

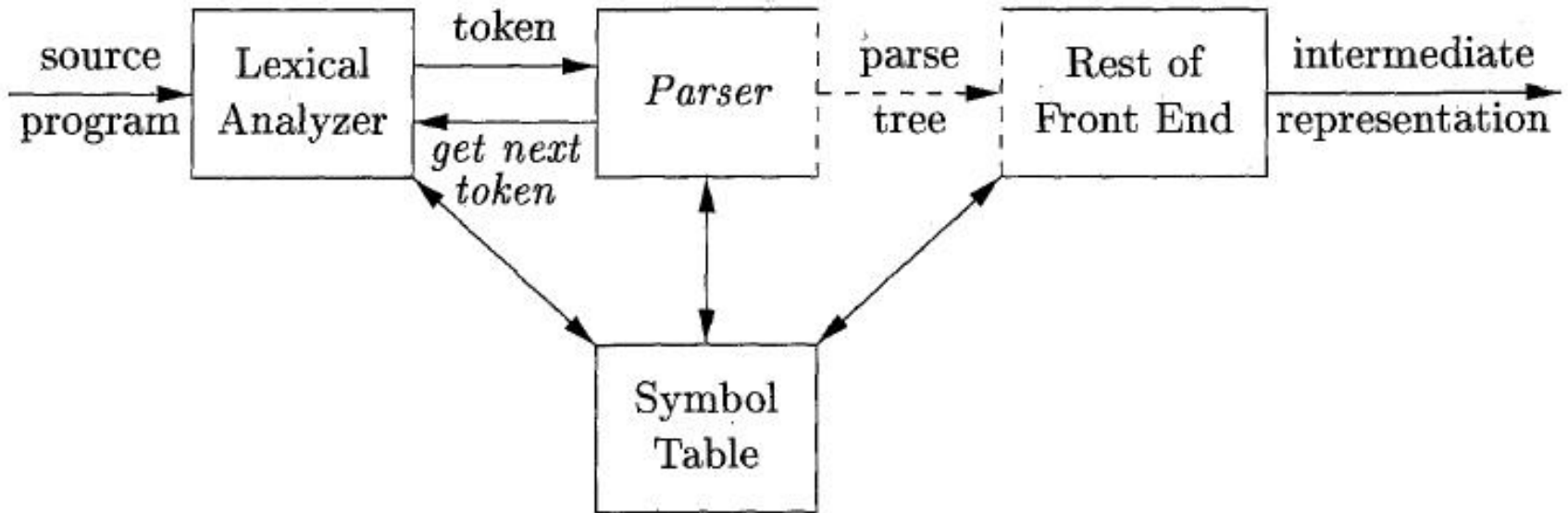


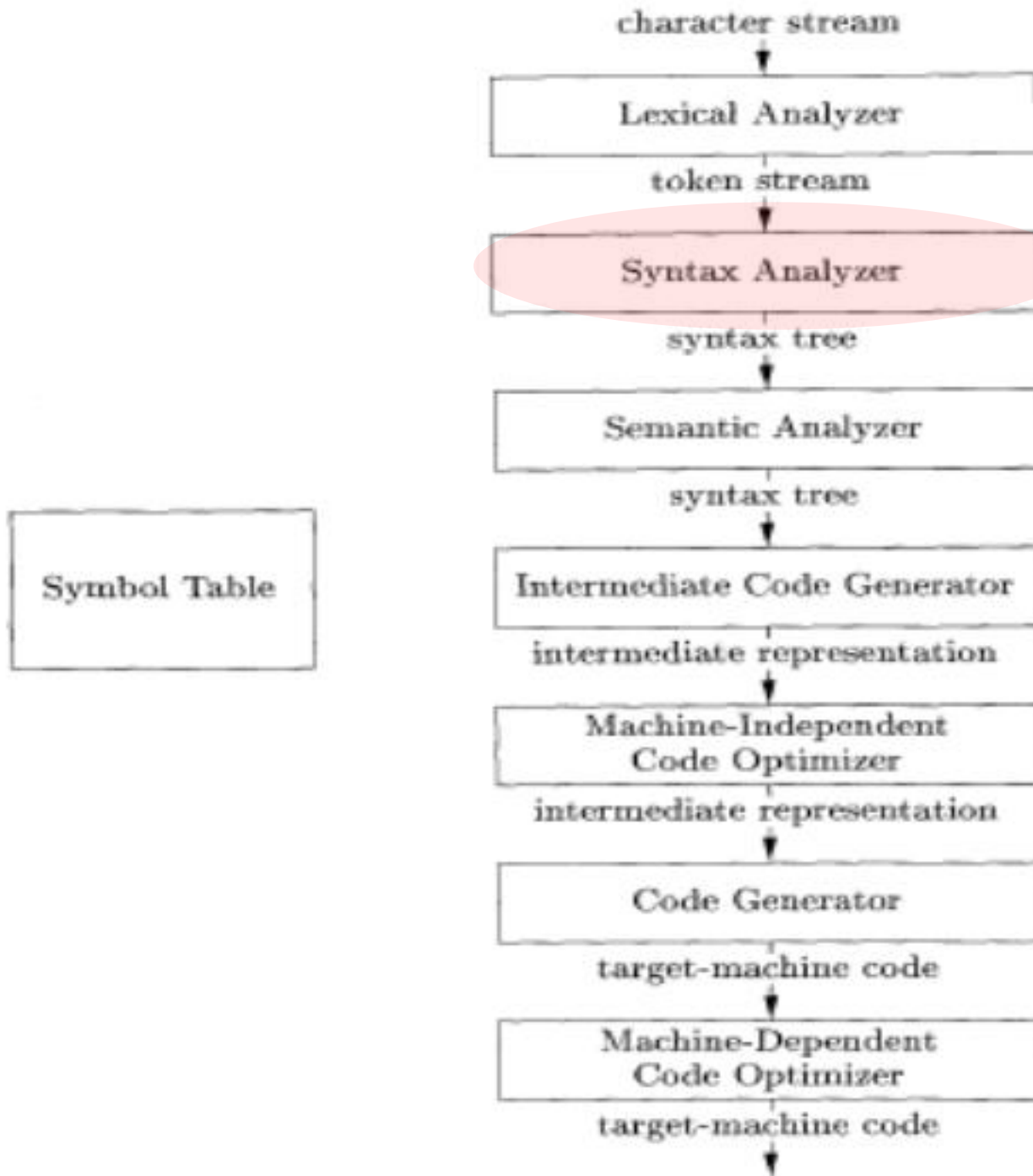
Compiler Principle ——Syntax Analysis

Zhizheng Zhang
Southeast University



1. The Role of Syntax Analyzer

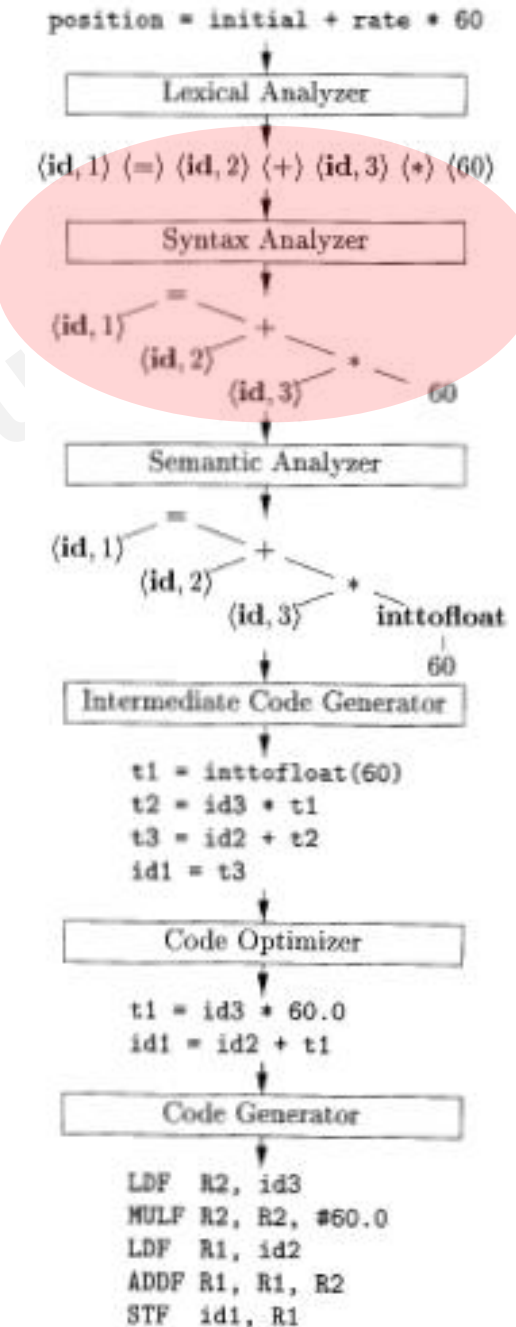






Example.

$\text{position} = \text{initial} + \text{rate} * 60$



Main Tasks

1. Generation of error messages.
2. Recover Errors.
3. *Syntax analysis, where the parser **Verifies** the sequence of tokens and produces the parse tree as output by **the sentential construction laws** in one of the three ways:*
 - a) Universal way
 - b) Top-Down
 - c) Bottom-Up



1. 1 An Example

<sentence> → <Subject><Predicate>

<Subject> → adjective noun

<Subject> → noun

<Predicate> → verb <Object>

<Object> → adjective noun

<Object> → noun



“young men like pop music”

Lexical Analyzer

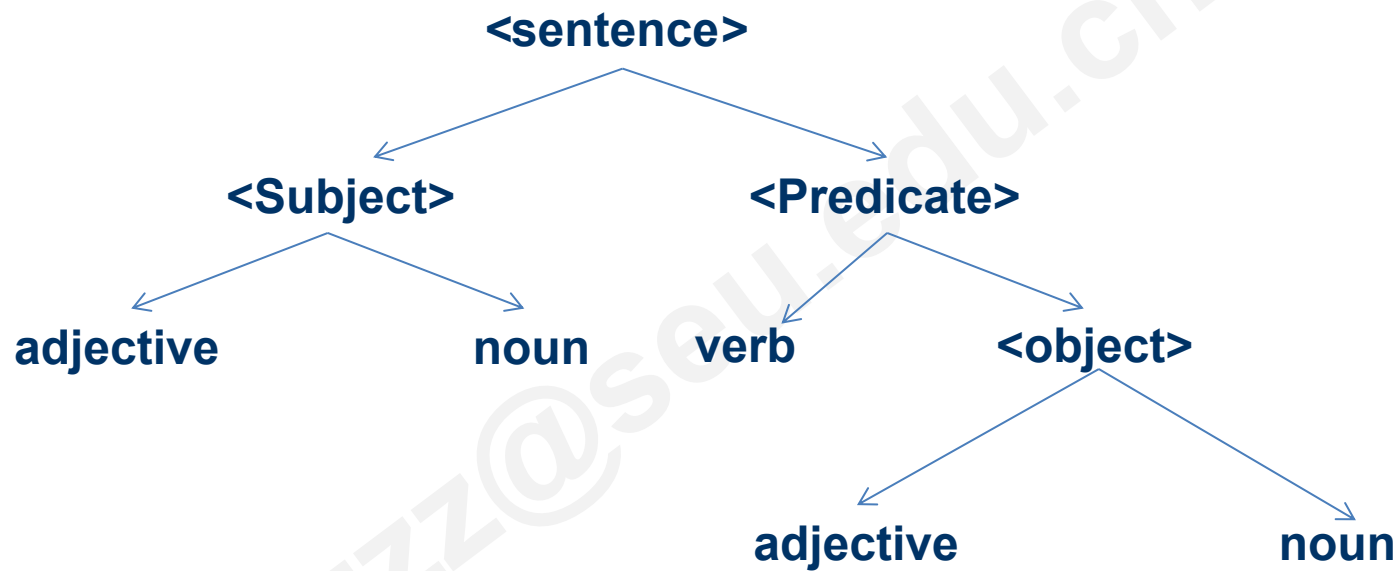
“(adjective,) (noun,) (verb,) (adjective,) (noun,)”

????

