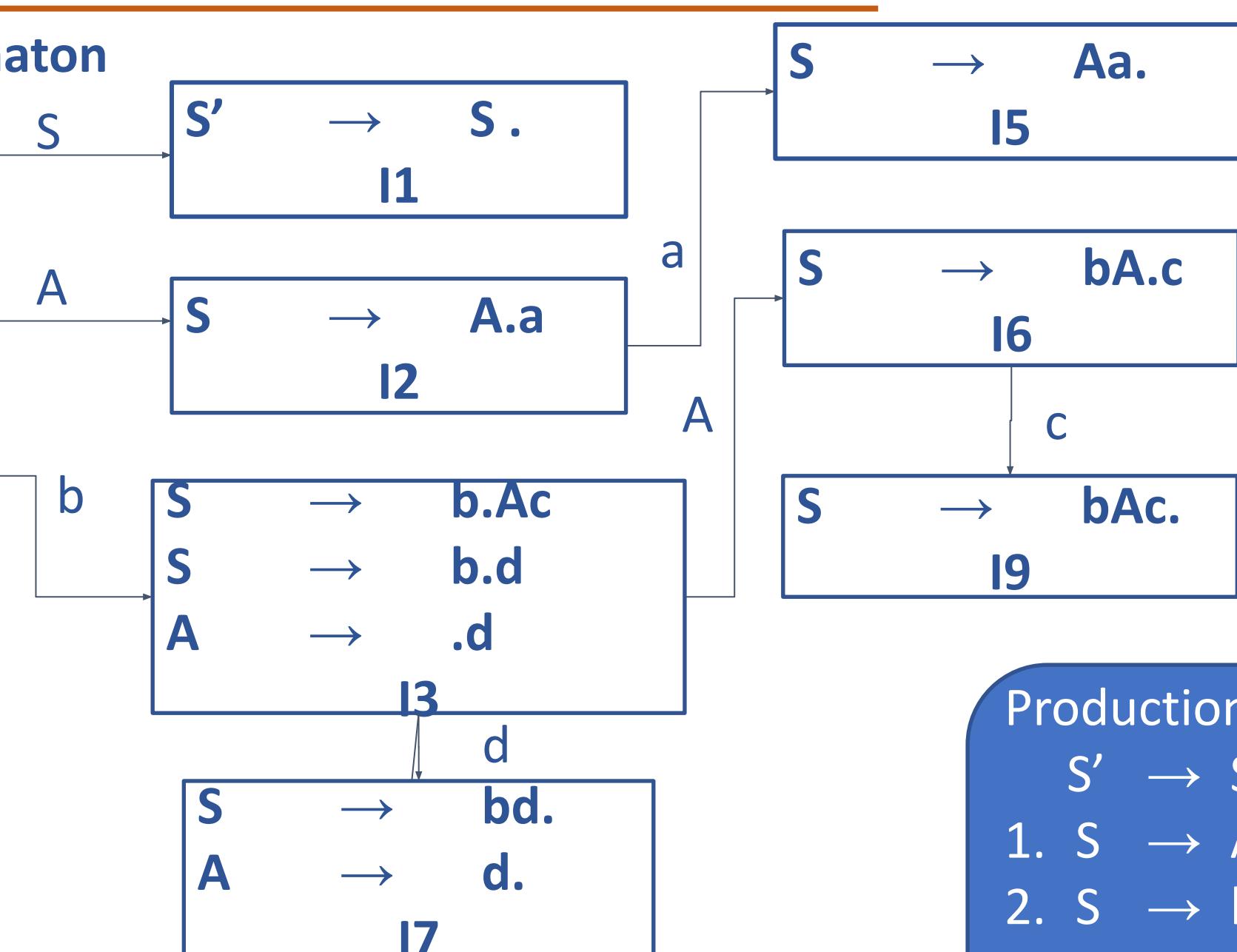
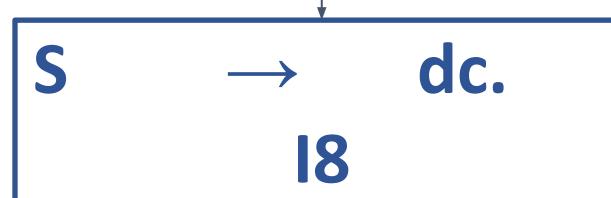
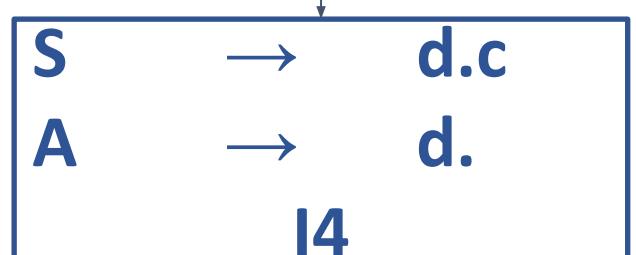
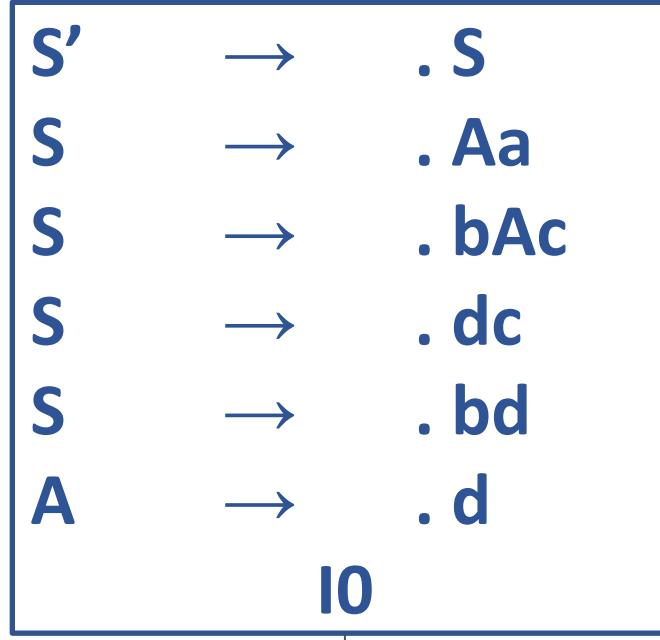
- | | | |
|--------|---------------|-------|
| S' | \rightarrow | S |
| 1. S | \rightarrow | Aa |
| 2. S | \rightarrow | bAc |
| 3. S | \rightarrow | dc |
| 4. S | \rightarrow | bd |
| 5. A | \rightarrow | d |

$\text{follow}(S) = \{\$\}$
 $\text{follow}(A) = \{a, c\}$

S	A
$\$$	a
	c

Example 20.5 (continued)

Making the LR(0) automaton



Production rules:

1. $S' \rightarrow S$
2. $S \rightarrow Aa$
3. $S \rightarrow bAc$
4. $S \rightarrow dc$
5. $S \rightarrow bd$
6. $A \rightarrow d$

Compiler Design

Example 20.5 (continued)

State	ACTION					GOTO	
	a	b	c	d	\$	S	A
0		s_3		s_4		1	2
1					Accept		
2	s_5						
3					s_6		5
4	r_5	r_5	s_8/r_5	r_5	r_5		
5	r_1	r_1	r_1	r_1	r_1		
6			s_9				
7	r_4/r_5	r_4/r_5	r_4/r_5	r_4/r_5	r_4/r_5		
8	r_3	r_3	r_3	r_3	r_3		
9	r_2	r_2	r_2	r_2	r_2		

Production rules:

- $S' \rightarrow S$
- 1. $S \rightarrow Aa$
- 2. $S \rightarrow bAc$
- 3. $S \rightarrow dc$
- 4. $S \rightarrow bd$
- 5. $A \rightarrow d$

S	A
\$	a
	c

Compiler Design

Example 20.5 (continued)

State	ACTION					GOTO	
	a	b	c	d	\$	S	A
0			s_3		s_4	1	2
1					Accept		
2	s_5						
3					s_6		5
4	r_5			s_8/r_5			
5							
6			s_9				
7	r_5		r_5		r_4		
8	r_3	r_3	r_3	r_3	r_3		
9	r_2	r_2	r_2	r_2	r_2		

Production rules:

- $S' \rightarrow S$
- 1. $S \rightarrow Aa$
- 2. $S \rightarrow bAc$
- 3. $S \rightarrow dc$
- 4. $S \rightarrow bd$
- 5. $A \rightarrow d$

S	A
\$	a
	c

Construct the LR(0) and SLR parsing tables for the following grammar:

$$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$$

$$A \rightarrow d$$

$$B \rightarrow d$$

Augmenting and numbering the grammar

S'	\rightarrow	S
1. S	\rightarrow	Aa
2. S	\rightarrow	bAc
3. S	\rightarrow	Bc
4. S	\rightarrow	bBa
5. A	\rightarrow	d
6. B	\rightarrow	d

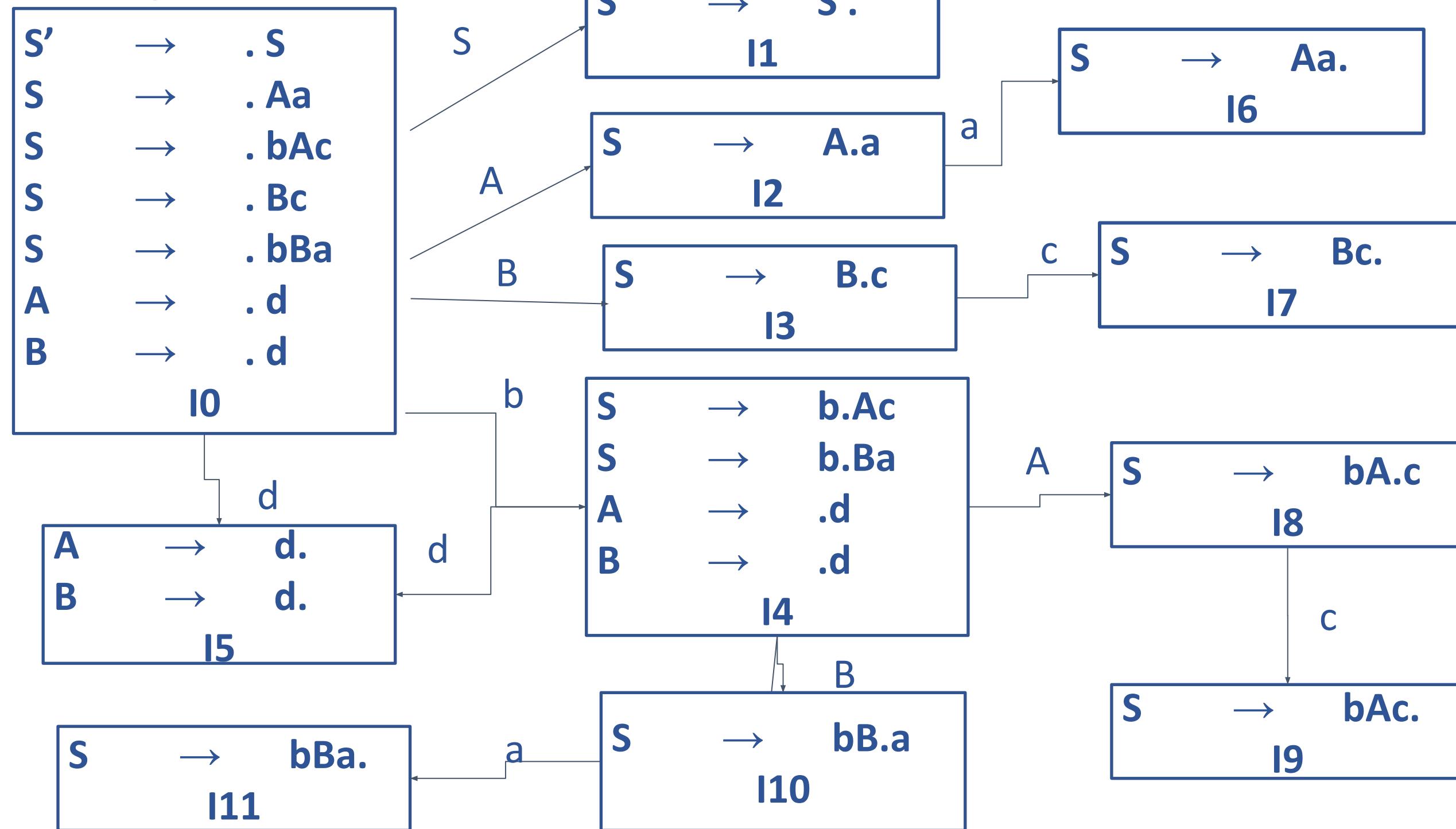
$\text{follow}(S) = \{\$\}$

$\text{follow}(A) = \{a, c\}$

$\text{follow}(B) = \{a, c\}$

S	A	B
$\$$	a	a
	c	c

Making the LR(0) automaton



Let us now make the LR(0) parsing table

State	ACTION					GOTO		
	a	b	c	d	\$	S	A	B
0		s_4			s_5	1	2	3
1					Accept			
2	s_6							
3			s_7					
4					s_5	8	10	
5	r_5/r_6	r_5/r_6	r_5/r_6	r_5/r_6	r_5/r_6			

Reduce-reduce conflict occurs at I4 as there are two final items.

Production rules:

- $S' \rightarrow S$
- 1. $S \rightarrow Aa$
- 2. $S \rightarrow bAc$
- 3. $S \rightarrow Bc$
- 4. $S \rightarrow bBa$
- 5. $A \rightarrow d$
- 6. $B \rightarrow d$

S	A	B
\$	a	a
	c	c

Let us now make the LR(0) parsing table

State	ACTION					GOTO		
	a	b	c	d	\$	S	A	B
0		s_4			s_5	1	2	3
1					Accept			
2	s_6							
3			s_7					
4				s_5		8	10	
5	r_5/r_6		r_5/r_6					

Reduce-reduce conflict occurs as
follow(A)=follow(B)

Production rules:

- $S' \rightarrow S$
- 1. $S \rightarrow Aa$
- 2. $S \rightarrow bAc$
- 3. $S \rightarrow Bc$
- 4. $S \rightarrow bBa$
- 5. $A \rightarrow d$
- 6. $B \rightarrow d$

S	A	B
\$	a	a
	c	c



THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu



Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Lavitra Kshitij Madan

Compiler Design

Unit 2

Table-driven BUP

Preet Kanwal
Department of Computer Science & Engineering

In this lecture, we will learn about:

- The need for LR(1)
- LR(1) automaton and LR(1) items
- CLR Parsers (with example problem)
- LALR Parsers
- Conflicts in LALR and CLR parsers

Analysis of SLR(1)

- Exploits lookahead in a small space.
- Small automaton – same number of states as in as LR(0).
- Works on many more grammars than LR(0)
- Too weak for most grammars: lose context from not having extra states.

In the previous lecture, we saw the need for parsers that are more powerful than SLR. The need is for those parsers that look into particular derivations while looking-ahead.

LALR (LookAhead LR) and CLR (Canonical LR) parsers use LR(1) automata because these parsers use LR(1) items.

These make use of canonical collection of LR(1) items.

While an ‘LR(0) automata’ uses ‘LR(0) item’ or simply ‘item’, that looks something like:

$$A \rightarrow X . Y Z$$

An ‘LR(1) item’ is made up of two parts:

LR(0) item + Look-ahead

For example:

$$A \rightarrow X . Y Z , \{a, b\}$$

Whenever we have a final item in a state,

e.g.: $S \rightarrow a A . , \{a, b\}$

the lookahead means, place the reduce move in the lookahead symbols – a, b

rather than placing them in

→ entire row (as in LR(0))

or

→ follow(LHS) (as in SLR)

When using LR(1), we start with:



When using LR(0), we start with:



A snapshot of the table looks like this for both the cases:

State	Action	Goto
	$\$$	S
1	Accept	

Bottom-up predictive parsing with

- L: Left-to-right scan
 - R: Rightmost derivation in Reverse
 - (1): One token lookahead
-
- Substantially more powerful than the other methods we've covered so far.
 - Tries to more intelligently find handles by using a lookahead token at each step

The Intuition behind LR(1)

- Guess which series of productions we are reversing.
- Use this information to maintain information about what lookahead to expect.
- When deciding whether to shift or reduce, use lookahead to disambiguate.

The Power of LR(1)

- Any LR(0) grammar is LR(1).
- Any LL(1) grammar is LR(1).
- Any deterministic CFL (a CFL parseable by a deterministic pushdown automaton) has an LR(1) grammar.

LR(1) Automata are Huge; Rarely used in Practice

- In a grammar with n terminals, could in theory be $O(2n)$ times as large as the LR(0) automaton.
- Replicate each state with all $O(2n)$ possible lookaheads.
- LR(1) tables for practical programming languages can have hundreds of thousands or even millions of states.
- Consequently, LR(1) parsers are rarely used in practice.

Compiler Design

Calculating Lookahead



Suppose we have an LR(1) item,

$$A \rightarrow \alpha . B \beta , a$$

- after dot, we have a NT (B),
hence we introduce the productions of B

$$B \rightarrow . \gamma , \boxed{}$$

- The lookahead for this item is calculated
from $A \rightarrow \alpha . B \beta , a$ [Parent item]
as because of this item, $B \rightarrow . \gamma$ [Child item]
was added.
The lookahead at this point = first (βa)

We start with the augmented production and with lookahead '\$',
i.e.,

$$S' \rightarrow . S , \$$$

Consider the following grammar:

$$\begin{array}{l} S \rightarrow L = R \mid R \\ L \rightarrow * R \mid id \\ R \rightarrow L \end{array}$$

Note: We have already seen that this grammar is not in SLR

Consider the following grammar:

$$\begin{array}{lcl} S & \rightarrow & L = R \mid R \\ L & \rightarrow & * R \mid id \\ R & \rightarrow & L \end{array}$$

Augmenting and numbering the grammar:

1. $S' \rightarrow S$
2. $S \rightarrow L = R$
3. $S \rightarrow R$
4. $L \rightarrow * R$
5. $L \rightarrow id$
6. $R \rightarrow L$

Example 21.1 solution (continued)

Step 1: Starting

$$S' \rightarrow . S, \$$$

Step 2-4: Repeatedly adding productions of NTs after “.”s

Recall that

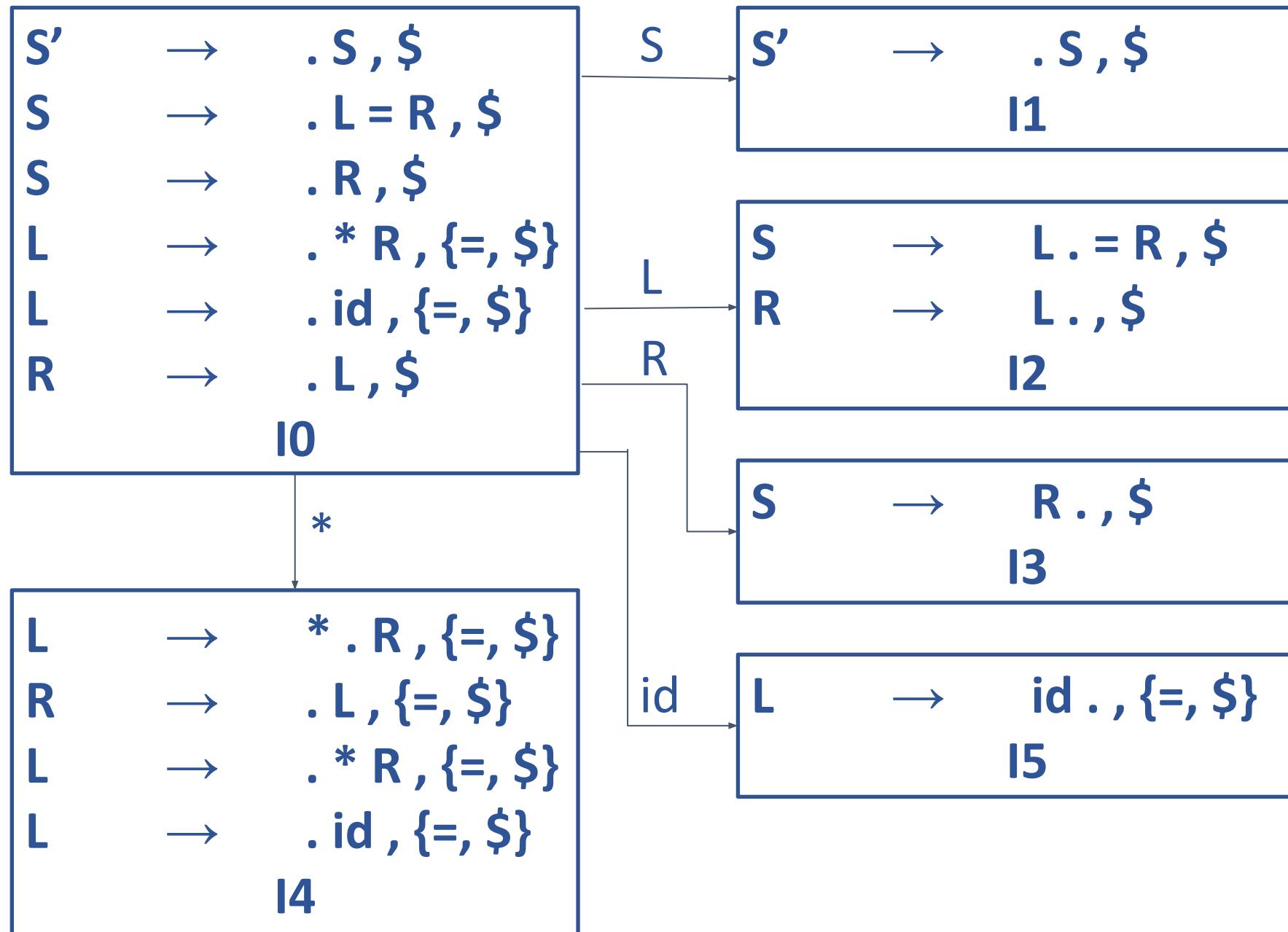
Look-ahead(child-item) = follow(NT) in the parent item

S'	\rightarrow	$. S, \$$
S	\rightarrow	$. L = R, \$$
S	\rightarrow	$. R, \$$
L	\rightarrow	$. * R, \{=, \$\}$
L	\rightarrow	$. id, \{=, \$\}$
R	\rightarrow	$. L, \$$
10		

Production rules:

$S' \rightarrow S$	$S \rightarrow L = R$
1. $S \rightarrow R$	
2. $L \rightarrow * R$	
3. $L \rightarrow id$	
4. $R \rightarrow L$	

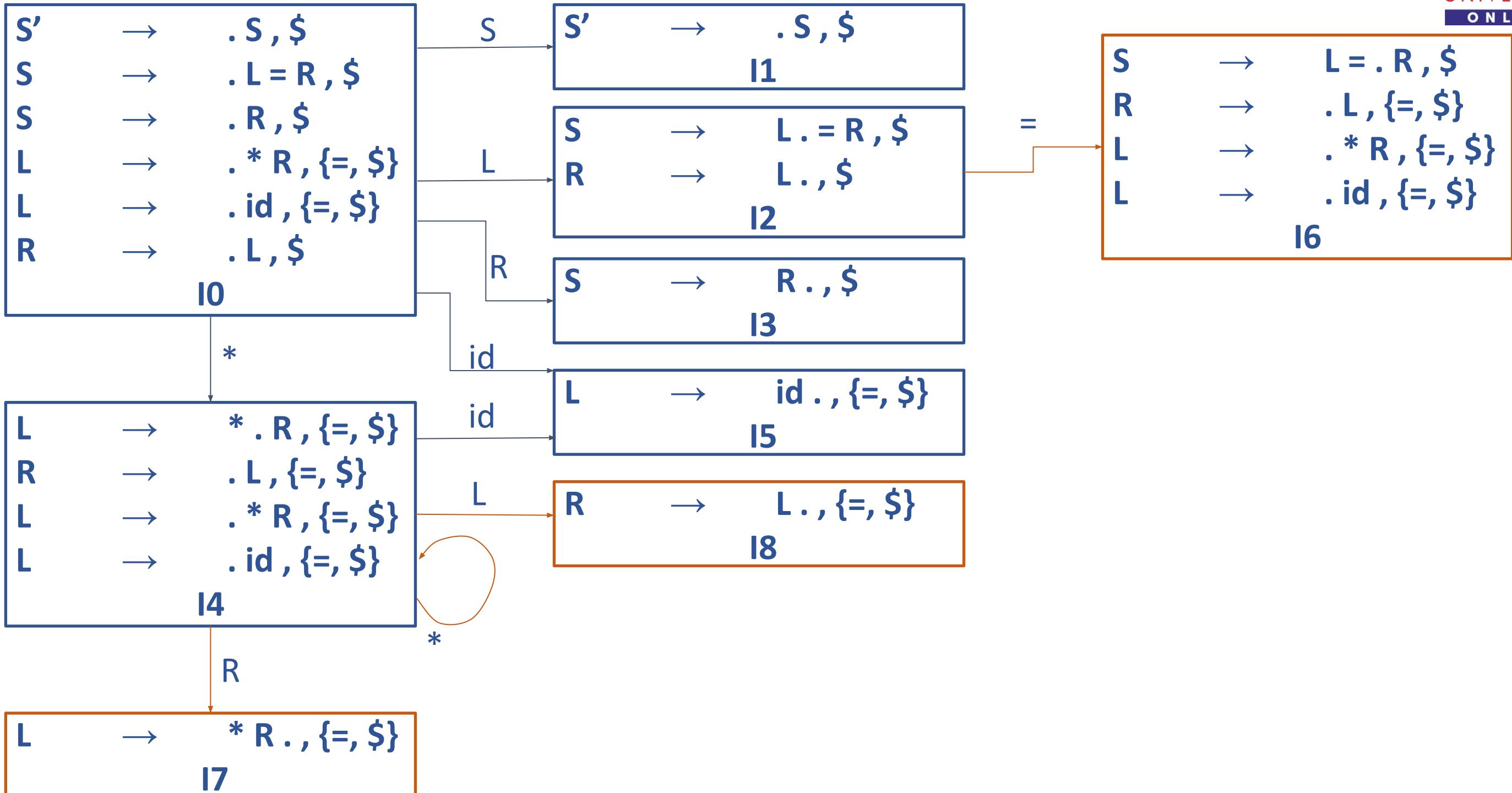
Step 5: Transitions using the goto function



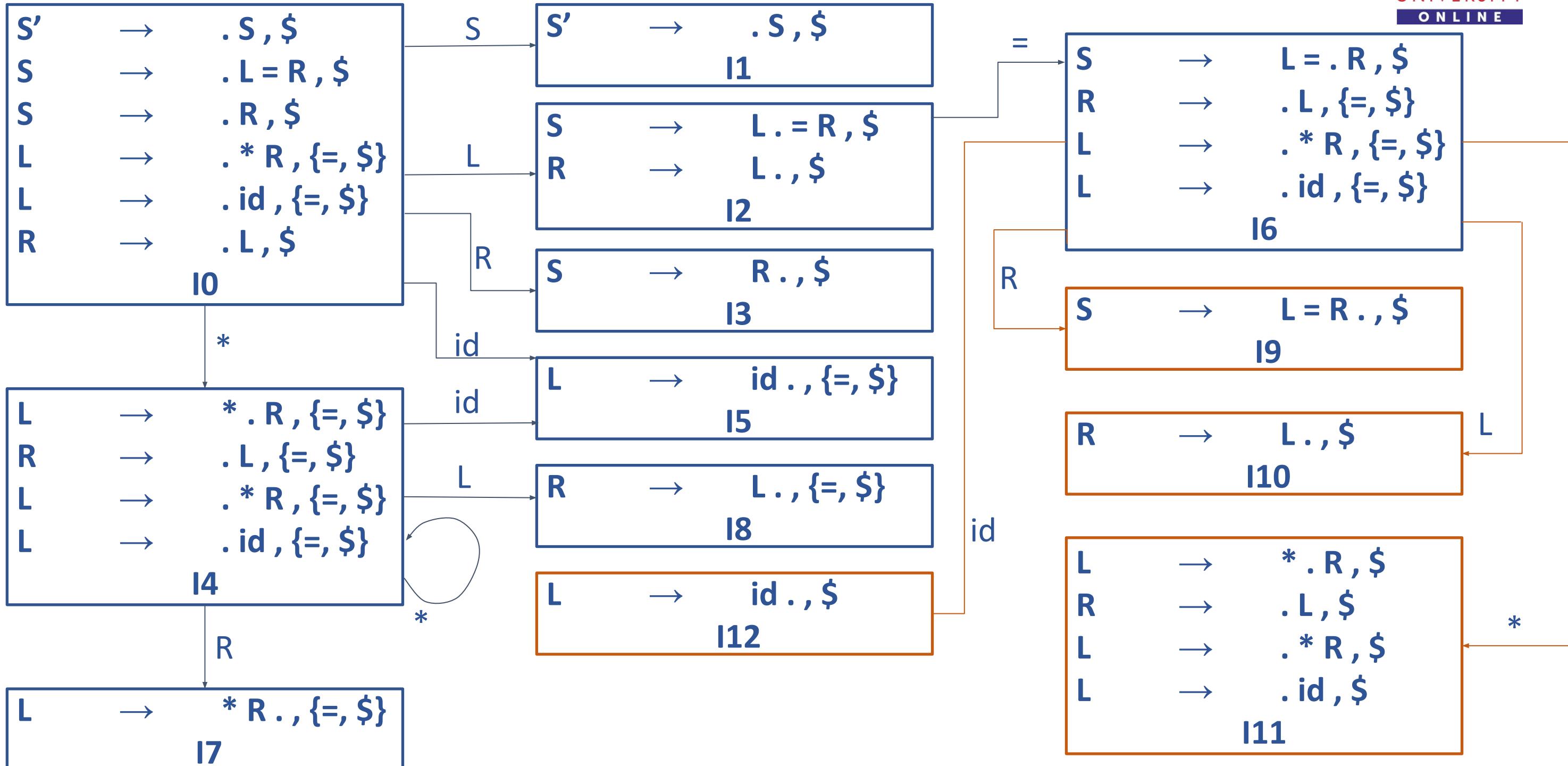
Production rules:

1. $S' \rightarrow S$
2. $S \rightarrow L = R$
3. $S \rightarrow R$
4. $L \rightarrow * R$
5. $L \rightarrow id$
6. $R \rightarrow L$