Top-Down

<sentence> → <Subject><Predicate>
→ adjective noun <Predicate>
→ adjective noun verb<object>
→ adjective noun verb adjective noun

Bottom-Up





1. 2 Errors Handling

Common programming errors include.

- a) **Lexical errors** include misspellings of identifiers, keywords, or operators - e.g., the use of an identifier *elipsesize* instead of *ellipsesize* – and missing quotes around text intended as a string.
- b) **Syntactic errors** include misplaced semicolons or extra or missing braces; that is, '("(" or ")".



- c) Semantic errors include type mismatches between operators and operands.** An example is a return statement in a Java method with result type void.
- d) Logical errors can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator = instead of the comparison operator ==.** The program containing = may be well formed; however, it may not reflect the programmer's intent.



NOTE:

At the very least, it must report the place in the source program where an error is detected.

recovery strategies:

a) panic-mode,

With this method, on discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found.

b) phrase-level,

On discovering an error, a parser performs local correction on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the parser to continue. A typical local correction is to replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon.



c) error-productions,

c) global-correction.

2. Context-free Grammar

A context-free grammar has four components:

- A set of terminal symbols.
- A set of nonterminals.

A set of productions. Each production consists of a nonterminal, called the head or left side of the production, an arrow, and a sequence of terminals and/or nonterminals, called the *body or right side of the production*.

- A start symbol.



Example. The grammar defines simple arithmetic expressions.

- a) **Terminal Symbols** {id, +, -, *, /, {, } }
- b) **Nonterminals** {expression, term, factor}
- c) **Productions**
 - expression \rightarrow expression + term
 - expression \rightarrow expression – term
 - expression \rightarrow term
 - term \rightarrow term * factor
 - term \rightarrow term / factor
 - term \rightarrow factor
 - factor \rightarrow (expression)
 - factor \rightarrow id
- d) **Start Symbol** *expression*.



For convenient description,

- I. Uppercase letters are used to denote nonterminals.**
- II. Lowercase letters represent strings of terminal symbols.**
- III. Lowercase Greek letters represent strings of terminal or nonterminal symbols**
- IV. Uppercase letter 'S' is used to represent the start symbol.**
- V. '|' means 'or'.**



Example. The grammar defines simple arithmetic expressions.

- a) **Terminal Symbols** {id, +, -, *, /, {, } }
- b) **Nonterminals** {E, T, F}
- c) **Productions**
 - $E \rightarrow E + T \mid E - T \mid T$
 - $T \rightarrow T * F \mid T / F \mid F$
 - $F \rightarrow (E) \mid \text{id}$
- d) **Start Symbol** E.



2.1 Derivation

Derivation.

Beginning with the start symbol, each rewriting step replaces a nonterminal by the body of one of its productions.



Example. Consider a grammar

$$E \rightarrow E + E \mid E - E \mid -E \mid (E) \mid id$$

A derivation is

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$$

Sentential form

If $S \xRightarrow{*} \alpha$, where S is the start symbol of a grammar G , we say that α is a *sentential form* of G .

Sentence

A *sentence* of G is a sentential form without nonterminals.

Language

The *language* generated by a grammar G is its set of sentences.

Leftmost Derivation

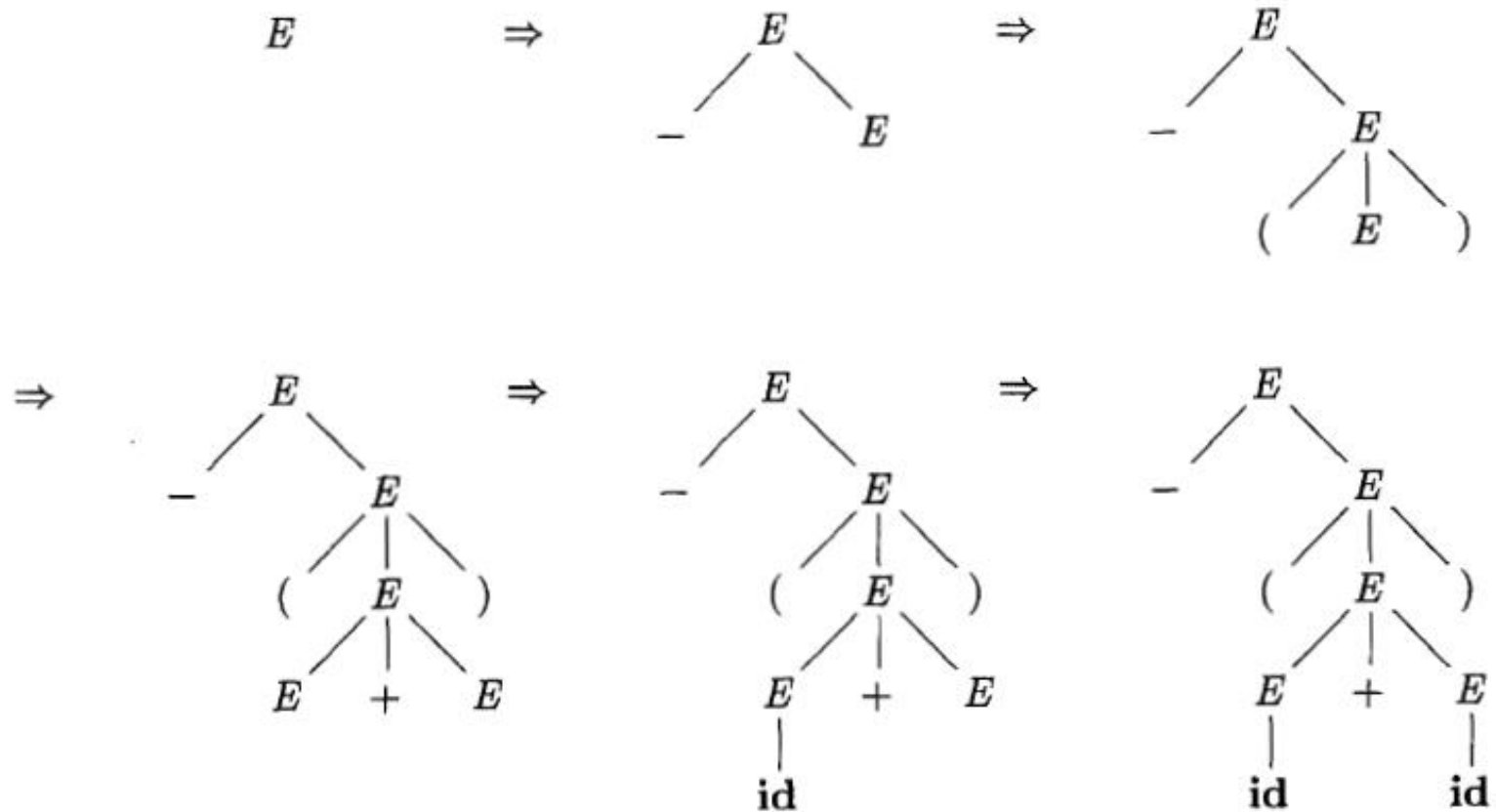
The leftmost nonterminal in each sentential is always chosen

$$E \Rightarrow_{lm} -E \Rightarrow_{lm} -(E) \Rightarrow_{lm} -(E + E) \Rightarrow_{lm} -(\text{id} + E) \Rightarrow_{lm} -(\text{id} + \text{id})$$

Rightmost Derivation



2.2 Parse Tree



A parse tree is a tree-based representation of derivation, where

- A vertex denotes a grammar symbol.
- An edge represents the father-children relationship between two grammar symbols among a derivation.

2.3 Ambiguity

A grammar that produces more than one parse tree for some sentence is said to be *ambiguous*.

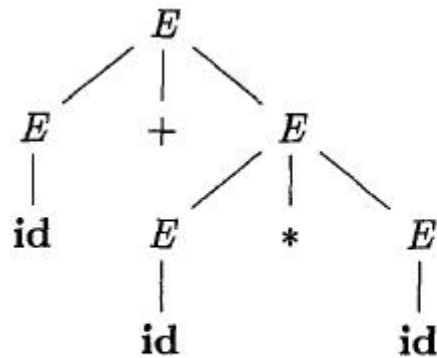
*Put another way, an **ambiguous** grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.*



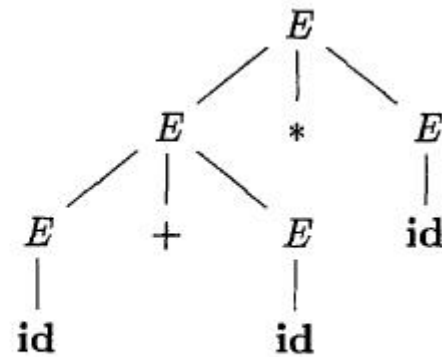
Example. Consider a grammar

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

For **id+id*id**, we have



(a)



(b)

2.4 CFG and RE

Why do we not represent the patterns of sentences using regular expressions?

Answer: Regular expressions are too weak to express the patterns of sentences.

Proposition.

For any RE r , a CFG G can be constructed such that $L(r) = L(G)$ by the following rules:

1. For each state i of the NFA, create a nonterminal A_i .
2. If state i has a transition to state j on input a , add the production $A_i \rightarrow aA_j$. If state i goes to state j on input ε , add the production $A_i \rightarrow A_j$.
3. If i is an accepting state, add $A_i \rightarrow \varepsilon$.
4. If i is the start state, make A_i be the start symbol of the grammar.

On the other hand, the language

$$L = \{a^n b^n \mid n > 1\}$$

has its CFG but not a RE.

**Please write down the
CFG of this language.**

NOTE: Grammars, are most useful for describing nested structures such as balanced parentheses, matching begin-end's, corresponding if-then-else's, and so on. These nested structures cannot be described by regular expressions.