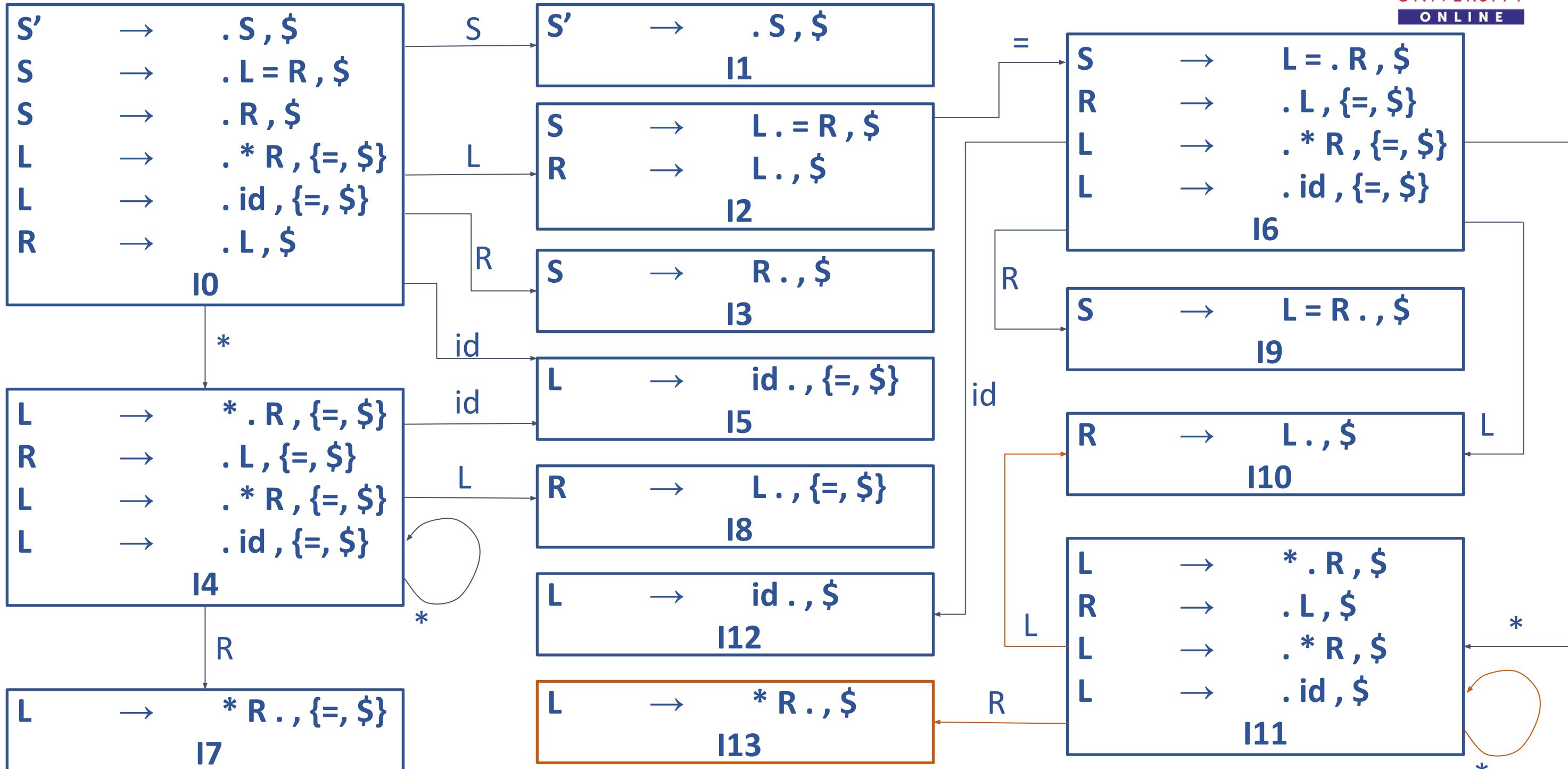
Example 21.1 solution (continued) : step 6



Example 21.1 solution (continued) : step 6



Example 21.1 solution (continued) : step 6



Example 21.1 solution (continued) - CLR parsing table

State	Action				Goto		
	*	id	=	\$	S	L	R
0							

Let us fill the CLR parsing table now.

State	Action				Goto		
	*	id	=	\$	S	L	R
0	s_4	s_5			1	2	3

Example 21.1 solution (continued) - CLR parsing table

State	Action				Goto		
	*	id	=	\$	S	L	R
0	s_4	s_5			1	2	3
1				Accept			
2			s_6	r_5			
3				r_2			
4	s_4	s_5				8	7
5			r_4	r_4			

Example 21.1 solution (continued) - CLR parsing table

State	Action				Goto		
	*	id	=	\$	S	L	R
0	s_4	s_5			1	2	3
1				Accept			
2			s_6	r_5			
3				r_2			
4	s_4	s_5				8	7
5			r_4	r_4			
6	s_{11}	s_{12}				10	9
7			r_3	r_3			
:	:	:	:	:	:	:	:

Table continued on
next slide.

Example 21.1 solution (continued) - CLR parsing table

State	Action							Goto		
	*	id	=	\$	S	L	R			
:	:	:	:	:	:	:	:			
8			r_5	r_5						

Example 21.1 solution (continued) - CLR parsing table

State	Action				Goto		
	*	id	=	\$	S	L	R
:	:	:	:	:	:	:	:
8			r_5	r_5			
9				r_1			
10				r_5			
11	s_{11}	s_{12}				10	13
12				r_4			

Example 21.1 solution (continued) - CLR parsing table

State	Action							Goto		
	*	id	=	\$	S	L	R			
:	:	:	:	:	:	:	:			
8				r_5	r_5					
9					r_1					
10					r_5					
11	s_{11}	s_{12}						10	13	
12					r_4					
13					r_3					

No conflicts occurred!

Example 21.1 solution (continued) - Merging states

Notice that there are some states with the same LR(0) itemsets but different lookaheads. All such pairs are mentioned below:

- I4 – I11
- I5 – I12
- I7 – I13
- I8 – I10

So, we merge them. For example, merging I4 and I11:

$$L \rightarrow * . R, \{=, \$\}$$

$$R \rightarrow . L, \{=, \$\}$$

$$L \rightarrow . * R, \{=, \$\}$$

$$L \rightarrow . id, \{=, \$\}$$

Example 21.1 solution (continued) - Merging states

LALR table will be smaller than CLR table in this case.

Note : While merging, check for any problems.

Example-

$$M[4, *] = s_4$$

and

$$M[11, *] = s_{11}$$

This will not be a problem since s_{4-11} will be a new state.

Also take care of reduce-reduce conflicts.

Example 21.1 solution (continued) - LALR parsing table

State	Action				Goto		
	*	id	=	\$	S	L	R
0	s_{4-11}	s_{5-12}			1	2	3
1				Accept			
2			s_6	r_{5-12}			
3				r_2			
4 - 11	s_{4-11}	s_{5-12}				8 - 10	7 - 13
5 - 12			r_4	r_4			
6	s_{4-11}	s_{5-12}				8 - 10	9
7 - 13			r_3	r_3			
8 - 10			r_{5-12}	r_{5-12}			
9				r_1			

Alternatively, we can take the lower number for each merged state :

4 - 11	as	4
5 - 12	as	5
7 - 13	as	7
8 - 10	as	8

Example 21.1 (continued) - Parsing an input

Now, considering the LALR table, let us parse the following string:

w = “* id = id”

Note: While parsing there must always be a state number at the top of the stack.

Starting with the initial configuration:

Stack	I/p buffer	Action
\$ 0	* id = id \$	

Referring the table, a “*” on state 0 is “ s_4 ”. It means that “*” and “4” must be pushed onto the stack because we read the symbol “*” and move to state 4.

Example 21.1 (continued) - Parsing an input

Stack	I/p buffer	Action
\$ 0	* id = id \$	A [0, *] = s_4 Push *, Push 4
\$ 0 * 4	id = id \$	

Similarly, on reading “id”, the table states that the action that must be performed is “ s_5 ” or pushing “id” and “5” onto the stack because we move to state 5.

Example 21.1 (continued) - Parsing an input

Stack	I/p buffer	Action
\$ 0	* id = id \$	A [0, *] = s_4 Push *, Push 4
\$ 0 * 4	id = id \$	A [4, id] = s_5 Push id , Push 5
\$ 0 * 4 id 5	= id \$	

Now, TOS is “5” and the next symbol in the input buffer is “=”. Using the table, we know that the action to take is “ r_4 ”, i.e., we must reduce using the fourth production rule:

$$L \rightarrow id$$

We must pop both- symbol and state (“5” and “id”)- and push the left-hand-side of the rule (“L”) and call the Goto function on the state that was on top before pushing “L” and push its output on the stack.

Example 21.1 (continued) - Parsing an input

Stack	I/p buffer	Action
\$ 0	* id = id \$	A [0, *] = s_4 Push *, Push 4
\$ 0 * 4	id = id \$	A [4, id] = s_5 Push id , Push 5
\$ 0 * 4 id 5	= id \$	A [5, =] = r_4 ($L \rightarrow id$) Pop 5 , Pop id , Push L Push Goto [4, L] = 8
\$ 0 * 4 L 8	= id \$	

Do note that “id” was not consumed yet.

Repeat this process until all of the input is read and we receive “Accept” in the action column.

Example 21.1 (continued) - Parsing an input

Stack	I/p buffer	Action
\$ 0	* id = id \$	A [0, *] = s_4 Push *, Push 4
\$ 0 * 4	id = id \$	A [4, id] = s_5 Push id , Push 5
\$ 0 * 4 id 5	= id \$	A [5, =] = r_4 ($L \rightarrow id$) Pop 5 , Pop id , Push L Push Goto [4, L] = 8
\$ 0 * 4 L 8	= id \$	A [8, =] = r_5 ($R \rightarrow L$) Pop 8 , Pop L , Push R Push Goto [4, R] = 7
\$ 0 * 4 R 7	= id \$	A [7, =] = r_3 ($L \rightarrow * R$) Pop 7, Pop R, Pop 4, Pop *, Push L Push Goto [0, L] = 2
\$ 0 L 2	= id \$	

If the production rule during the reduce move has x symbols on RHS, pop $2*x$ symbols from stack.

Table continued on next slide.

Example 21.1 (continued) - Parsing an input

Stack	I/p buffer	Action
\$ 0 L 2	= id \$	A [2, =] = s_6 Push = , Push 6
\$ 0 L 2 = 6	id \$	A [6, id] = s_5 Push id , Push 5
\$ 0 L 2 = 6 id 5	\$	A [5, \$] = r_4 ($L \rightarrow id$) Pop 5 , Pop id , Push L Push Goto [6, L] = 8
\$ 0 L 2 = 6 L 8	\$	A [8, \$] = r_5 ($R \rightarrow L$) Pop 8 , Pop L , Push R Push Goto [6, R] = 9
\$ 0 L 2 = 6 R 9	\$	A [9, \$] = r_1 ($S \rightarrow L = R$) Pop L 2 = 6 R 9 , Push S Push Goto [0, S] = 1
\$ 0 S 1	\$	A [1, \$] = Accept

LR(1) and SLR(1)

- **SLR(1) is weak because it has no contextual information.**
- **LR(1) is impractical because its contextual information makes the automaton too big.**
- **Can we retain the LR(1) automaton's contextual information without all its states?**

This is where LALR comes into action.

Compiler Design

LALR Parsers



- There may be some grammar, where there exist at least one set of two or more states that have the same LR(0) itemsets but different look-aheads.
- These itemsets are merged together to reduce the number of states.
- The resulting parsing table is the LALR parsing table.
- While merging states, there may arise some problems in the form of conflicts and thus LALR parser is weaker than CLR parser. But, it is more powerful than LR(0) parsers.
- Yacc is an LALR parser!

- Maintains context.
- Lookup sets based on the fine-grained LR(1) automaton.
- Each state's lookup relevant only for that state.
- Keeps the automaton small.
- Resulting automaton has same size as LR(0) automaton.

LALR(1) is Powerful!

- Every LR(0) grammar is LALR(1).
- Every SLR(1) grammar is LALR(1)
- Most(but not all) LR(1) grammars are LALR(1).

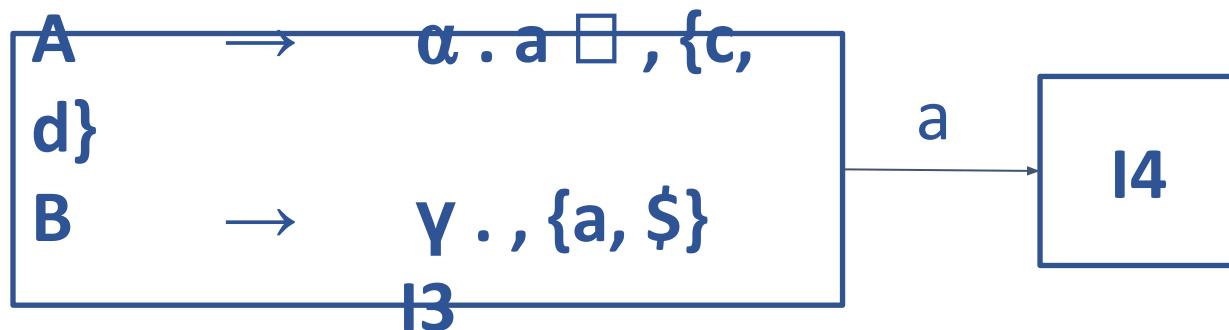
There exist a reduce-reduce (r/r) conflict in CLR if and only if there are two or more final items in a state with the same lookahead symbols.

Example:

A	→	a . , a
B	→	□ . , a
I3		

A shift/reduce (s/r) conflict occurs in CLR when there is a final item and a shift on a terminal where that terminal also exists in the lookahead set of the final item.

Example:



The reduce move will be placed under 'a' which also has a shift move

Merging LR(1) states can introduce a reduce/reduce conflict.

There exist a r/r conflict in LALR if and only if, upon merging, there are two or more final items in a state that have the same lookahead.

Example:

A	→	$\alpha ., a$
B	→	$\square ., b$
I3		

A	→	$\alpha ., b$
B	→	$\square ., a$
I4		

A	→	$\alpha ., \{a, b\}$
B	→	$\square ., \{a, b\}$
I7		

Often these conflicts appear without any good reason; this is one limitation of LALR(1).

Merging LR(1) states cannot introduce a shift/reduce conflict.

Why?

Since the items have the same core, a shift/reduce conflict in a LALR(1) state would have to also exist in one of the LR(1) states it was merged from.

So, the same condition as that of CLR parser holds true if not already checked previously.

Compiler Design

Example problem



Homework problem

Consider the following grammar:

$$S \rightarrow AaAb \mid BbBa$$

$$A \rightarrow \lambda$$

$$B \rightarrow \lambda$$

Note: This grammar is not in SLR



THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu



Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Lavitra Kshitij Madan

Compiler Design

Unit 2

Table-driven BUP

Preet Kanwal
Department of Computer Science & Engineering

In this lecture, we will practice questions around:

- CLR Parsers
- LALR Parsers

We will learn about:

- Comparison between all table-driven bottom-up parsers

Consider the following grammar:

$$S \rightarrow AaAb \mid BbBa$$

$$A \rightarrow \lambda$$

$$B \rightarrow \lambda$$

Note: This grammar is not in SLR

Consider the following grammar:

$$S \rightarrow A a A b \mid B b B a$$

$$A \rightarrow \lambda$$

$$B \rightarrow \lambda$$

Augmenting and numbering the grammar:

$$S' \rightarrow S$$

1. $S \rightarrow A a A b$

2. $S \rightarrow B b B a$

3. $A \rightarrow \lambda$

4. $B \rightarrow \lambda$

Step 1: Starting

Starting with the initial item.

$$S' \rightarrow . S , \$$$

Step 2: Adding productions of NTs after the dot

Recall how to calculate the lookahead for the child items

The parent will specify the look-ahead.

Look-ahead(child-item) = follow(NT) in the parent item

$$S' \rightarrow . S [, \$] \Rightarrow \text{The look-ahead here is '}'}$$

Production rules:

- 1. $S' \rightarrow S$
- 2. $S \rightarrow A a A b$
- 3. $S \rightarrow B b B a$
- 4. $A \rightarrow \lambda$
- 5. $B \rightarrow \lambda$

$S' \rightarrow . S , \$$	}	Parent item
$S \rightarrow . A a A b , \$$	}	Child items
$S \rightarrow . B b B a , \$$		

Step 3: Repeating step 2 until no more items

For finding the lookaheads of the child items of items 2 and 3,
consider the following:

$$S \rightarrow . A [a A b , \$]$$

Since “a” follows “A”, it is the look-ahead.

The items in the state now are:

$$S' \rightarrow . S , \$$$

$$S \rightarrow . A a A b , \$$$

$$S \rightarrow . B b B a , \$$$

$$A \rightarrow . , a$$

$$B \rightarrow . , b$$

Production rules:

$$S' \rightarrow S$$

$$1. S \rightarrow A a A b$$

$$2. S \rightarrow B b B a$$

$$3. A \rightarrow \lambda$$

$$4. B \rightarrow \lambda$$

Step 4: Making closure

S'	\rightarrow	. $S, \$$
S	\rightarrow	. $A a A b, \$$
S	\rightarrow	. $B b B a, \$$
A	\rightarrow	. , a
B	\rightarrow	. , b
		I0

Production rules:

1. $S' \rightarrow S$
2. $S \rightarrow A a A b$
3. $S \rightarrow B b B a$
4. $A \rightarrow \lambda$
5. $B \rightarrow \lambda$

Step 5: Transitions using the goto function

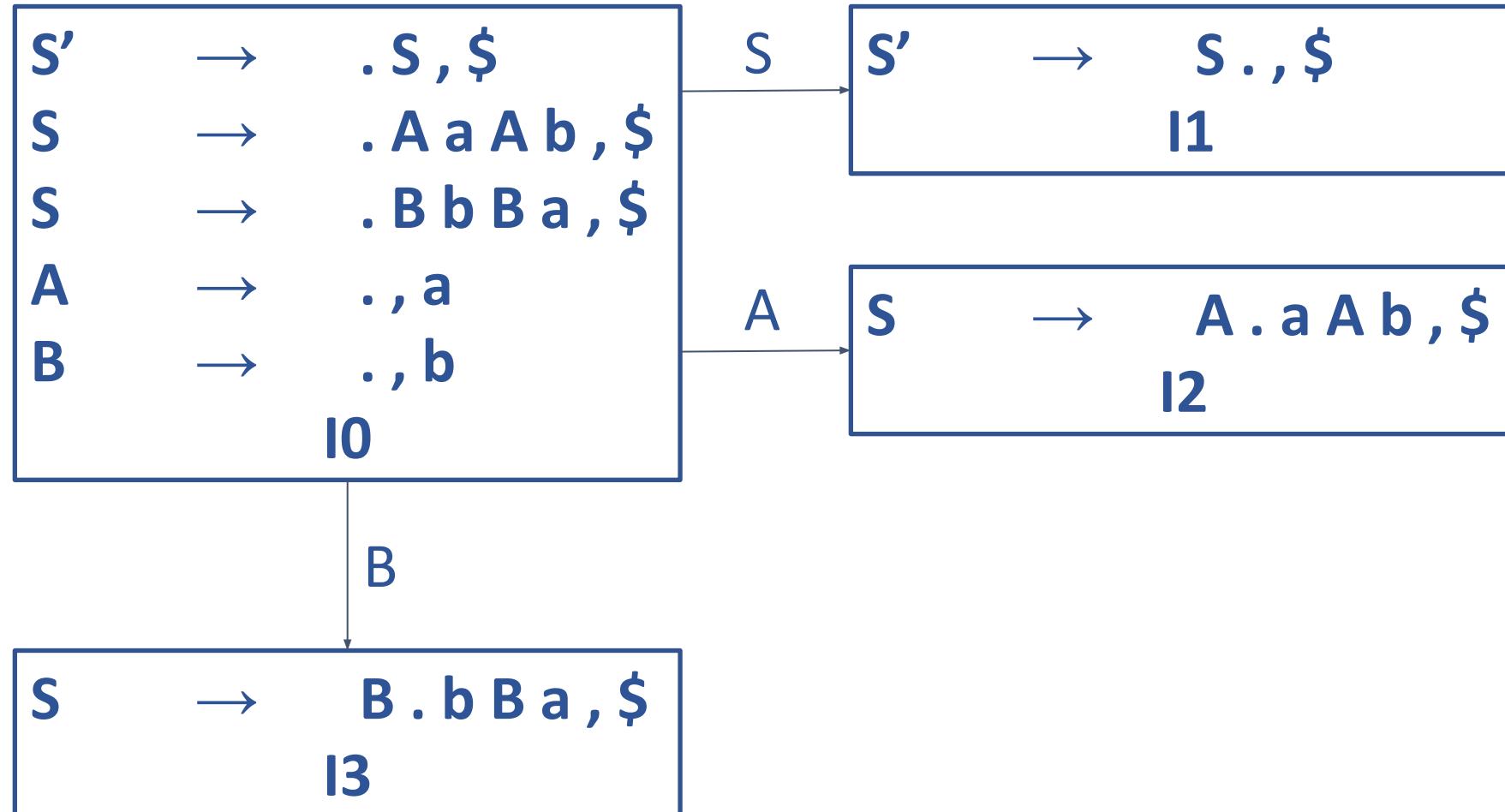
This state has transitions on 'S', 'A' and 'B' to the following

states:

- State I1 : $S' \rightarrow S . , \$$
- State I2 : $S \rightarrow A . a A b, \$$
- State I3 : $S \rightarrow B . b B a, \$$

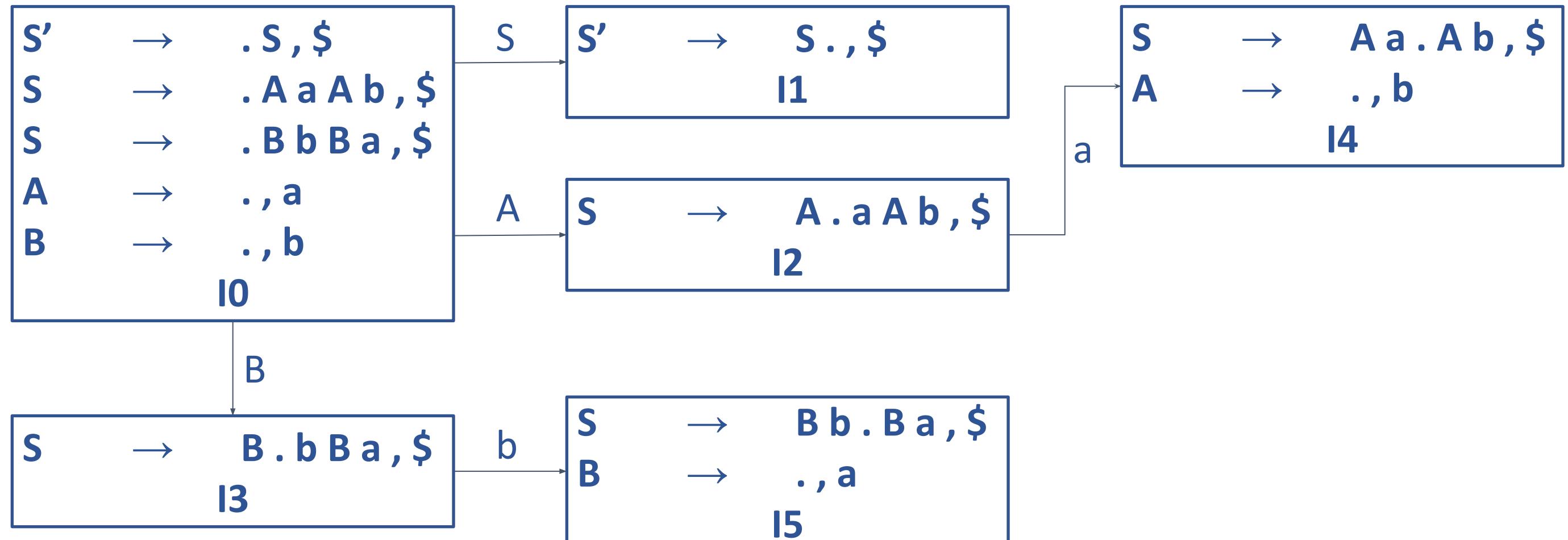
Example 22.1 solution (continued)

Step 5 continued

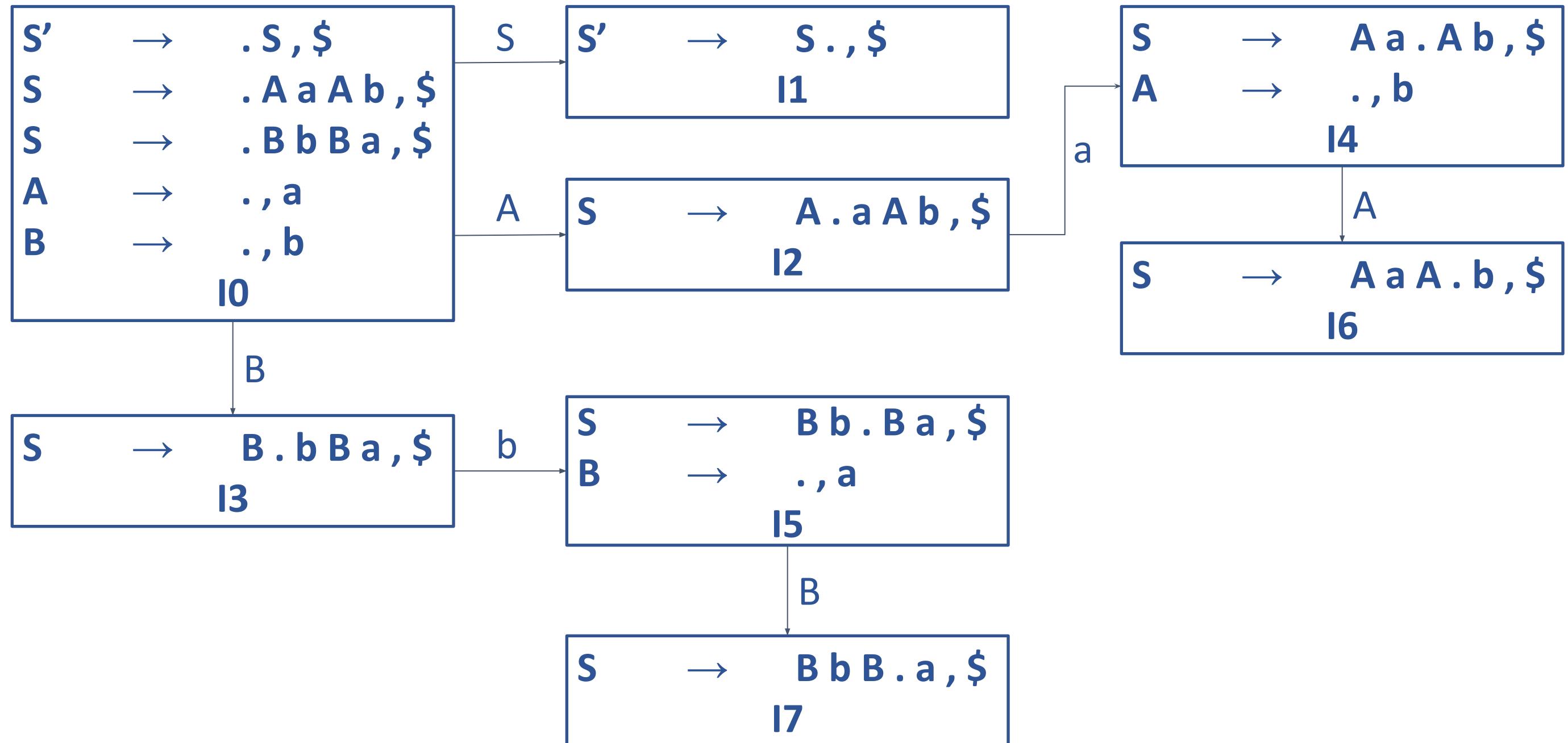


Example 22.1 solution (continued)

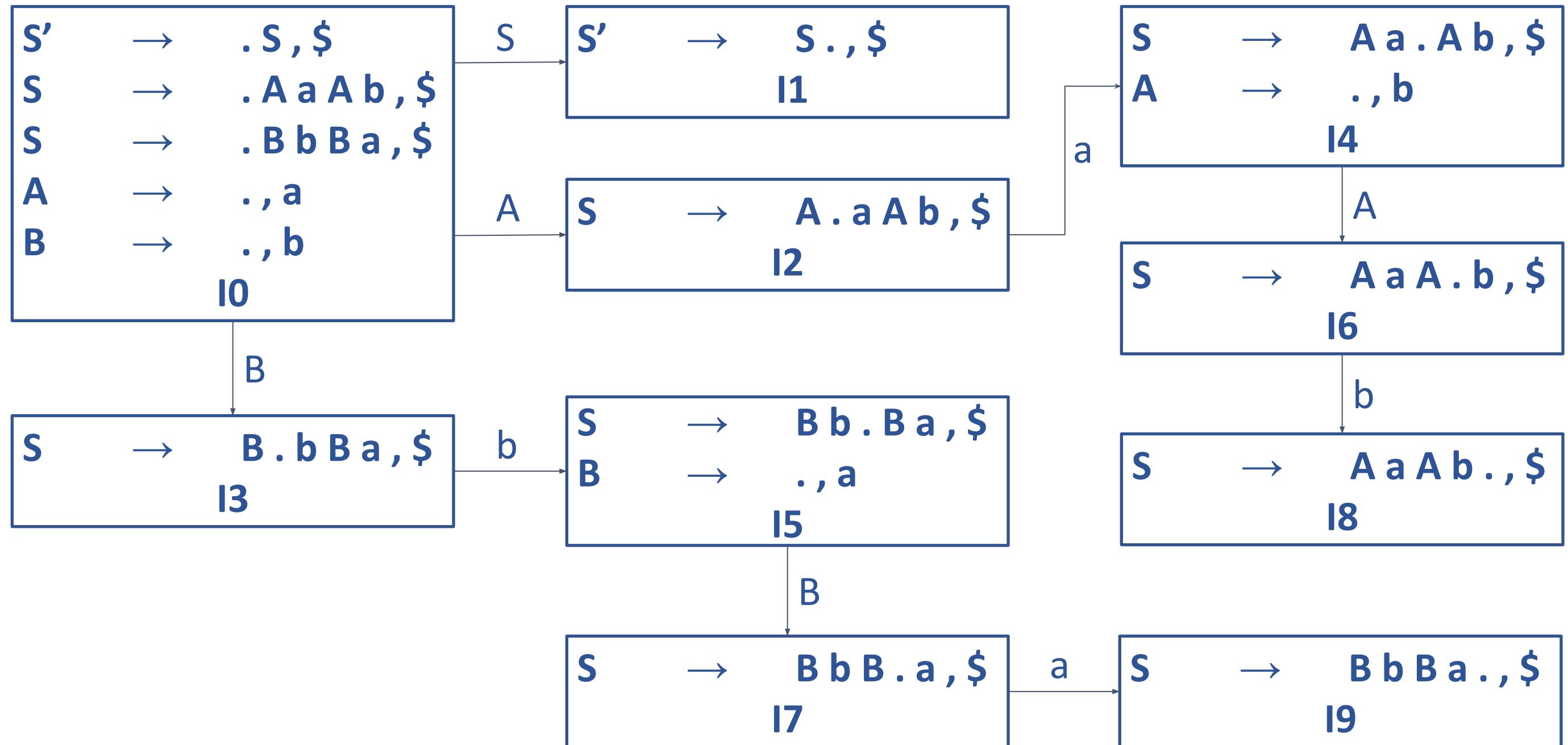
Step 6: Repeat steps 2-5 until no more states can be added



Step 6 continued



Step 6 continued



Example 22.1 solution (continued)

State	Action			Goto		
	a	b	\$	S	A	B
0						

Let us fill the CLR parsing table now.