**Thus, the approach to lexical analysis
does not work for syntax analysis.**



3. A Short Introduction of Parsing

Seu_zzz@seu.edu.cn

E.g. 1) $S \rightarrow xAy$ 2) $A \rightarrow **$ 3) $A \rightarrow *$, and Verify “ $x*y$ ”

Top-Down:

(1) Left-most derivation

● $S \rightarrow xAy \rightarrow x*y$

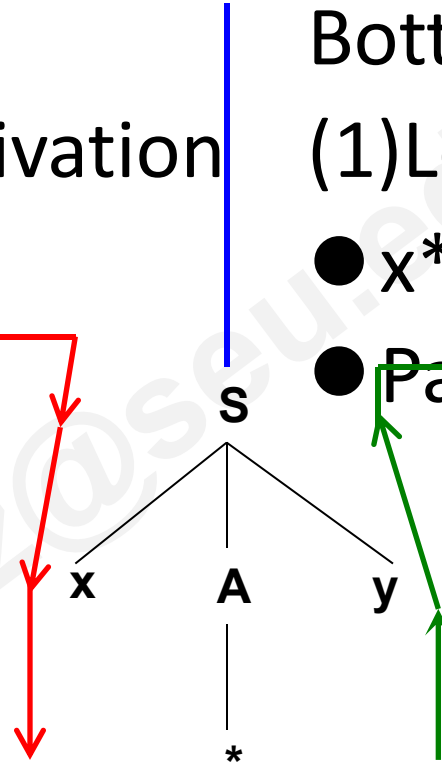
● Parser Tree

Bottom-Up:

(1) Left-most reduction

● $x*y \rightarrow xAy$

● Parser Tree



How codes?



PDA
Model

Top-Down

Bottom-Up

$x * y \$$

controller
rules

output

1,3

S
\$

Sentential
form

?

1) $S \rightarrow xAy$ 2) $A \rightarrow **$ 3) $A \rightarrow *,$

$x * y \$$

controller
rules

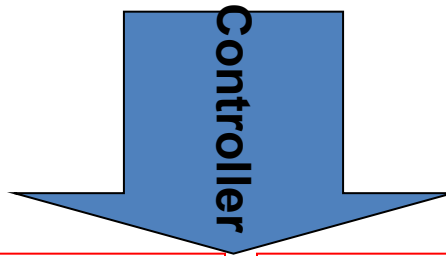
output

3,1

\$

String

?



T-D PDA Controller

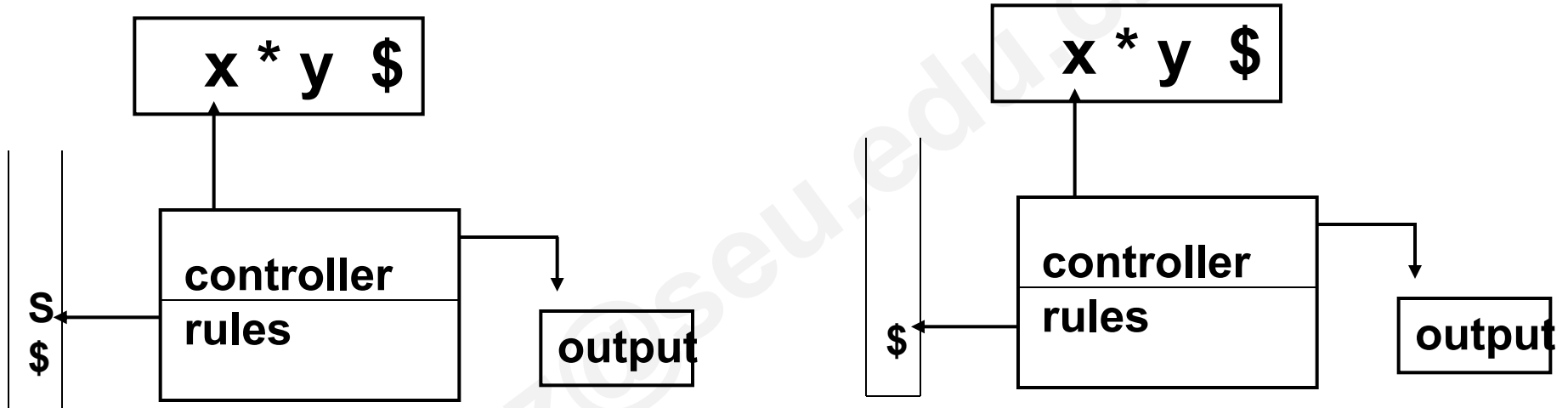
- IF “x” is the top of the stack and is a non-terminal, then select a production rule “ $x \rightarrow \dots$ ” randomly, replace “x” with the right hand side of the rule, and output the label the rule — **derivation**.
- IF “x” is the top of the stack and is same to that under the reading head, then... — **Matching**.
- IF (2) fail, then make a backtracking action to the scene before the last derivation and select a new rule — **backtracking**
- IF there no new rule, **fail**
- IF there is only “\$” in the stack, and “\$” is under the reading point, **success**

B-U PDA Controller

- IF the several top symbols in stack is a Handling, then **reduction**,
- **else if** “\$” is under the reading point then fail, **else Move** the symbol under reading head into stack.
- IF there is only \$S in the stack, and “\$” is under the reading head, **success**

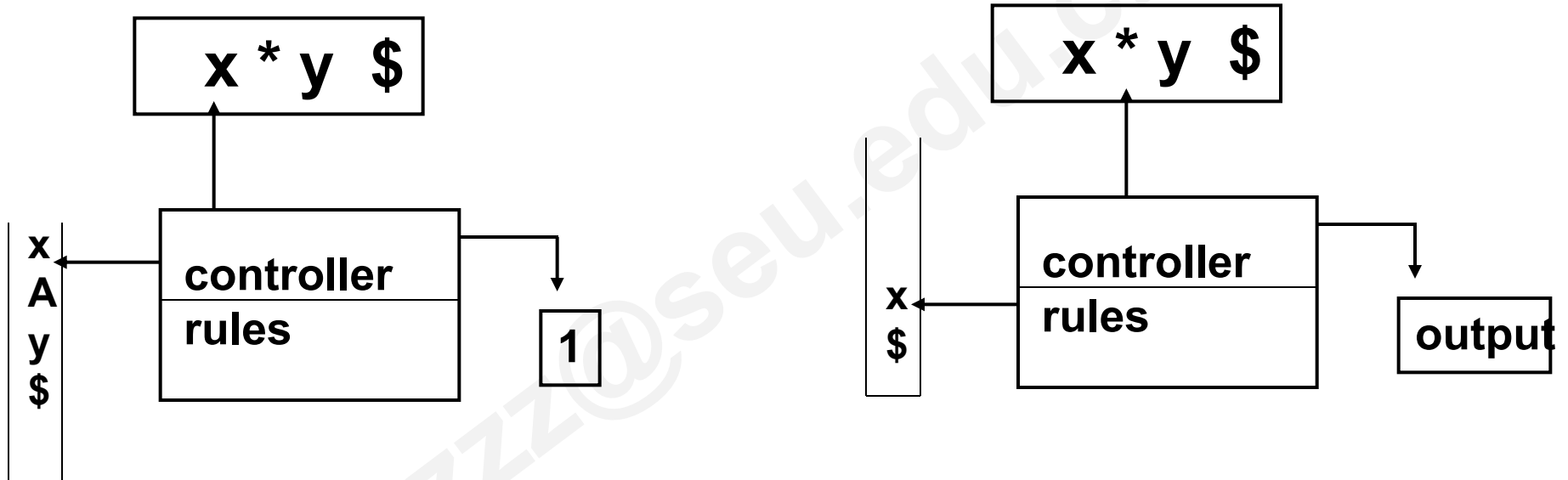


E.G.



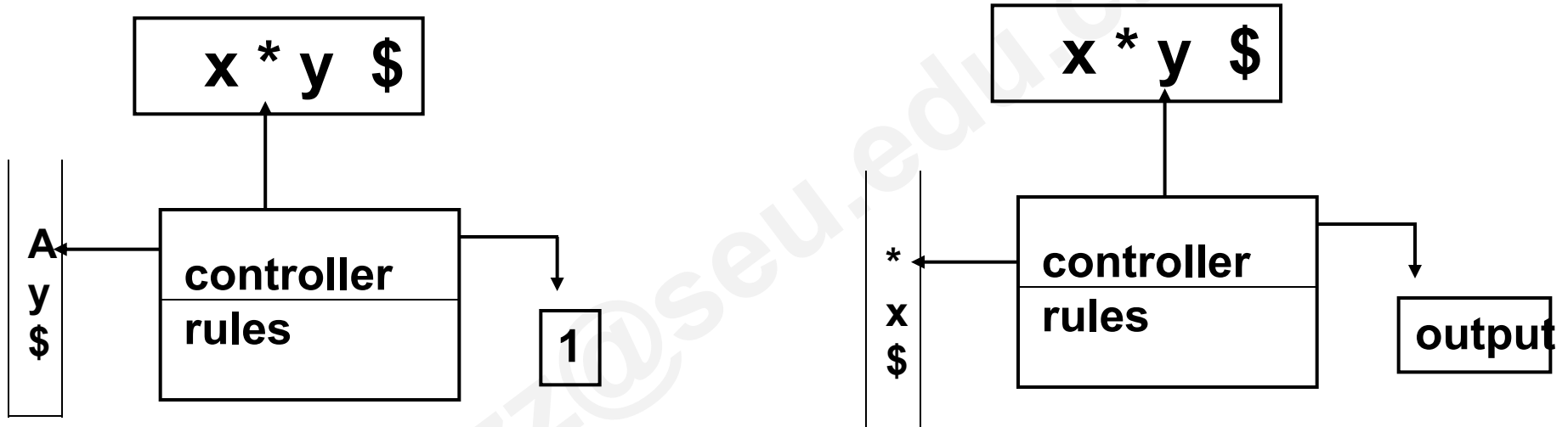


E.G.



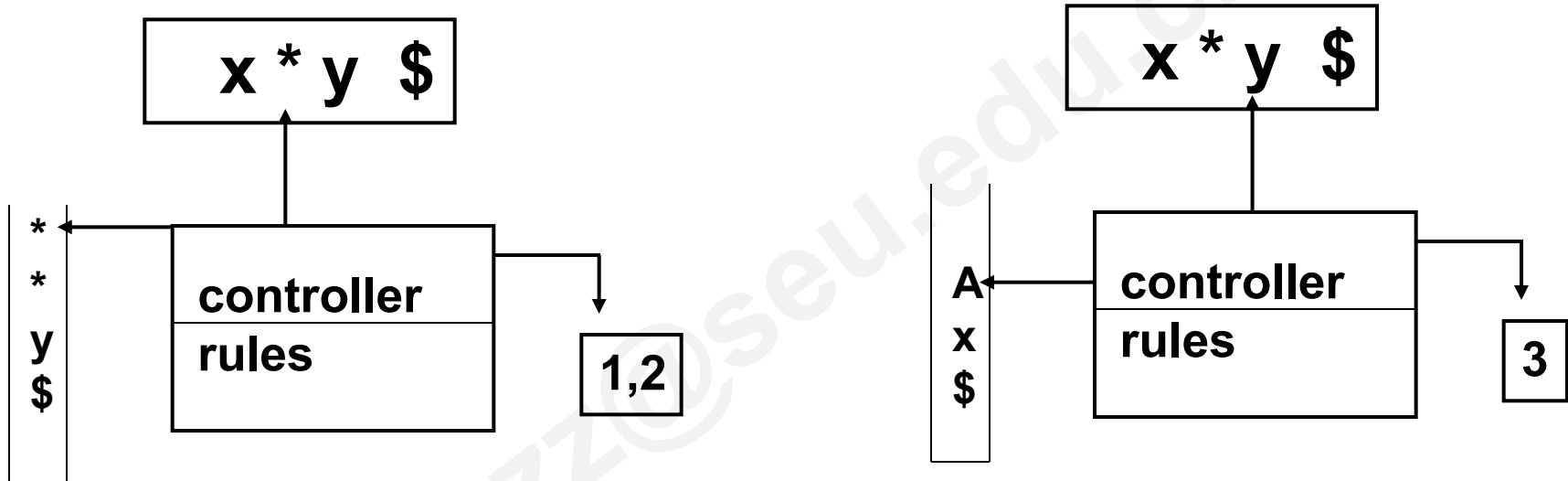


E.G.