State	Action			Goto		
	a	b	\$	S	A	B
0	r_3	r_4		1	2	3

Notice how the reduce move is placed only in the necessary columns. This is the power of LR(1).

Filling the table for the rest of the states.

State	Action			Goto		
	a	b	\$	S	A	B
0	r_3	r_4		1	2	3
1			Accept			
2	s_4					
3		s_5				

Compiler Design

Example 22.1 solution (continued)

State	Action			Goto		
	a	b	\$	S	A	B
0	r_3	r_4		1	2	3
1			Accept			
2	s_4					
3		s_5				
4		r_3			6	
5	r_4					7

Compiler Design

Example 22.1 solution (continued)

State	Action			Goto		
	a	b	\$	S	A	B
0	r_3	r_4		1	2	3
1			Accept			
2	s_4					
3		s_5				
4		r_3			6	
5	r_4					7
6		s_8				
7	s_9					

Example 22.1 solution (continued)

State	Action			Goto		
	a	b	\$	S	A	B
0	r_3	r_4		1	2	3
1			Accept			
2	s_4					
3		s_5				
4		r_3			6	
5	r_4					7
6		s_8				
7	s_9					
8			r_1			
9			r_2			

Note that no two states have the same LR(0) items with different lookaheads and hence no states can be merged. Therefore, LALR table is the same as CLR table in this case.

Is the following grammar in CLR and LALR?

$$\begin{array}{ll} S & \rightarrow Aa \mid bAc \mid Bc \mid bBa \\ A & \rightarrow d \\ B & \rightarrow d \end{array}$$

Is the following grammar in CLR and LALR?

$$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$$

$$A \rightarrow d$$

$$B \rightarrow d$$

Augmenting and numbering the grammar:

$$S' \rightarrow S$$

$$1. \quad S \rightarrow Aa$$

$$2. \quad S \rightarrow bAc$$

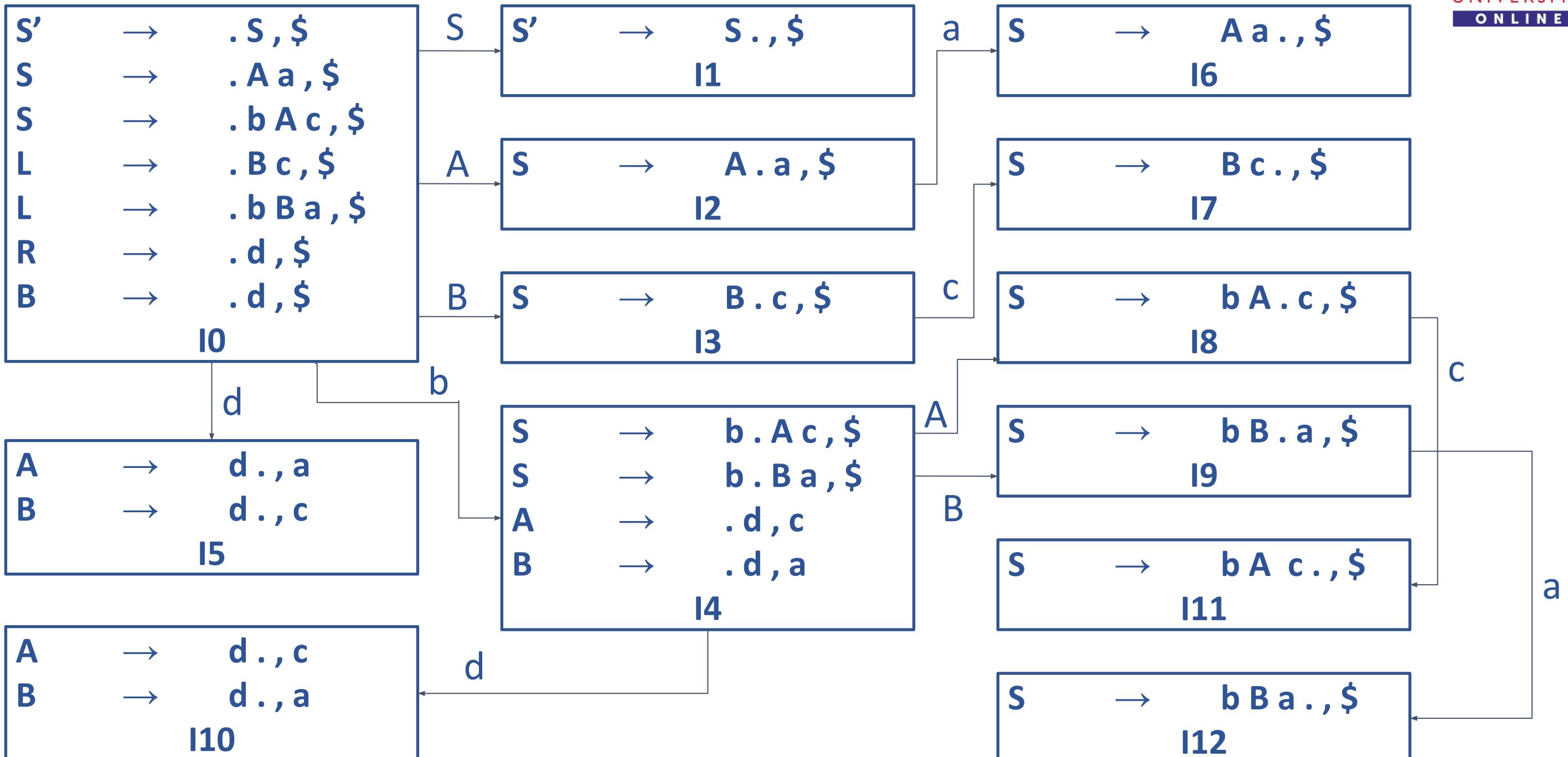
$$3. \quad L \rightarrow Bc$$

$$4. \quad L \rightarrow bBa$$

$$5. \quad A \rightarrow d$$

$$6. \quad B \rightarrow d$$

Example 22.2 solution (continued)



Example 22.2 solution (continued) - CLR parsing table

State	Action					Goto		
	a	b	c	d	\$	s	A	B
0		s_4		s_5		1	2	3
1					Accept			
2	s_6							
3			s_7					
4					s_{10}		8	9
5	r_5		r_6					
6					r_1			
:	:	:	:	:	:	:	:	:

Table continued on
next slide.

Example 22.2 solution (continued) - CLR parsing table

State	Action					Goto		
	a	b	c	d	\$	S	A	B
:	:	:	:	:	:	:	:	:
7					r_3			
8			s_{11}					
9	s_{12}							
10	r_6		r_5					
11					r_2			
12					r_4			

Grammar part of LR(1)

Example 22.2 solution (continued) - LALR parsing table

States 5 and 10 have the same LR(0) itemset but different lookaheads. But there is a catch.

Note the cells with text in orange colour.

When these states are merged, there will be a reduce-reduce conflict.

A snapshot of the LALR parsing table is given below:

State	Action				
	a	b	c	d	\$
5 - 10	r ₅ / r ₆		r ₆ / r ₅		

Consider the following grammar:

$$\begin{array}{l} S \rightarrow Aa \mid bAc \mid dc \mid bd \\ A \rightarrow d \end{array}$$

Recall that this grammar is not in LR(0) and SLR

Check if it is in CLR and/or LALR

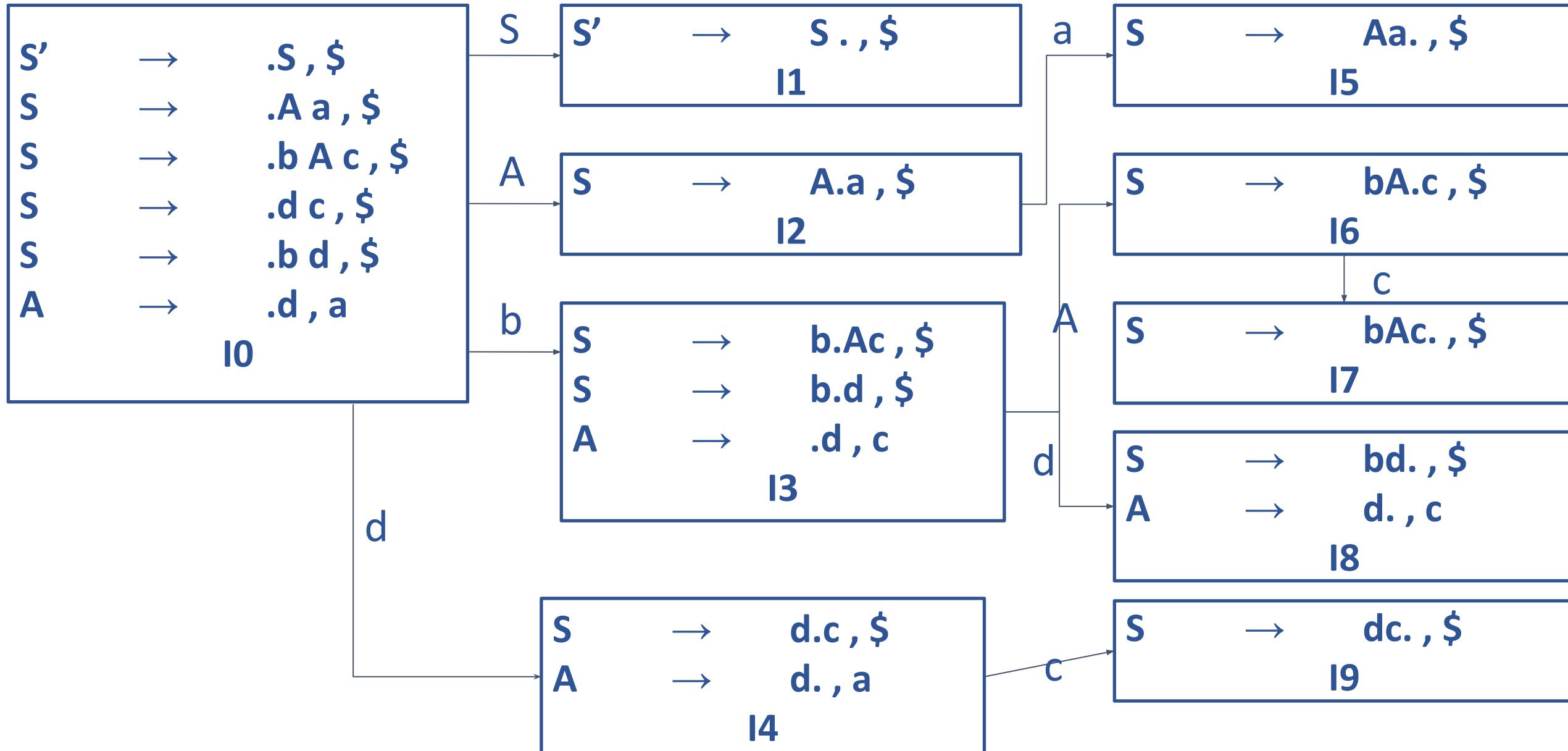
Consider the following grammar:

$$\begin{array}{l} S \rightarrow Aa \mid bAc \mid dc \mid bd \\ A \rightarrow d \end{array}$$

Augmenting and numbering the grammar:

- $S' \rightarrow S$
- 1. $S \rightarrow Aa$
- 2. $S \rightarrow bAc$
- 3. $S \rightarrow dc$
- 4. $S \rightarrow bd$
- 5. $A \rightarrow d$

Example 22.3 solution (continued)



Example 22.2 solution (continued) - CLR parsing table

State	Action					Goto	
	a	b	c	d	\$	S	A
0		s_3		s_4		1	2
1					Accept		
2	s_5						
3				s_8			6
4	r_5		s_9				
5					r_1		
6			s_7				
7					r_2		
8			r_5		r_4		
9					r_3		

The given grammar is in
CLR and LALR

Consider the following grammar:

$$\begin{array}{lcl} S & \rightarrow & AA \\ A & \rightarrow & aA \mid b \end{array}$$

Recall that this grammar is in LR(0)

Make its CLR, LALR parsing tables and parse the string “abb”

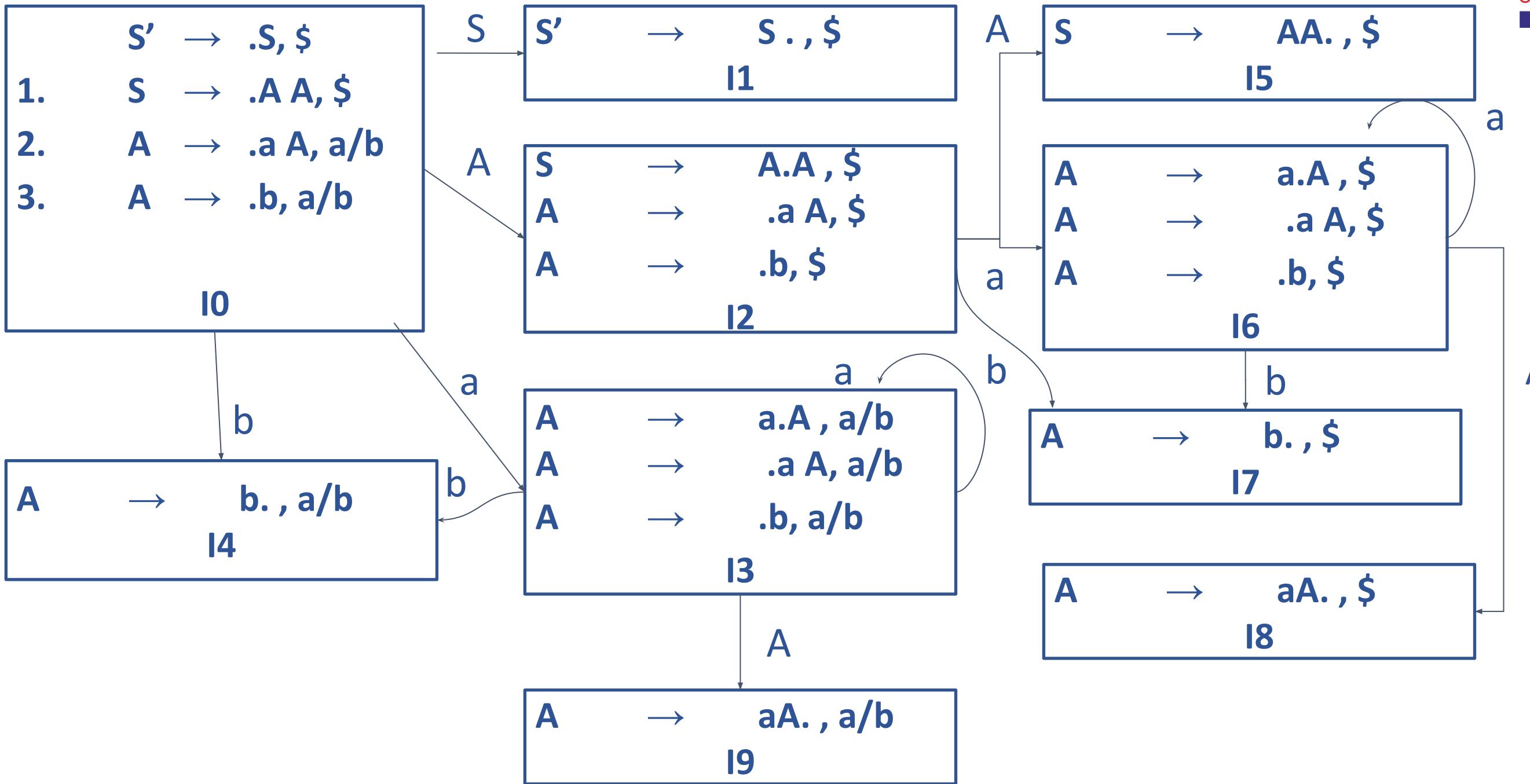
Consider the following grammar:

$$\begin{array}{lcl} S & \rightarrow & AA \\ A & \rightarrow & aA \mid b \end{array}$$

Augmenting and numbering the grammar:

- $S' \rightarrow S$
- 1. $S \rightarrow AA$
- 2. $A \rightarrow aA$
- 3. $A \rightarrow b$

Example 22.4 solution (continued)



Example 22.2 solution (continued) - CLR parsing table

State	Action			Goto	
	a	b	\$	S	A
0	s_3	s_4		1	2
1			Accept		
2	s_6	s_7			5
3	s_3	s_4			9
4	r_3	r_3			
5			r_1		
6	s_6	s_7			8
7			r_3		
8			r_2		
9	r_2	r_2			

The given grammar is in
CLR

Example 22.2 solution (continued) - CLR parsing table

State	Action			Goto	
	a	b	\$	S	A
0	s_3	s_4		1	2
1			Accept		
2	s_6	s_7			5
3,6	$s_{3,6}$	$s_{4,7}$			8,9
4,7	r_3	r_3	r_3		
5			r_1		
8,9	r_2	r_2	r_2		

For LALR merge:

- I3, I6
- I4, I7
- I8, I9

The given grammar is in LALR

Consider the following grammar:

$$\begin{array}{lcl} E & \rightarrow & T + E \mid T \\ T & \rightarrow & id \end{array}$$

Recall that this grammar is in SLR(1)

Make its CLR, LALR parsing tables and parse the string “id + id”

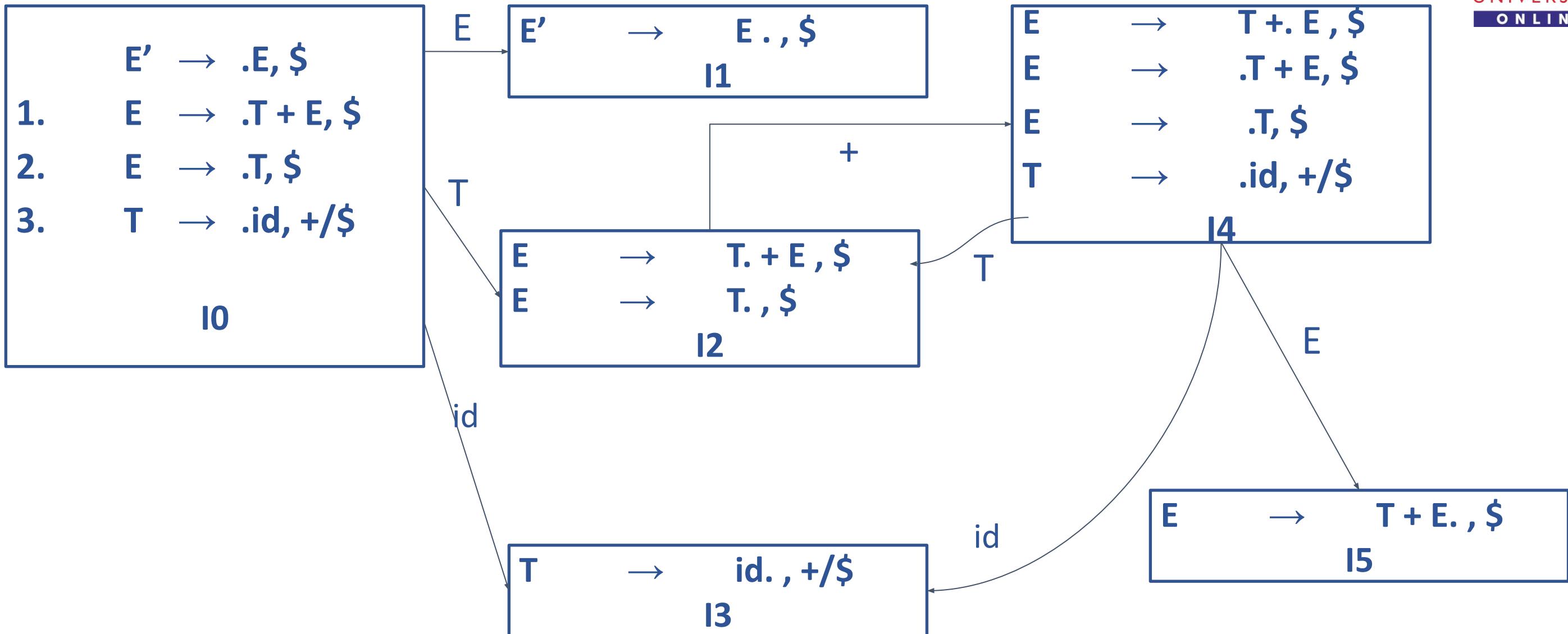
Consider the following grammar:

$$\begin{array}{lcl} E & \rightarrow & T + E \mid T \\ T & \rightarrow & id \end{array}$$

Augmenting and numbering the grammar:

1. $E' \rightarrow E$
2. $E \rightarrow T + E$
3. $E \rightarrow T$
4. $T \rightarrow id$

Example 22.5 solution (continued)



Example 22.2 solution (continued) - CLR parsing table

State	Action			Goto	
	id	+	\$	E	T
0	s_3			1	2
1			Accept		
2		s_4	r_2		
3		r_3	r_3		
4	s_3			5	2
5			r_1		

The given grammar is in
CLR and LALR

Consider the following grammar:

$$\begin{array}{lcl} S & \rightarrow & dA \mid aB \\ A & \rightarrow & bA \mid c \\ B & \rightarrow & bB \mid c \end{array}$$

Recall that this grammar is in LR(0)

Make its CLR, LALR parsing tables and parse the string “abc”

Consider the following grammar:

$$S \rightarrow dA \mid aB$$

$$A \rightarrow bA \mid c$$

$$B \rightarrow bB \mid c$$

Augmenting and numbering the grammar:

$$S' \rightarrow S$$

1. $S \rightarrow dA$

2. $S \rightarrow aB$

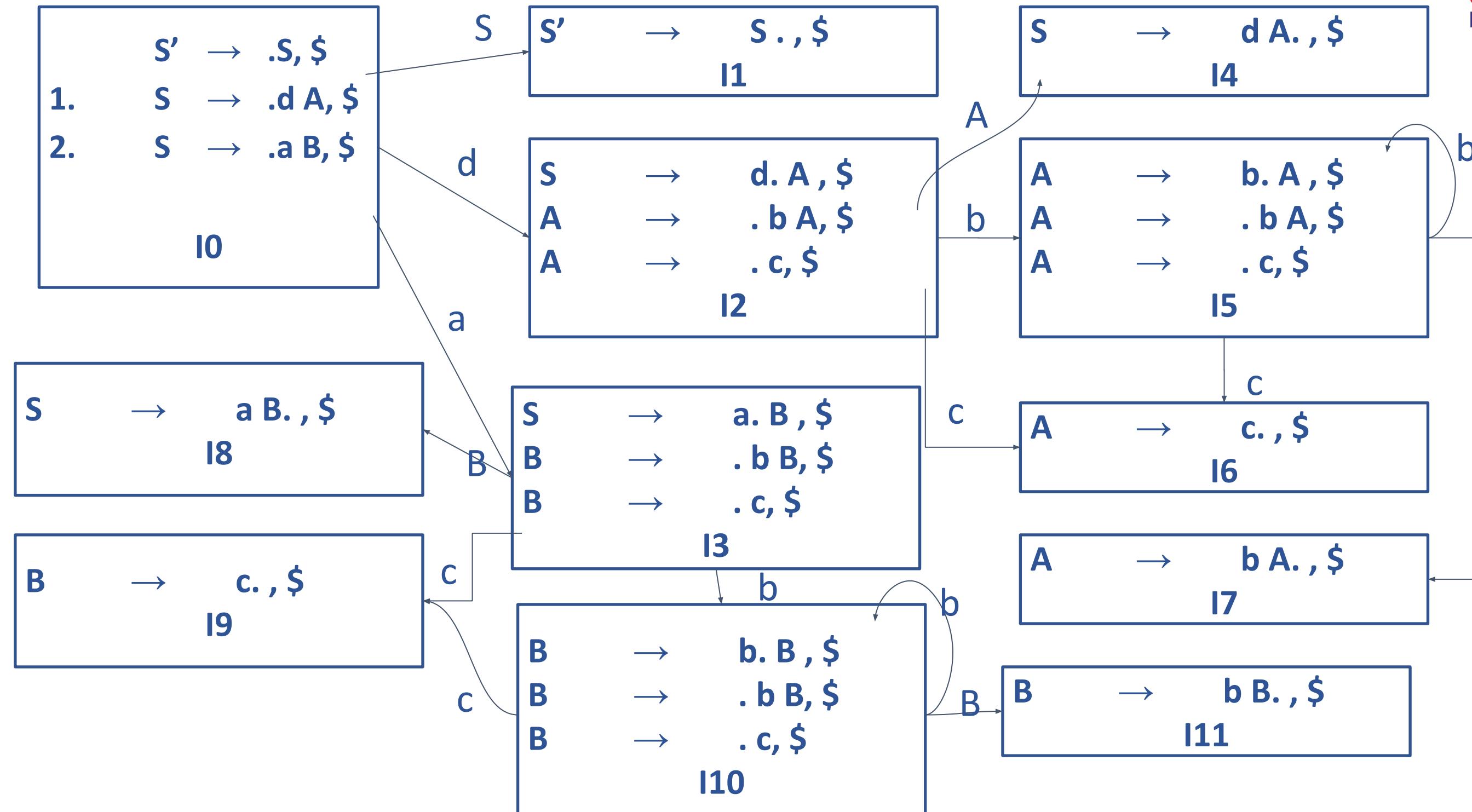
3. $A \rightarrow bA$

4. $A \rightarrow c$

5. $B \rightarrow bB$

6. $B \rightarrow c$

Example 22.6 solution (continued)



Example 22.2 solution (continued) - CLR parsing table

State	Action					Goto		
	a	b	c	d	\$	S	A	B
0	s_3				s_2	1		
1					Accept			
2		s_5	s_6				4	
3		s_{10}	s_9					8
4					r_1			
5		s_5	s_6				7	
6					r_4			
:	:	:		:	:	:	:	:

The given grammar is in
CLR and LALR

Example 22.2 solution (continued) - CLR parsing table

State	Action					Goto		
	a	b	c	d	\$	S	A	B
:	:	:	:	:	:	:	:	:
7					r_3			
8					r_2			
9					r_6			
10		s_{10}	s_9					11
11					r_5			

Grammar part of CLR and LALR

Consider the following grammar:

$$S \rightarrow SS^+ | SS^* | a$$

Recall that this grammar is in LR(0)

Make its CLR, LALR parsing tables and parse the string

“aa+aa*”

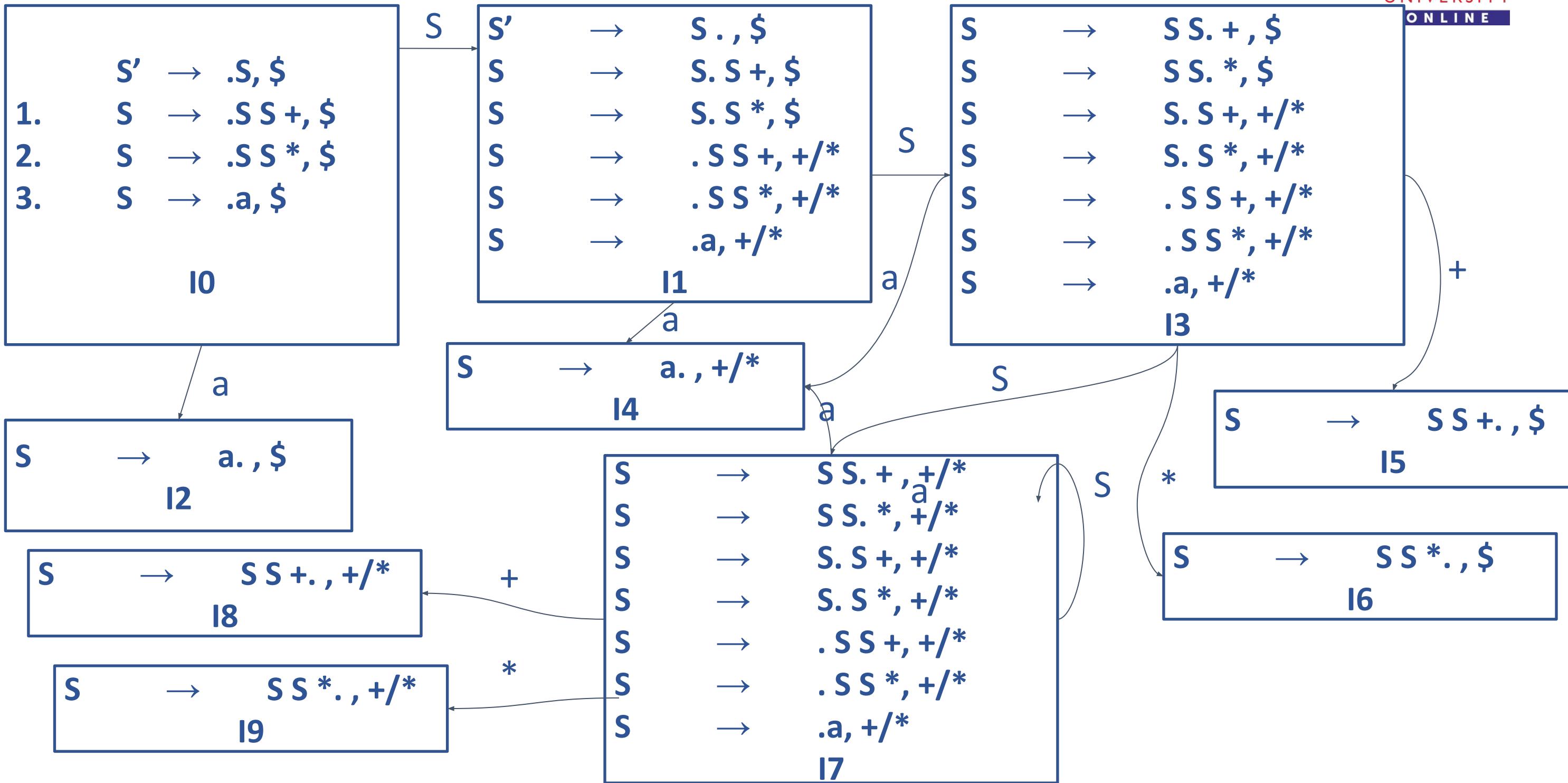
Consider the following grammar:

$$S \rightarrow SS^+ | SS^* | a$$

Augmenting and numbering the grammar:

- $S' \rightarrow S$
- 1. $S \rightarrow SS^+$
- 2. $S \rightarrow SS^*$
- 3. $S \rightarrow a$

Example 22.7 solution (continued)



Example 22.2 solution (continued) - CLR parsing table

State	Action				Goto
	a	+	*	\$	
0	s_2				1
1	s_4			Accept	3
2				r_3	
3	s_2	s_5	s_6		7
4		r_3	r_3		
5				r_1	
6				r_2	
:	:	:	:	:	:

Example 22.2 solution (continued) - CLR parsing table

State	Action					Goto
	a	+	*	\$	s	
:	:	:	:	:	:	
7	s_4	s_8	s_9			7
8		r_1	r_1			
9		r_2	r_2			

Grammar part of CLR

Example 22.2 solution (continued) - CLR parsing table

State	Action				Goto
	a	+	*	\$	
0	s_2				1
1	s_4			Accept	3
2,4		r_3	r_3	r_3	
3,7	$s_{2,4}$	$s_{5,8}$	$s_{6,9}$		7
5,8		r_1	r_1	r_1	
6,9		r_2	r_2	r_2	

For LALR merge:

- I2, I4
- I3, I7
- I5, I8
- I6, I9

Grammar part of LALR

Find out if the given grammar is in LR(0), SLR, LALR and CLR.

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

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

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

Is the given grammar in LL(1) and LALR(1)?

$$S \rightarrow (X \mid E \text{ sq }) \mid F ($$

$$X \rightarrow E) \mid F \text{ sq })$$

$$E \rightarrow A$$

$$F \rightarrow A$$

$$A \rightarrow \lambda$$

Is the language finite?

Is the given grammar in LL, LR(0), SLR, LALR and LR?

$$S \rightarrow (X \mid E) \mid F)$$

$$X \rightarrow E) \mid F]$$

$$E \rightarrow A$$

$$F \rightarrow A$$

$$A \rightarrow \lambda$$

Provide a production with the shortest RHS that introduces s/r conflict in SLR parser for the given grammar:

$$S \rightarrow A b$$

$$A \rightarrow c b A d$$

$$A \rightarrow c A d$$

$$A \rightarrow \lambda$$

Is the given grammar in LL, LR(0), SLR, LALR and LR?

$$S \rightarrow A b b x \quad | \quad B b b y$$

$$A \rightarrow x$$

$$B \rightarrow x$$

Specify the value of i and j such that the grammar in LL(i) and LR(j).

Intuitively, for two reasons:

1) Lookahead makes handle-finding easier.

- The LR(0) automaton says whether there could be a handle later on based on no right context.
- The LR(1) automaton can predict whether it needs to reduce based on more information.

2) More states encode more information.

- LR(1) lookaheads are very good because there's a greater number of states to be in.
- Goal: Incorporate lookahead without increasing the number of states.

LR parsers use shift and reduce actions to reduce the input to the start symbol.

- LR parsers cannot deterministically handle shift/reduce or reduce/reduce conflicts.
- However, they can nondeterministically handle these conflicts by guessing which option to choose.

Compiler Design

Error Handling

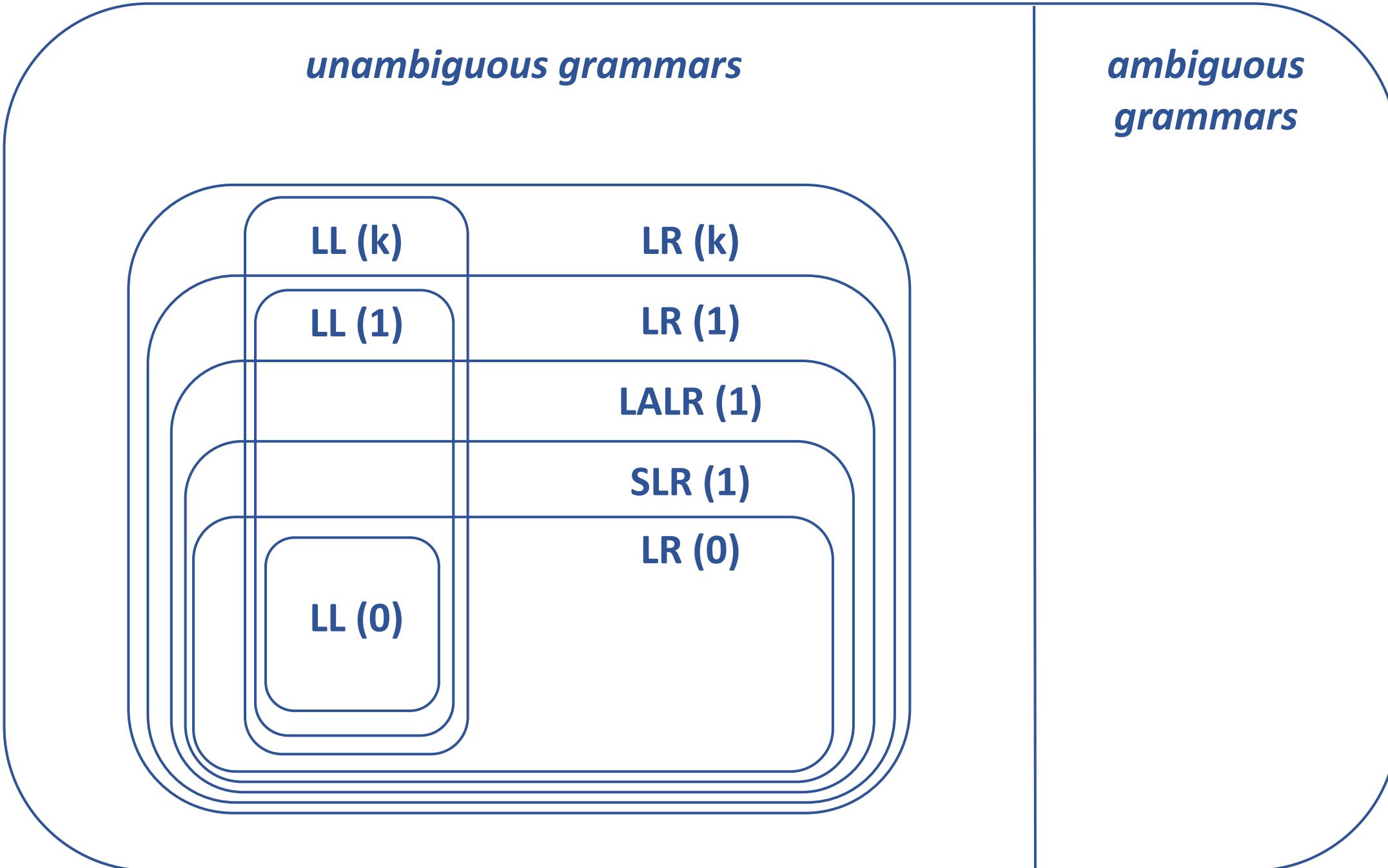


- What should the parser do when it encounters an error?
- Could just say “syntax error,” but we’d like more detailed messages.
- How do we resume parsing after an error?
[error-recovery strategies discussed in previous lectures]

CLR will not make any reduction if there is an error in the string. LALR and SLR may make a few reductions before declaring error.

Summary of all the table-driven bottom-up parsers

Parser	LR(0)	SLR(1) or SLR (Simple LR)	LALR(1) or LALR (LookAhead LR)	CLR(1) or CLR or LR(1) or LR (Canonical LR)
DFA	LR(0) automata LR(0) item		LR(1) automata LR(1) item = LR(0) item + lookahead	
Reduce Move Placement in Action part of the Parsing table	Place the reduce move in the entire row for the state that contains a final item	Place the reduce move only in the Follow(LHS).	Place the reduce move in the lookahead	Place the reduce move in the lookahead
Power	Least Powerful	More powerful than LR(0)	More powerful than SLR	Most Powerful
Number of States	n : Number of States in a parser $n(\text{LR}(0)) = n(\text{SLR}) = n(\text{LALR}) \leq n(\text{CLR})$			



Note that this diagram refers to **grammars**, not languages, e.g. there may be an equivalent LR(1) grammar that accepts the same language as another non-LR(1) grammar. No ambiguous grammar is LL(1) or LR(1), so we must either rewrite the grammar to remove the ambiguity or resolve conflicts in the parser table or implementation.

The hierarchy of LR variants is clear: every LR(0) grammar is SLR(1) and every SLR(1) is LALR(1) which in turn is LR(1). But there are grammars that don't meet the requirements for the weaker forms that can be parsed by the more powerful variations.

Error repair:

- Both LL(1) and LALR(1) parsers possess the valid prefix property. What is on the stack will always be a valid prefix of a sentential form. Errors in both types of parsers can be detected at the earliest possible point without pushing the next input symbol onto the stack.
- LL(1) parse stacks contain symbols that are predicted but not yet matched. This information can be valuable in determining proper repairs.
- LALR(1) parse stacks contain information about what has already been seen, but do not have the same information about the right context that is expected.
- This means deciding possible continuations is somewhat easier in an LL(1) parser.

Efficiency:

- Both require a stack of some sort to manage the input. That stack can grow to a maximum depth of n , where n is the number of symbols in the input.
- If you are using the runtime stack (i.e. function calls) rather than pushing and popping on a data stack, you will probably pay some significant overhead for that convenience (i.e. a recursive descent parser takes that hit).
- If both parsers are using the same sort of stack, LL(1) and LALR(1) each examine every non-terminal and terminal when building the parse tree, and so parsing speeds tend to be comparable between the two.



THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu



Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Kavya P K

Compiler Design

Unit 3: Syntax Directed Definitions

Preet Kanwal

Department of Computer Science & Engineering

Compiler Design

Lecture Overview



In this lecture, you will learn about -

- **Semantic Analysis**
- **What is Syntax Directed Translation?**
- **Semantic Rules**
- **Syntax-directed Definitions**
- **Types of Attributes**
- **S-Attributed SDD and its evaluation**

Compiler Design

Semantic Analysis



- Semantic Analysis computes additional information related to the meaning of the program once the syntactic structure is known.
- In typed languages as C, semantic analysis involves adding information to the symbol table and performing type checking.
- The information to be computed is beyond the capabilities of standard parsing techniques, therefore it is not regarded as syntax.
- For Semantic Analysis, we need both a Representation Formalism and an Implementation Mechanism.
- Syntax Directed Translations is an illustration of Representation Formalism.

What is a Syntax Directed Translation?

- The Principle of **Syntax Directed Translation** states that the meaning of an input sentence is related to its syntactic structure, i.e., to its Parse-Tree.
- Translations for programming language constructs guided by context-free grammars.
 - We associate **attributes** to the grammar symbols representing the language constructs.
 - Values for attributes are computed by **Semantic Rules** associated with grammar productions.

Compiler Design

Semantic Rules



There are two notations for attaching semantic rules:

1. **Syntax Directed Definitions** - High-level specification hiding many implementation details.
2. **Translation Schemes** - More implementation oriented, indicates the order in which semantic rules are to be evaluated.

Compiler Design

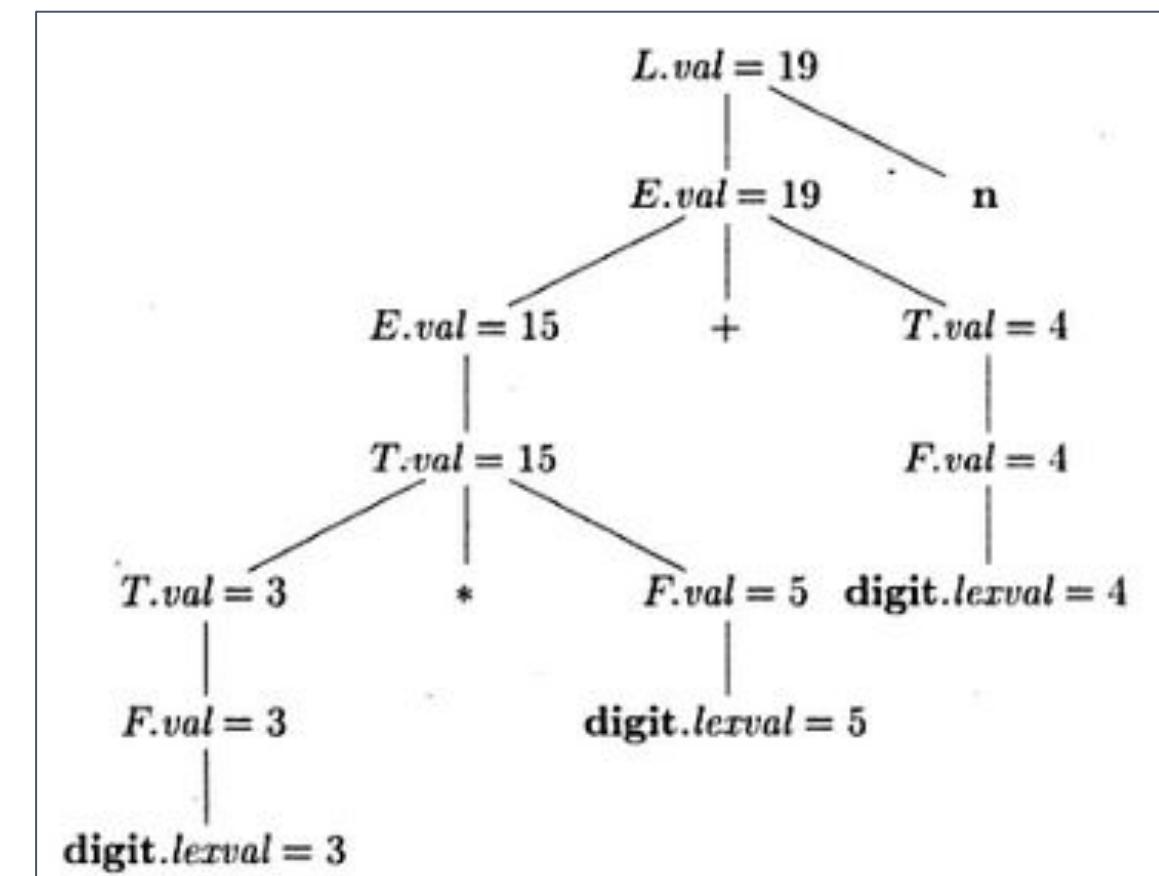
Semantic Rules



Evaluation of Semantic Rules may -

- **Generate Code**
- **Insert information into the Symbol Table**
- **Perform Semantic Check**
- **Issue error messages**
- **etc.**

- Syntax Directed Definitions (SDD) are a generalization of context-free grammars in which -
 - Grammar symbols have an associated set of attributes.
 - Productions are associated with Semantic Rules for computing the values of attributes.
- This formalism generates Annotated Parse Trees, where each node of the tree is a record with a field for each attribute.
 - Example - F.val indicates the attribute val for grammar symbol F



Formally, SDDs are defined as follows -

Each production, $A \rightarrow \alpha$, is associated with a set of semantic rules -

$b := f(c_1, c_2, \dots, c_k)$

where f is a function,

b is either

1. A synthesized attribute of A , and c_1, c_2, \dots, c_k are attributes of the grammar symbols of the production, or
2. An inherited attribute of a grammar symbol in α , and c_1, c_2, \dots, c_k are attributes of grammar symbols in α or attributes of A .