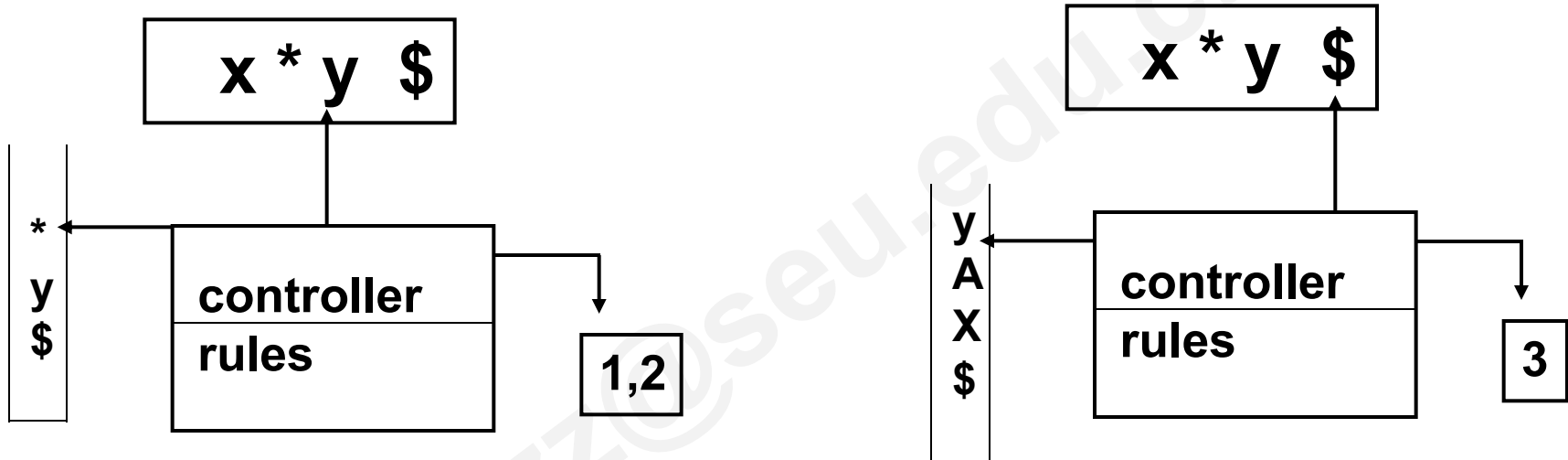


E.G.



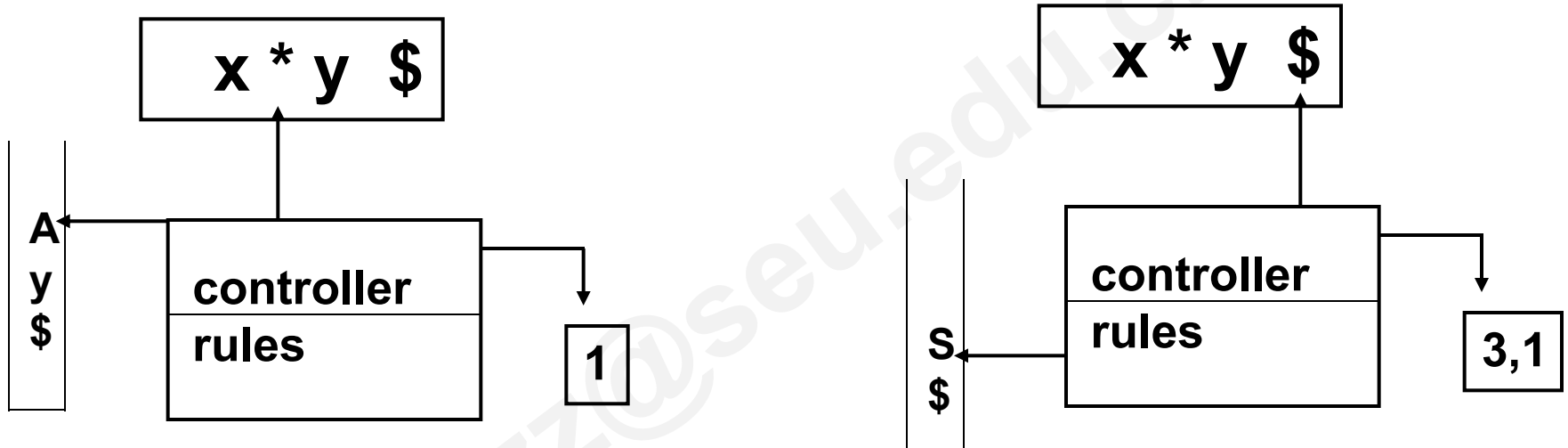


E.G.



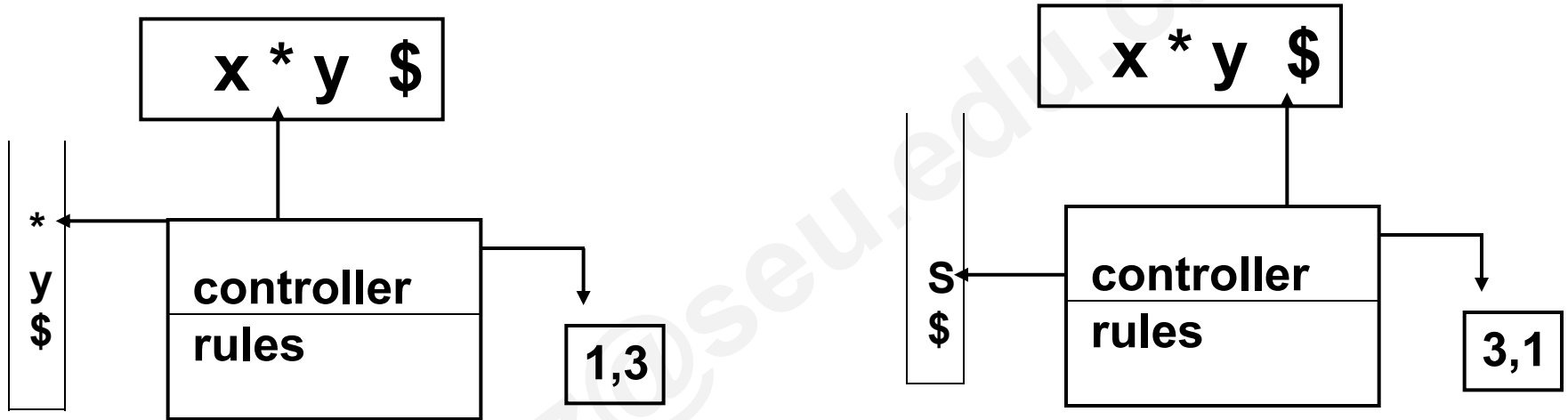


E.G.



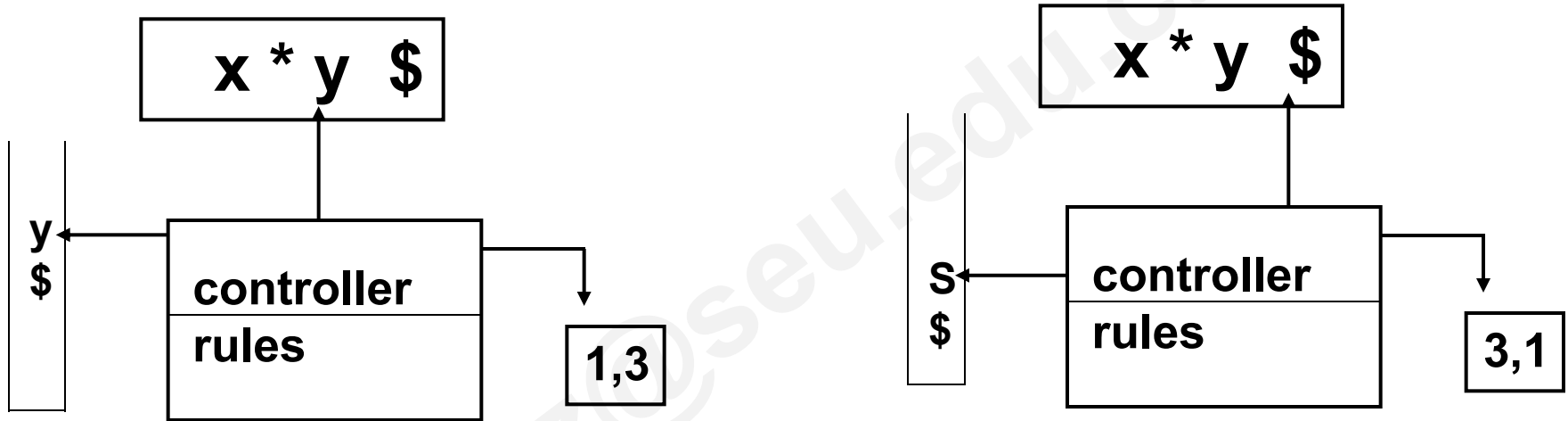


E.G.



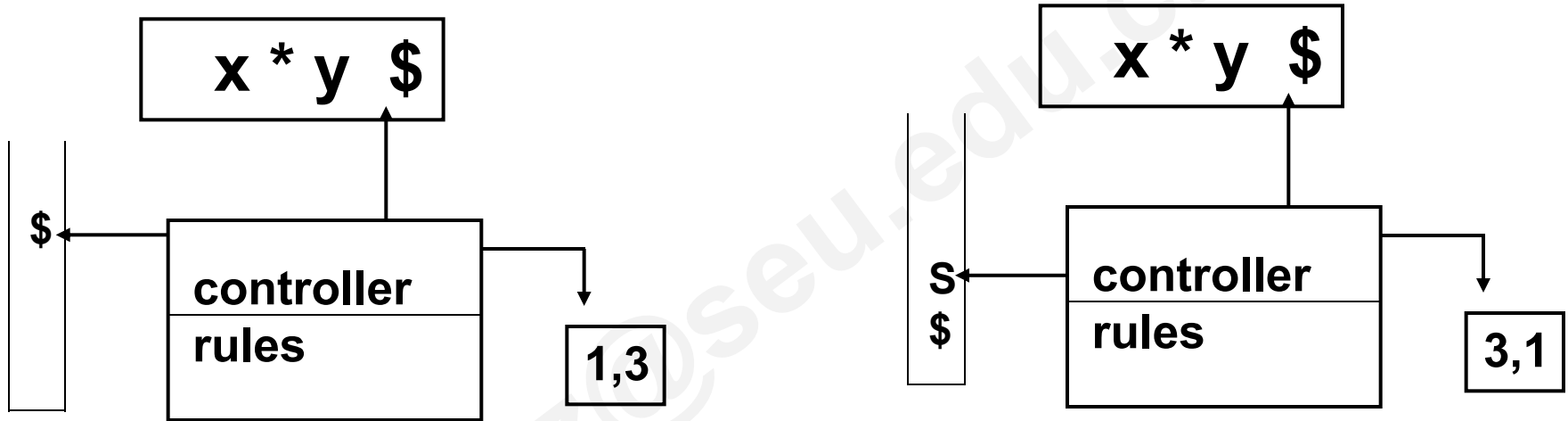


E.G.





E.G.





Discussion

1. How to choose a production for each derivation/reduction?
2. **Ambiguity** causes.....
3. **Left recursion** causes
4. **Backtracking** causes
5. **Left common factor**
6. **ϵ -production** causes...
7. **Non-context-free** grammar.....



4. Top-Down Parsing

Seu_zzz@seu.edu.cn



4.1 Writing a Grammar

Task: writing a grammar satisfying

- ϵ -free.
- Without ambiguity.
- Without left recursion.
- Without common factor.
- Context-free



4.1.1 Eliminating Ambiguity

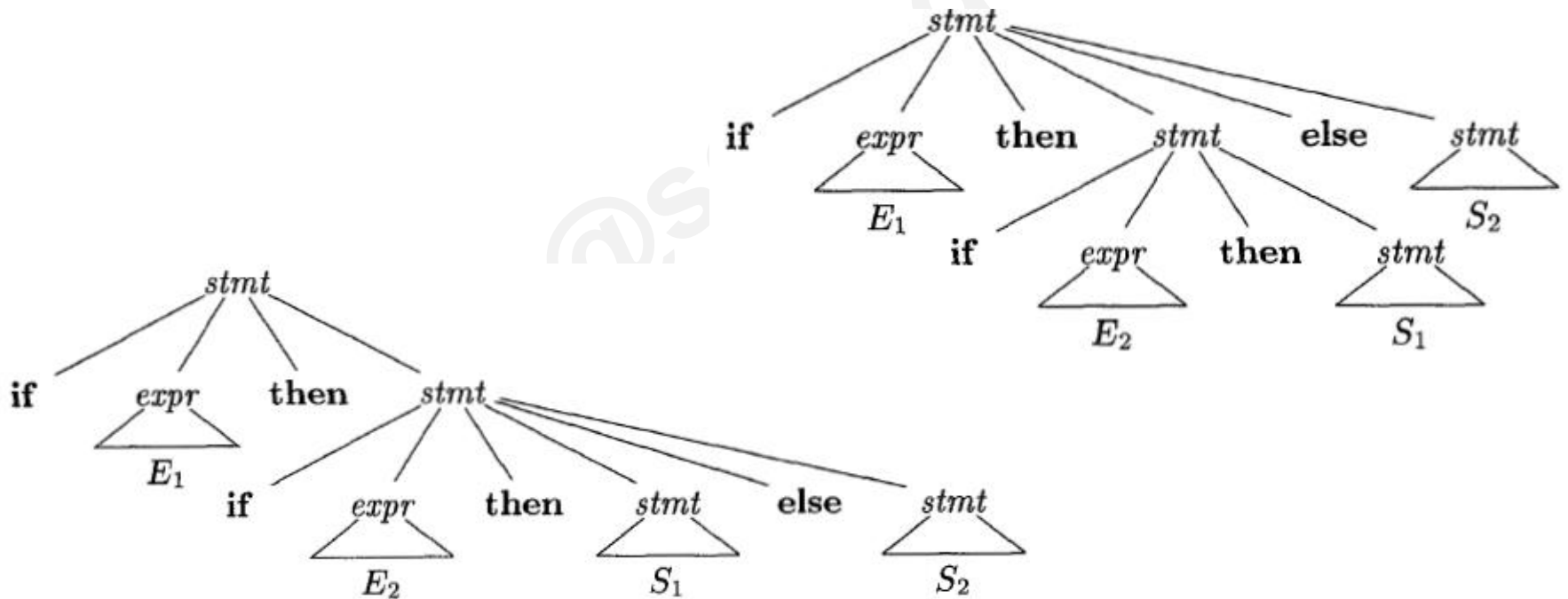
Manually

Consider $stmt \rightarrow$

- $if\ expr\ then\ stmt$
- $if\ expr\ then\ stmt\ else\ stmt$
- $other$



if E_1 then if E_2 then S_1 else S_2





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

Match each else with the
closest unmatched then.

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

4.1.2 Eliminating ε

A context- grammar without ε -production should satisfy the conditions as following

- If there is the production $S \rightarrow \varepsilon$, S should not appear in right-side of any production, where S is the Start Symbol of the grammar;
- There are no other ε -productions .

The algorithm to construct a context-free grammar without ε -production: $G=(V_N, V_T, P, S) \longrightarrow G'=(V'_N, V'_T, P', S')$

(1) Find out all non-terminal symbols that can derive ε after some steps, and put them into the set V_0 ;

(2) Construct the P' set of productions of G' as following steps:

(A) If an symbol in V_0 appears in the right-side of a production, change the production into two productions: substitute the symbol in ε and itself in the production respectively; put the new productions into P'

(B) Otherwise, put the productions relating to the symbol into P' except for ε -production relating to the symbol

(C) If there exists the production of the form $S \rightarrow \varepsilon$ in P , change the production into $S' \rightarrow \varepsilon \mid S$ and put them into P' , let S' be the Start Symbol of G' , let $V'_N = V_N \cup \{S'\}$,



Example: Let $G1 = (\{S\}, \{a, b\}, P, S)$, where

P : (0) $S \rightarrow \varepsilon$ (1) $S \rightarrow aSbS$ (2) $S \rightarrow bSaS$

(1) $V_0 = \{S\}$

(2) P' (1) $S \rightarrow abS \mid aSbS \mid aSb \mid ab$

(2) $S \rightarrow baS \mid bSaS \mid bSa \mid ba$

(0) $S' \rightarrow \varepsilon \mid S$

So: $G1' = (\{S', S\}, \{a, b\}, P', S')$, where

P' : (0) $S' \rightarrow \varepsilon \mid S$

(1) $S \rightarrow abS \mid aSbS \mid aSb \mid ab$

(2) $S \rightarrow baS \mid bSaS \mid bSa \mid ba$

4.1.3 Eliminating Left Recursion

Basic form of **left recursion**

Left recursion is the grammar contains the following kind of productions.

$P \rightarrow P\alpha | \beta$ Immediate recursion

or

$P \rightarrow Aa, A \rightarrow Pb$ Indirect recursion



Elimination of **immediate left recursion**

$$P \rightarrow P\alpha | \beta$$

$$\Rightarrow P \rightarrow \beta \alpha^*$$

$$\Rightarrow P \rightarrow \beta P' \quad P' \rightarrow \alpha P' | \varepsilon$$

Algorithm : Eliminating left recursion.

INPUT: Grammar G with no cycles or ε -productions.

OUTPUT: An equivalent grammar with no left recursion.

METHOD:

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

Example. (1) $S \rightarrow Qc|c$ (2) $Q \rightarrow Rb|b$ (3) $R \rightarrow Sa|a$

Answer: 1) Arrange the non-terminals in the order: R, Q, S

2) for R: no actions.

for Q: $Q \rightarrow Rb|b \longrightarrow Q \rightarrow Sab|ab|b$

for S: $S \rightarrow Qc|c \longrightarrow S \rightarrow Sabc|abc|bc|c;$

then get $S \rightarrow (abc|bc|c)S'$

$S' \rightarrow abcS' | \varepsilon$

3) Because R, Q is not reachable, so delete them
so, the grammar is :

$S \rightarrow (abc|bc|c)S'$

$S' \rightarrow abcS' | \varepsilon$



4.1.4 Eliminating Common Left Factors

If the grammar contains the productions like

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

Chang them into $A \rightarrow \delta A'$

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



Algorithm : Left factoring a grammar.

INPUT: Grammar G.

OUTPUT: An equivalent left-factored grammar.

METHOD: For each nonterminal A, find the longest prefix δ *common to two or more* of its alternatives. If $\delta \neq \epsilon$ - replace all of the A-productions

$$A \rightarrow \delta\beta_1 \mid \delta\beta_2 \mid \dots \mid \delta\beta_n \mid \gamma,$$

Where γ represents all alternatives that do not begin with α , by

$$\begin{aligned} A &\rightarrow \delta A' \mid \gamma \\ A' &\rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n \end{aligned}$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

Example.

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

$$S \rightarrow i E t S S' \mid a$$
$$S' \rightarrow e S \mid \varepsilon$$
$$E \rightarrow b$$



4.1.5 Non-context-free Grammar

There are some tasks in real compilers where the grammar used in the language is beyond context-freeness.

E.g., $L = \{wcw \mid w \text{ is in } (alb)^*\}$



Assignments

Exercise 4.3.1

Seu_zzz@seu.edu.cn



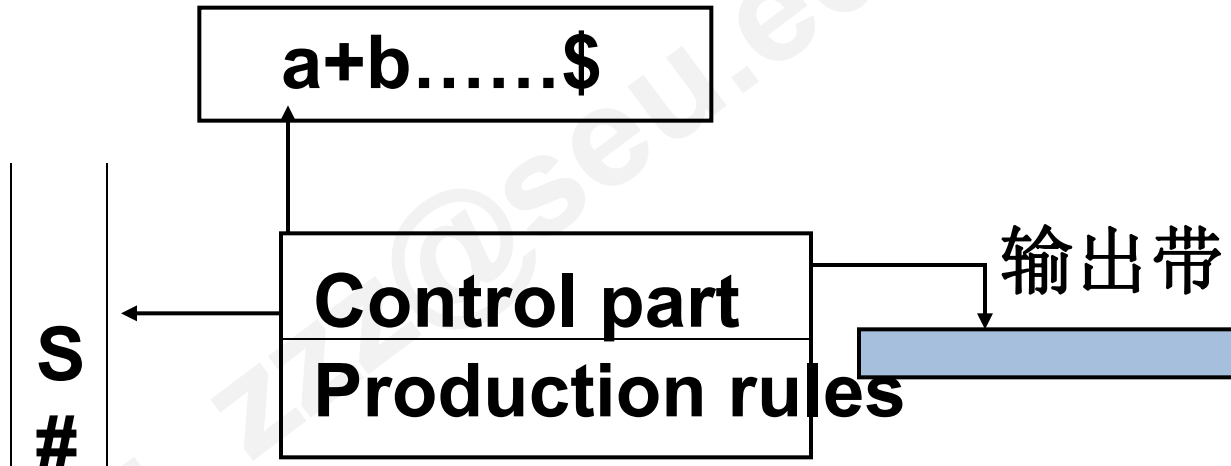
4.2 Recursive-Descent Parsing

```
void A() {  
1)  Choose an A-production,  $A \rightarrow X_1X_2 \dots X_k$ ;  
2)  for ( i = 1 to k ) {  
3)      if (  $X_i$  is a nonterminal )  
4)          call procedure  $X_i()$  ;  
5)      else if (  $X_i$  equals the current input symbol a )  
6)          advance the input to the next symbol;  
7)      else /* an error has occurred */;  
    }  
}
```

Basic Top-Down Parsing Framework



Push Down Automata





Runing Recursive-descent parsing

- (1). if “x” is the top symbol of the stack and is non-terminal, then find a production rule as “ $x \rightarrow \dots$ ” randomly, replace “x” with the right of the rule, and output the No of the rule——**derivation**。
- (2). if “x” is the top symbol of the stack and is same to that under the reading point, then... ——**Matching**。
- (3). if (2) fail, then make a **backtracking action to the scene** before the last derivation and select a new rule ——**backtracking**
- (4). If there no new rule, **fail**
- (5). if there is only “\$” in the stack, and “\$” is under the reading point , **success**

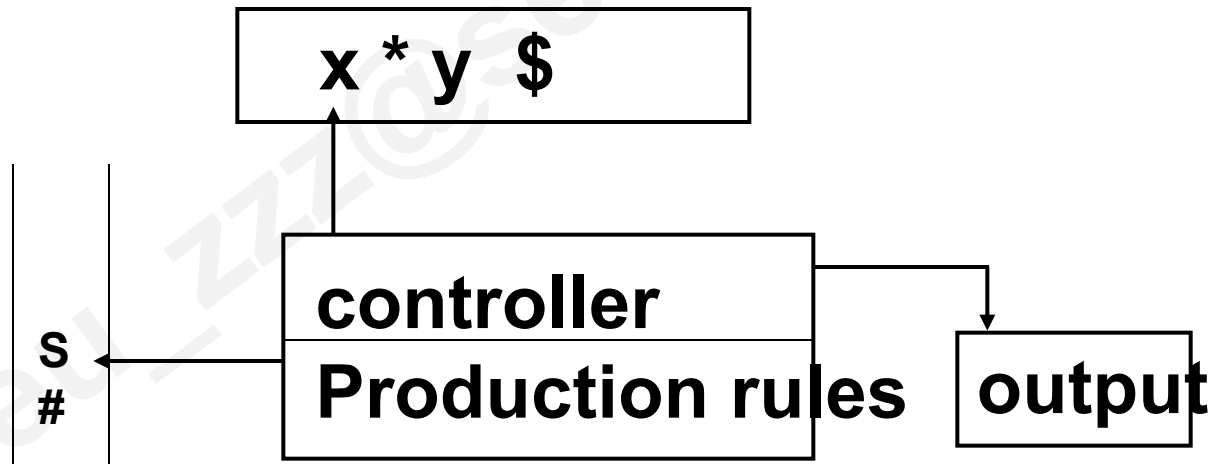


Example.

1) $S \rightarrow xAy$

2) $A \rightarrow **$

3) $A \rightarrow *$

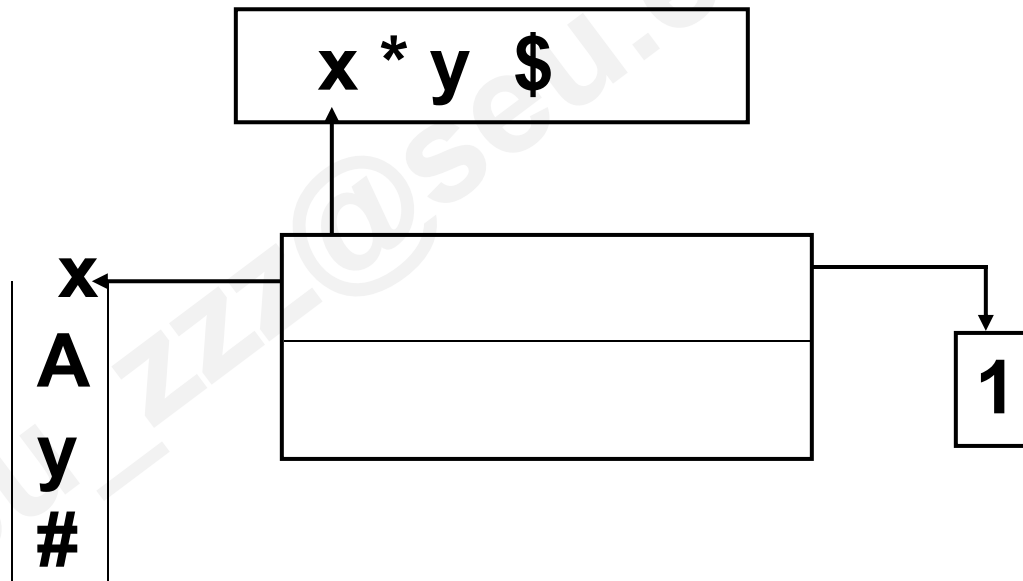




1) $S \rightarrow xAy$

2) $A \rightarrow **$

3) $A \rightarrow *$

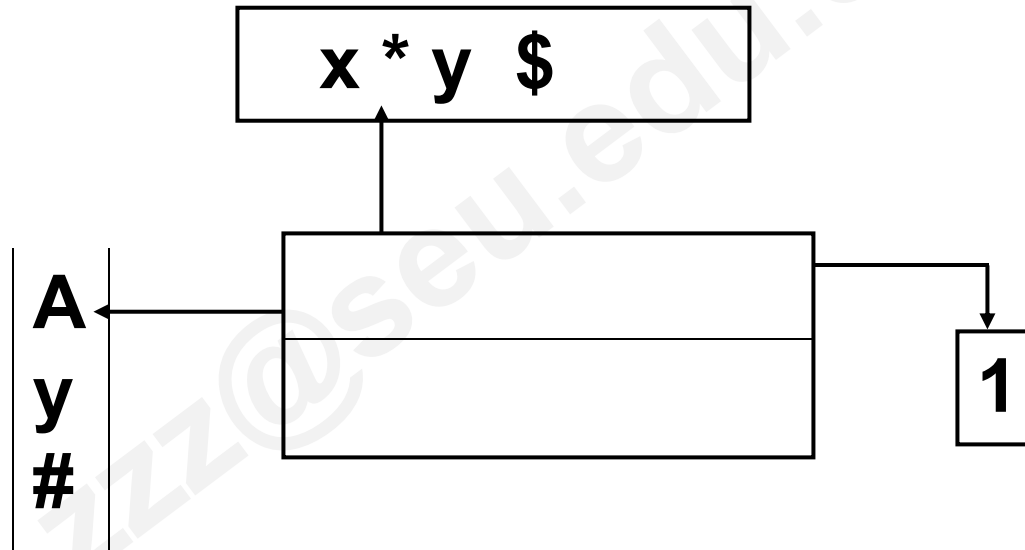




1) $S \rightarrow xAy$

2) $A \rightarrow **$

3) $A \rightarrow *$

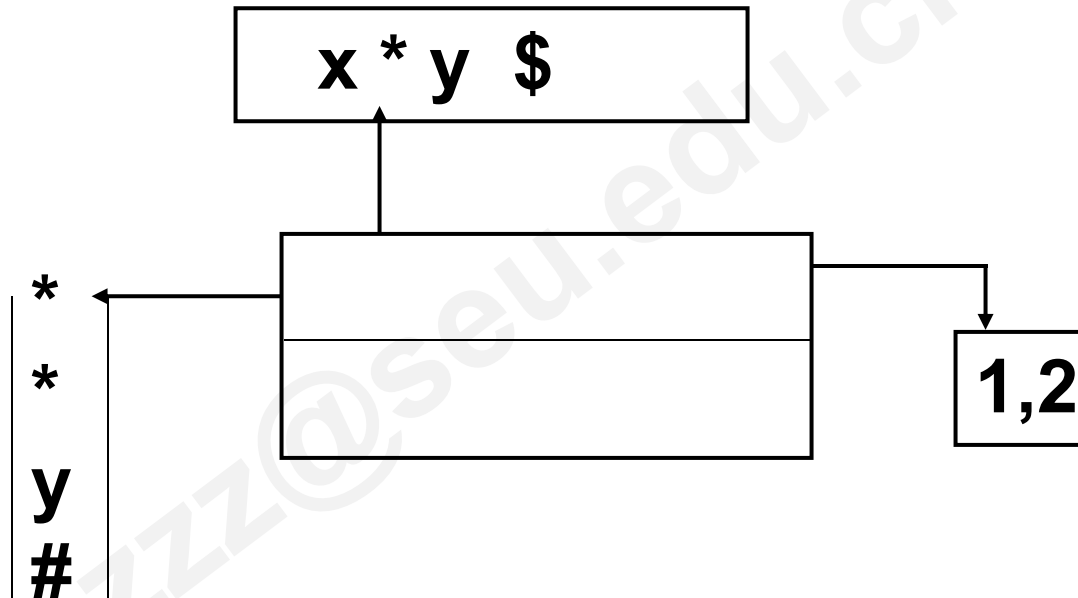




1) $S \rightarrow xAy$

2) $A \rightarrow **$

3) $A \rightarrow *$

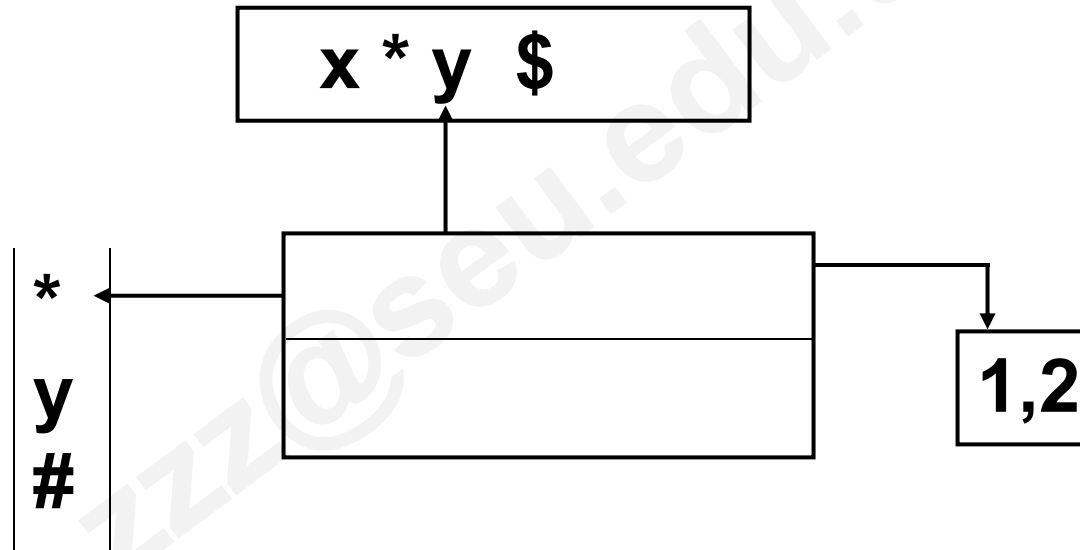




1) $S \rightarrow xAy$

2) $A \rightarrow **$

3) $A \rightarrow *$

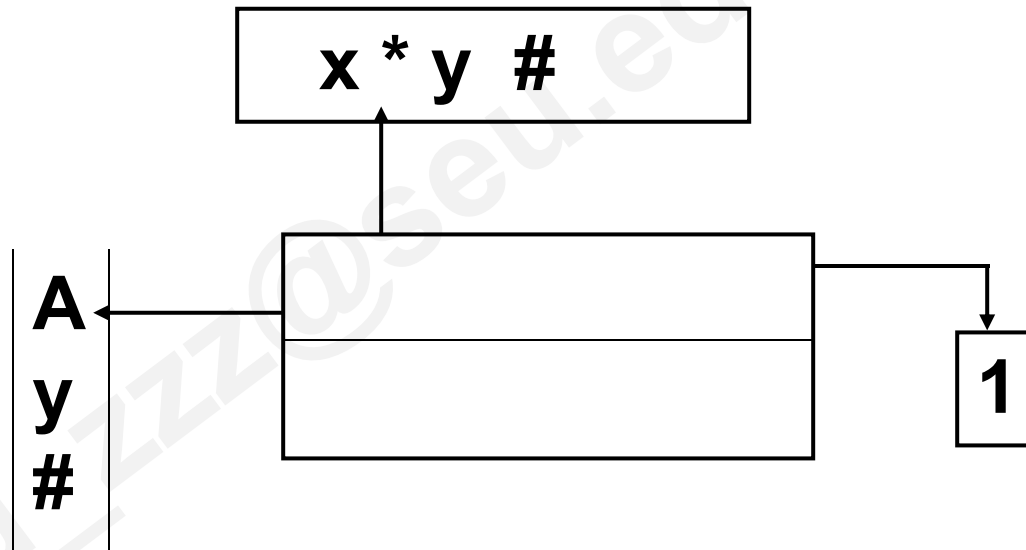




1) $S \rightarrow xAy$

2) $A \rightarrow **$

3) $A \rightarrow *$

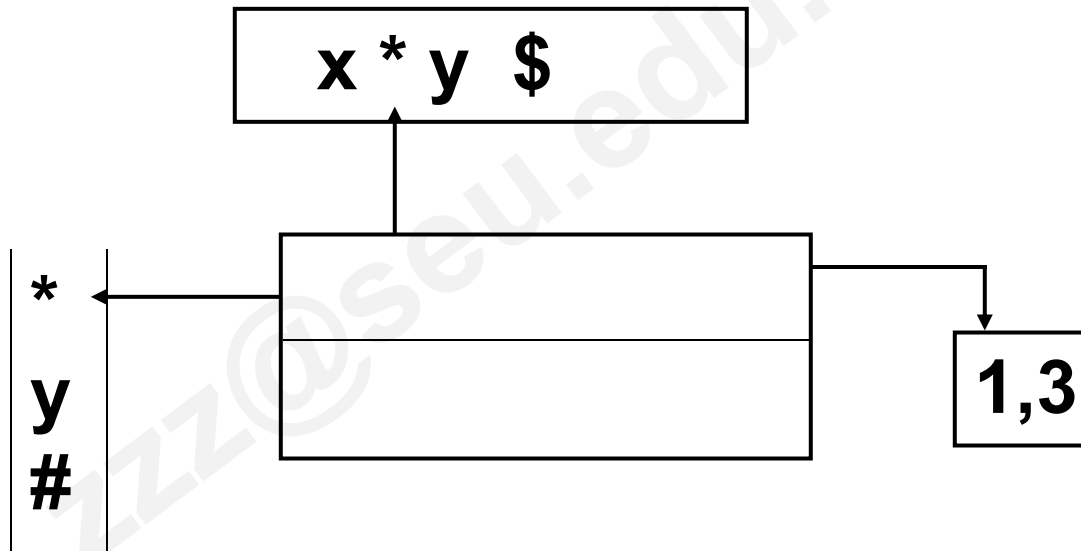




1) $S \rightarrow xAy$

2) $A \rightarrow **$

3) $A \rightarrow *$

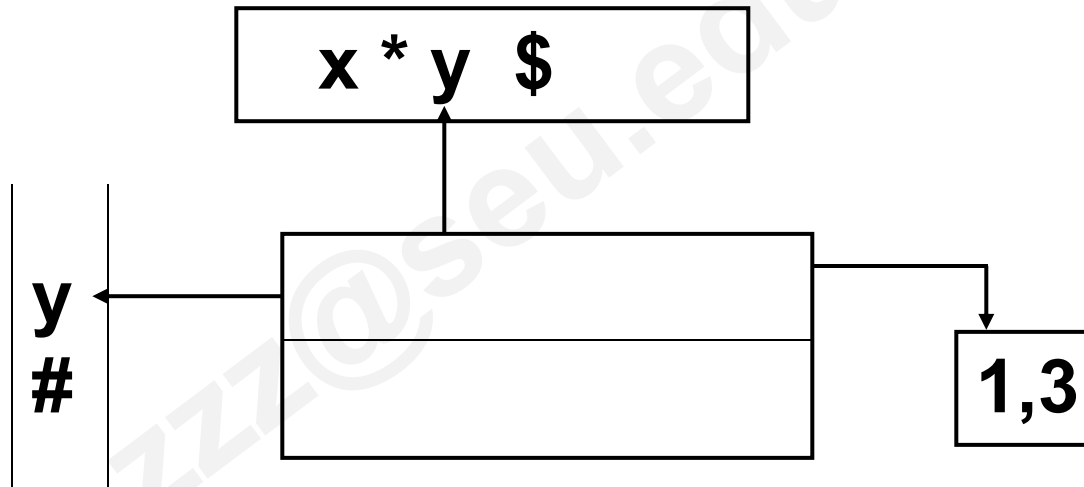




1) $S \rightarrow xAy$

2) $A \rightarrow **$

3) $A \rightarrow *$

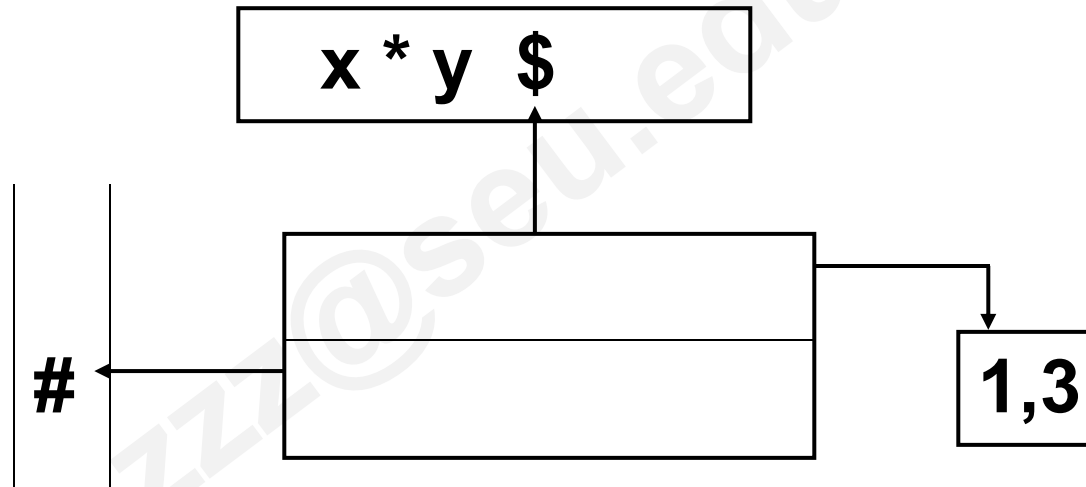




1) $S \rightarrow xAy$

2) $A \rightarrow **$

3) $A \rightarrow *$



4.3 LL(1)

Left to right scanning and leftmost derivation with one step prediction.



Example. Consider the following grammar, and parse the string $\text{id+id*id\$}$

$$1. E \rightarrow TE'$$

$$2. E' \rightarrow +TE'$$

$$3. E' \rightarrow \varepsilon$$

$$4. T \rightarrow FT'$$

$$5. T' \rightarrow *FT'$$

$$6. T' \rightarrow \varepsilon$$

$$7. F \rightarrow \text{id}$$

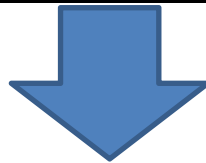
$$8. F \rightarrow (E)$$

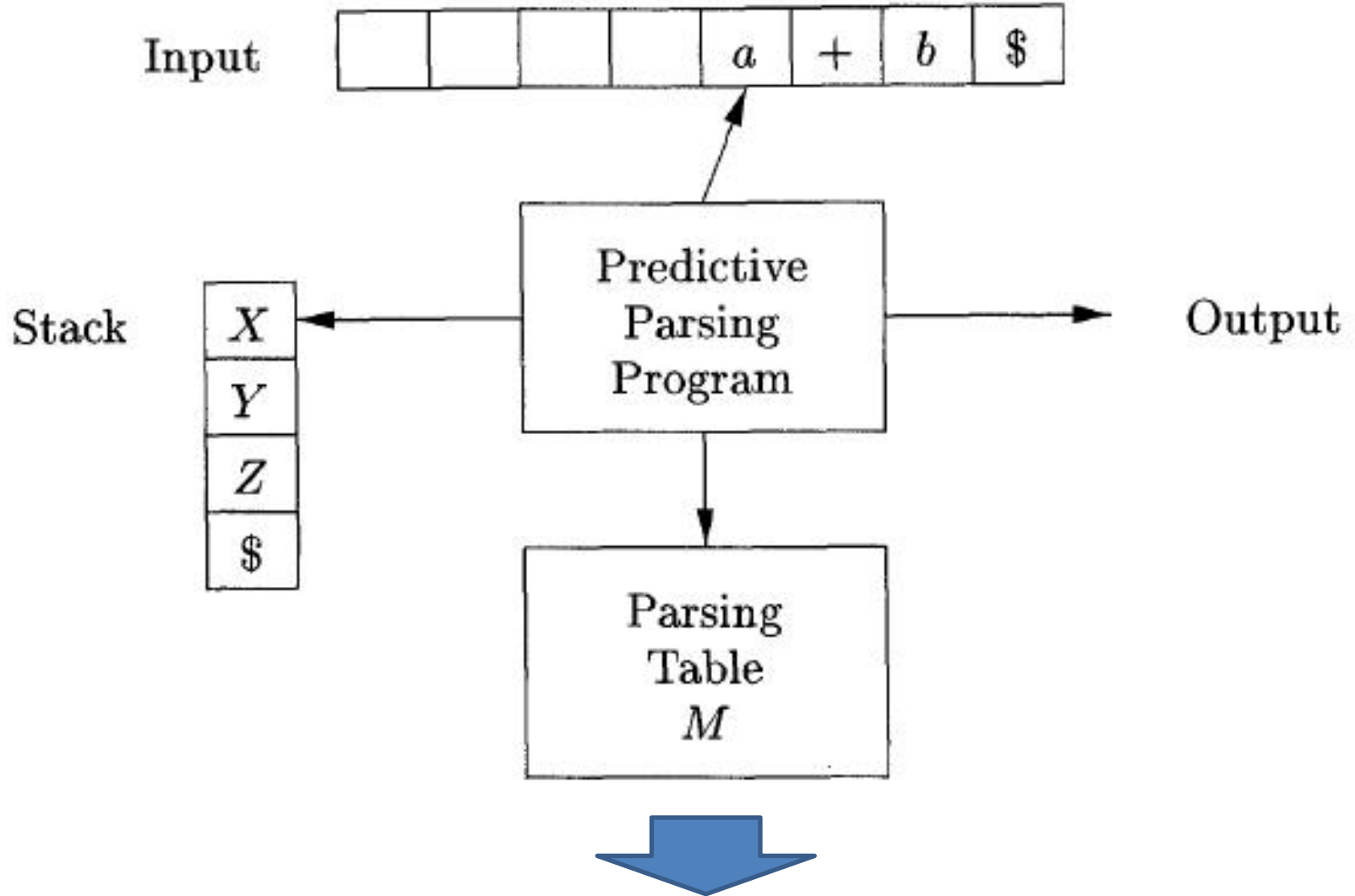




Parsing table M

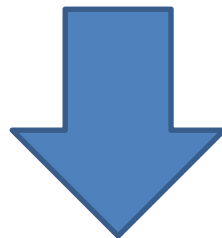
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow i$			$F \rightarrow (E)$		







```
set  $ip$  to point to the first symbol of  $w$ ;  
set  $X$  to the top stack symbol;  
while (  $X \neq \$$  ) { /* stack is not empty */  
    if (  $X$  is  $a$  ) pop the stack and advance  $ip$ ;  
    else if (  $X$  is a terminal )  $error()$ ;  
    else if (  $M[X, a]$  is an error entry )  $error()$ ;  
    else if (  $M[X, a] = X \rightarrow Y_1 Y_2 \cdots Y_k$  ) {  
        output the production  $X \rightarrow Y_1 Y_2 \cdots Y_k$ ;  
        pop the stack;  
        push  $Y_k, Y_{k-1}, \dots, Y_1$  onto the stack, with  $Y_1$  on top;  
    }  
    set  $X$  to the top stack symbol;  
}
```





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

That is great! If we have an available prediction table of a grammar. But

- Does each grammar has a prediction table such that an entry is deterministic?**
- How to construct the prediction table for a given grammar?**

4.3.1 Prediction Table

FIRST & FOLLOW

FIRST:

- If α is any string of grammar symbols, let $FIRST(\alpha)$ be the set of terminals that begin the string derived from α .
- If $\alpha \xrightarrow{+} \varepsilon$, then ε is also in $FIRST(\alpha)$
- That is :

$$\alpha \in V^*, First(\alpha) = \{a \mid \alpha \xrightarrow{+} a \dots, a \in V_T\}$$



FOLLOW:

- For non-terminal A , to be the set of terminals a that can appear immediately to the right of A in some sentential form.
- That is: $\text{Follow}(A) = \{a \mid S \xrightarrow{+} \dots Aa \dots, a \in V_T\}$
If $S \rightarrow \dots A$, then $\$ \in \text{FOLLOW}(A)$.



Computing FIRST(α)

1. If X is a terminal, then $\text{FIRST}(X) = \{X\}$.
2. If X is a nonterminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in $\text{FIRST}(X)$ if for some i , a is in $\text{FIRST}(Y_i)$, and ϵ is in all of $\text{FIRST}(Y_1), \dots, \text{FIRST}(Y_{i-1})$; that is, $Y_1 \cdots Y_{i-1} \xRightarrow{*} \epsilon$. If ϵ is in $\text{FIRST}(Y_j)$ for all $j = 1, 2, \dots, k$, then add ϵ to $\text{FIRST}(X)$. For example, everything in $\text{FIRST}(Y_1)$ is surely in $\text{FIRST}(X)$. If Y_1 does not derive ϵ , then we add nothing more to $\text{FIRST}(X)$, but if $Y_1 \xRightarrow{*} \epsilon$, then we add $\text{FIRST}(Y_2)$, and so on.
3. If $X \rightarrow \epsilon$ is a production, then add ϵ to $\text{FIRST}(X)$.



Computing Follow(α)

- (1) Place \$ in FOLLOW(S), where S is the start symbol and # is the input right end-marker.
- (2) If there is $A \rightarrow \alpha B \beta$ in G, then add **First(β)- $\{\epsilon\}$** to Follow(B).
- (3) If there is $A \rightarrow \alpha B$, or $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ , then add Follow(A) to Follow(B).



Example. Construct FIRST & FOLLOW for each non-terminals

$$1. E \rightarrow TE'$$

$$2. E' \rightarrow +TE'$$

$$3. E' \rightarrow \varepsilon$$

$$4. T \rightarrow FT'$$

$$5. T' \rightarrow *FT'$$

$$6. T' \rightarrow \varepsilon$$

$$7. F \rightarrow i$$

$$8. F \rightarrow (E)$$



Answer:

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

$\text{First}(E') = \{ +, \varepsilon \}$

$\text{First}(T') = \{ *, \varepsilon \}$

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

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

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



Construction of the prediction table

- Input. Grammar G .
- Output. Parsing table M .

Method.

1. For each production $A \rightarrow \alpha$, do steps 2 and 3.
2. For each terminal a in $\text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$.
3. If ε is in $\text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal b in $\text{FOLLOW}(A)$. If ε is in $\text{FIRST}(\alpha)$ and $\#$ is in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, \#]$.
4. Make each undefined entry of M be error.



	i	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow i$			$F \rightarrow (E)$		

A grammar whose parsing table has no multiply-defined entries is said to be **LL(1) grammar**.

(1) No ambiguous can be LL(1).

(2) **Left-recursive** grammar cannot be LL(1).

(3) A grammar G is LL(1) if and only if whenever $A \rightarrow \alpha$ | β are two distinct productions of G :

1). For no terminal a do both α and β derive strings beginning with a .

2). At most one of α and β can derive the empty string.

3). If $\beta \rightarrow \varepsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A).

4.3.2 Example

-----LL(1) Grammar

Please eliminate the left recursion and extract maximum common left factors (if there are) from the following context free grammar, and then decide the resulted grammar is whether a LL(1) grammar by constructing the related LL(1) parsing table. If it is, then verify “*i (b) t i b t a e a*” using the table.

$$S \rightarrow iCtS | iCtSeS | a$$
$$C \rightarrow C \text{ or } D | D$$
$$D \rightarrow D \text{ or } E | E$$
$$E \rightarrow (C) | b$$



4.3.3 Example

-----non LL(1) Grammar

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \varepsilon$

$E \rightarrow b$



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



NOTE: If a grammar is not LL(1), then it is still a recursive predictive parsing.

4.4 Error Recovery in LL(1)

Panic Mode: Skipping symbols on the the input until a token in a selected set of synchronizing tokens appears.

Phrase Level Recovery: Filling in the blank entries in the predictive parsing table with pointers to error routines.



5. Bottom-up Parsing

id * id

F * id
|
id

T * id
|
 F
|
id

T * F
| |
 F id
|
id

T
/ | \
 T * F
| |
 F id
|
id

E
|
 T
/ | \
 T * F
| |
 F id
|
id



5.1 Shift-Reduce Parsing

STACK	INPUT	ACTION
\$	id₁ * id₂ \$	shift
\$ id₁	* id₂ \$	reduce by $F \rightarrow \text{id}$
\$ F	* id₂ \$	reduce by $T \rightarrow F$
\$ T	* id₂ \$	shift
\$ T *	id₂ \$	shift
\$ T * id₂	\$	reduce by $F \rightarrow \text{id}$
\$ T * F	\$	reduce by $T \rightarrow T * F$
\$ T	\$	reduce by $E \rightarrow T$
\$ E	\$	accept



Shift/Reduce conflict: the parser cannot decide whether to shift or to reduce.

STACK	INPUT	ACTION
\$	id ₁ * id ₂ \$	shift
\$ id ₁	* id ₂ \$	reduce by $F \rightarrow \text{id}$
\$ F	* id ₂ \$	reduce by $T \rightarrow F$
\$ T	* id ₂ \$	shift
\$ T *	id ₂ \$	shift
\$ T * id ₂	\$	reduce by $F \rightarrow \text{id}$
\$ T * F	\$	reduce by $T \rightarrow T * F$
\$ T	\$	reduce by $E \rightarrow T$
\$ E	\$	accept



Reduce/Reduce conflict: the parser cannot decide whether to shift or to reduce.

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



5.2 LR Parsing

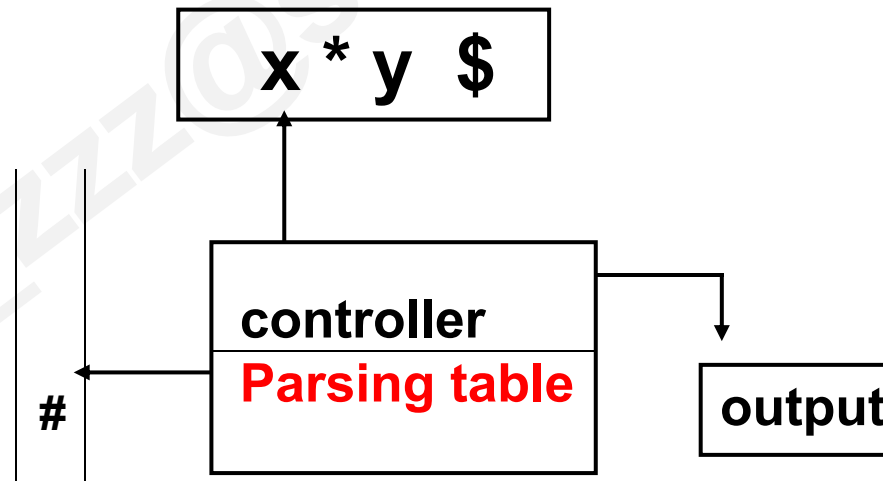
LR(k) parsing is the most prevalent type of bottom-up parsing.

- “L” is for left-to-right scanning.
- “R” is for constructing a rightmost derivation in reverse.
- “k” is for the number of input symbols of lookahead to eliminate conflicts.



A **LR(k) parsing table** is used to determine what actions among reduce and shift is available by

- I. The top symbol in the stack, and
- II. k numbers of symbols from the position where the reading pointer is on.
- III. The state of the controller.





Example. A parsing table of the following grammar

$$1. E \rightarrow E + T$$

$$2. E \rightarrow T$$

$$3. T \rightarrow T * F$$

$$4. T \rightarrow F$$

$$5. F \rightarrow (E)$$

$$6. F \rightarrow \text{id}$$





STATE	ACTION						GOTO		
	id	+	*	()	\$	<i>E</i>	<i>T</i>	<i>F</i>
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			





	STACK	SYMBOLS	INPUT	ACTION
(1)	0		id * id + id \$	shift
(2)	0 5	id	* id + id \$	reduce by $F \rightarrow \text{id}$
(3)	0 3	F	* id + id \$	reduce by $T \rightarrow F$
(4)	0 2	T	* id + id \$	shift
(5)	0 2 7	$T *$	id + id \$	shift
(6)	0 2 7 5	$T * \text{id}$	+ id \$	reduce by $F \rightarrow \text{id}$
(7)	0 2 7 10	$T * F$	+ id \$	reduce by $T \rightarrow T * F$
(8)	0 2	T	+ id \$	reduce by $E \rightarrow T$
(9)	0 1	E	+ id \$	shift
(10)	0 1 6	$E +$	id \$	shift
(11)	0 1 6 5	$E + \text{id}$	\$	reduce by $F \rightarrow \text{id}$
(12)	0 1 6 3	$E + F$	\$	reduce by $T \rightarrow F$
(13)	0 1 6 9	$E + T$	\$	reduce by $E \rightarrow E + T$
(14)	0 1	E	\$	accept



Algorithm LR-parsing algorithm.

Input An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G .

Output If w is in $L(G)$, the reduction steps of a bottom-up parse for w ; otherwise