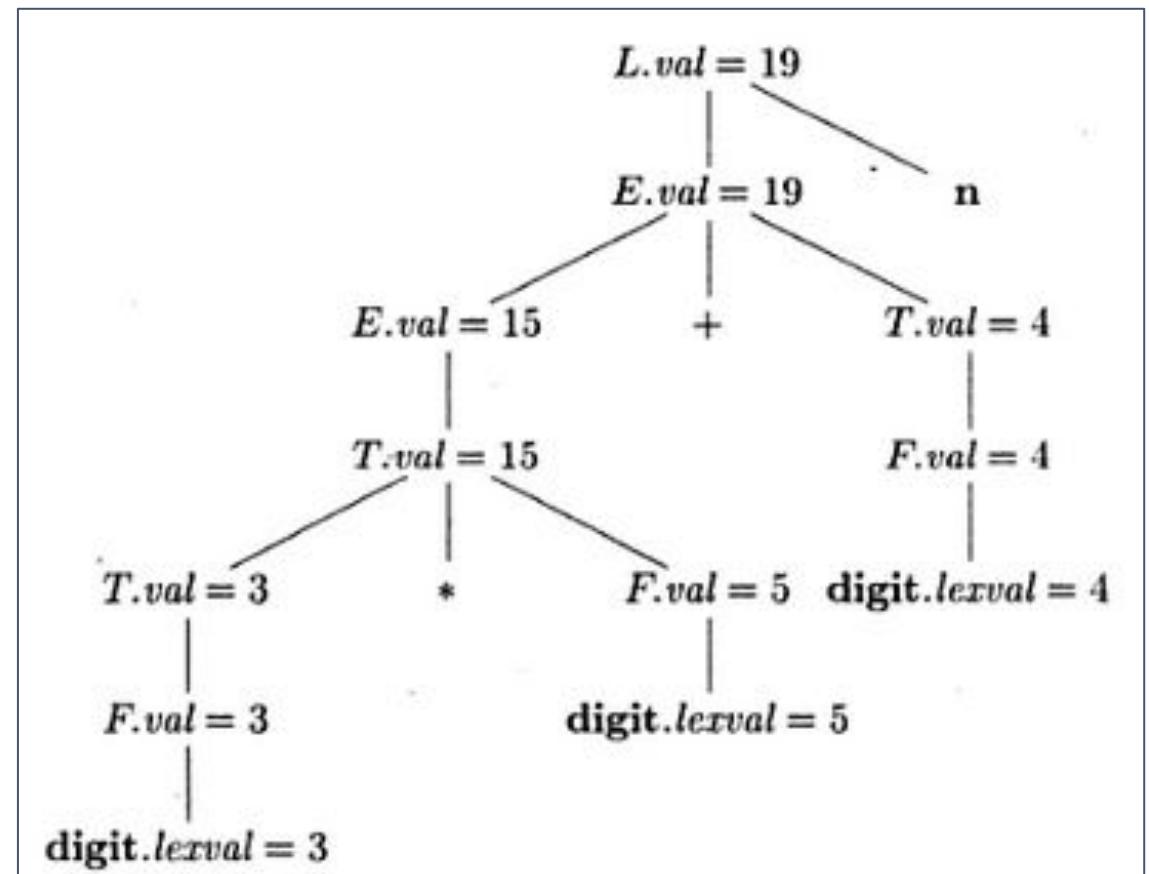
Example - SDD to implement a Simple Desk Calculator

Production	Semantic Rule
$E \rightarrow E_1 + T$	$\{ E.val = E_1.val + T.val \}$
$E \rightarrow T$	$\{ E.val = T.val \}$
$T \rightarrow T_1 * F$	$\{ T.val = T_1.val * F.val \}$
$T \rightarrow F$	$\{ T.val = F.val \}$
$F \rightarrow \text{num}$	$\{ F.val = \text{num.lexval} \}$
$F \rightarrow \text{id}$	$\{ F.val = \text{id.lexval} \}$

num.lexval gets its value from the lexical analyser

- There are two kinds of attributes -
 - **Synthesized Attributes** - computed from the values of the attributes of the children nodes.
 - **Inherited Attributes** - computed from the values of the attributes of both the siblings and the parent nodes.

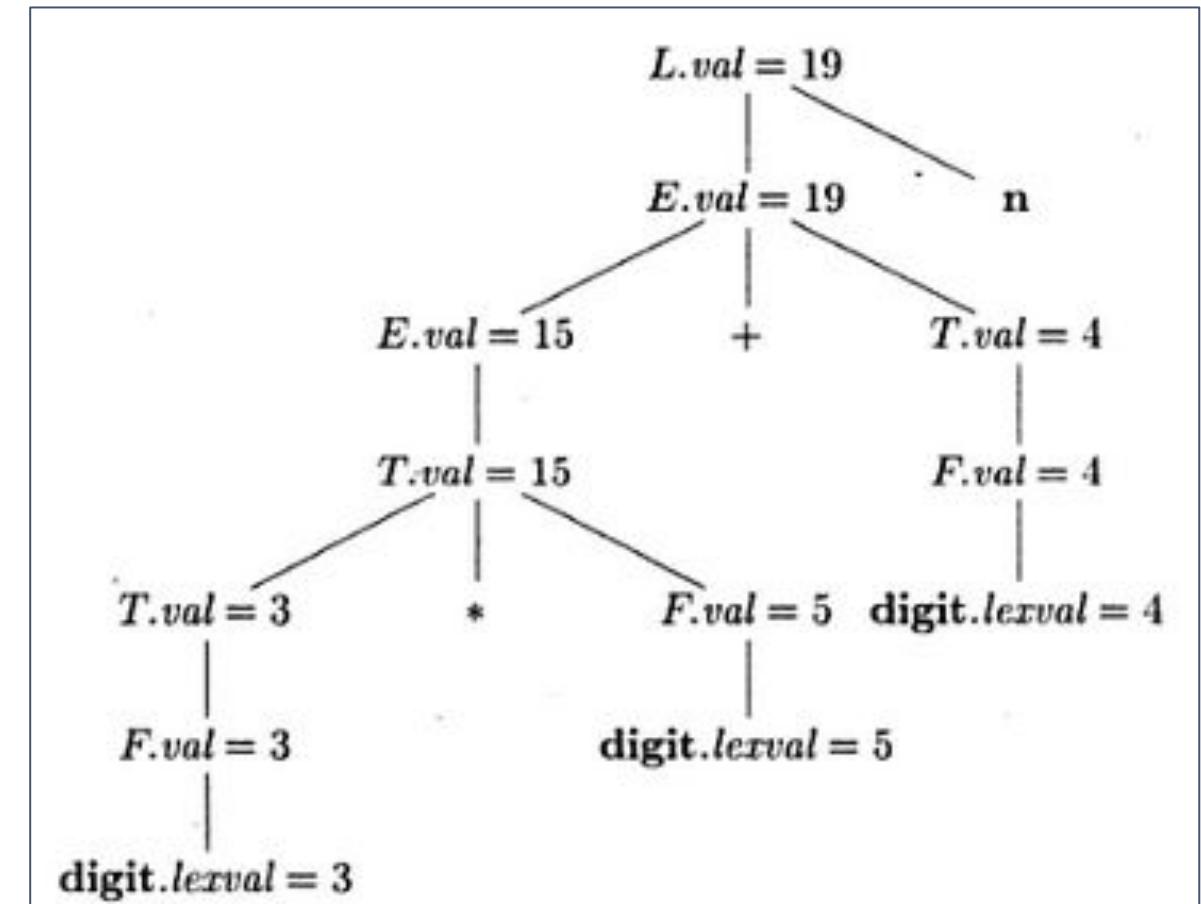
- The Syntax Directed Definition associates to each non terminal a synthesized attribute called **val**.
- Terminal symbols are assumed to have synthesised attributes supplied by the lexical analyser.
 - For example, in the given parse tree **digit.lexval** gets its value from the lexer.



- Procedure calls define values of Dummy synthesized attributes of the non-terminal on the left-hand side of the production.
 - For example, print in the given table -

Production	Semantic Rule
$L \rightarrow E_n$	$print(E.val)$
$E \rightarrow E_1 + T$	$\{ E.val = E_1.val + T.val \}$
$E \rightarrow T$	$\{ E.val = T.val \}$
$T \rightarrow T_1 * F$	$\{ T.val = T_1.val * F.val \}$
$T \rightarrow F$	$\{ T.val = F.val \}$
$F \rightarrow (E)$	$\{ F.val = E.val \}$
$F \rightarrow \text{digit}$	$\{ F.val = \text{digit}.lexval \}$
$F \rightarrow \text{id}$	$\{ F.val = \text{id}.lexval \}$

- An SDD with only synthesized attributes is called an **S-Attributed Definition**.
- **Evaluation order - Semantic rules in a S-Attributed Definition can be evaluated by a bottom-up or Post-Order traversal of the parse-tree.**
- The above arithmetic grammar is an example of an **S-Attributed Definition**.
- The figure represents the annotated parse-tree for the input **3*5+4n**.



Compiler Design

Inherited Attributes

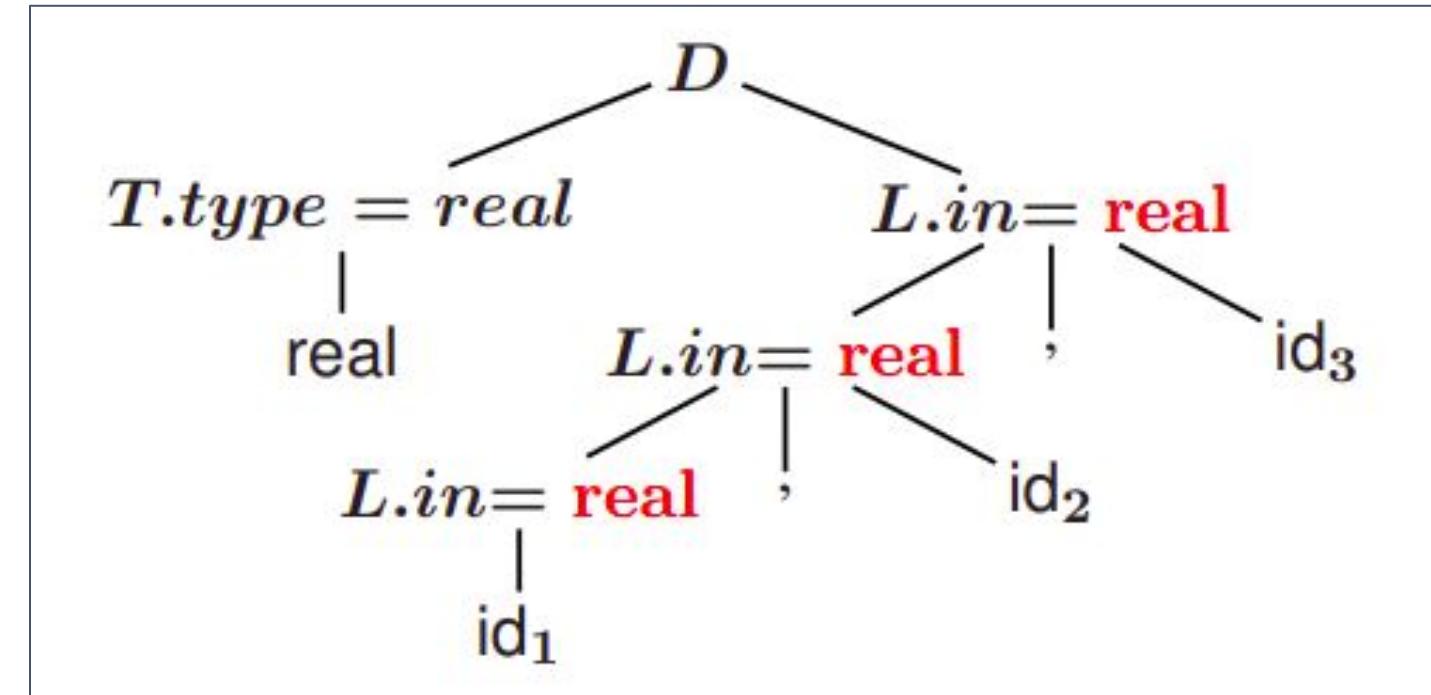


- Inherited Attributes are useful for expressing the dependence of a construct on the context in which it appears.
- Unlike synthesized attributes, the order in which the inherited attributes of the children are computed is important.
- Inherited attributes of the children can depend from both left and right siblings.
- Evaluation Order - Inherited attributes cannot be evaluated by a simple PreOrder traversal of the parse-tree.
 - Exception - Inherited attributes that do not depend on right children can be evaluated by a classical PreOrder traversal.

- Consider the following syntax directed definition for Type Declarations.
- The non terminal T has a synthesized attribute $type$ determined by the keyword in the declaration.
- The production $D \rightarrow T L$ is associated with the semantic rule $L.in = T.type;$ which set the inherited attribute $L.in$

Production	Semantic Rule
$D \rightarrow T L$	$\{ L.in = T.type; \}$
$T \rightarrow \text{int}$	$\{ T.type = \text{integer}; \}$
$T \rightarrow \text{real}$	$\{ T.type = \text{float}; \}$
$L \rightarrow L_1 , id$	$\{ L_1.in = L.in;$ $\text{addType}(id.entry, L.in); \}$
$L \rightarrow id$	$\{ \text{addType}(id.entry, L.in); \}$

- The given diagram illustrates the annotated parse-tree for the input **real id1, id2, id3**
- **L.in** is then inherited top-down the tree by the other L-nodes.
- At each **L-node** the procedure **addtype** inserts the type of the identifier into the symbol table.



Evaluating an SDD over a given input consists of the following steps -

- 1. Construct Parse Tree for given input.**
- 2. Construct Dependency graph.**
- 3. Topologically sort the nodes of Dependency graph.**
- 4. Produce as output Annotated Parse Tree.**



THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu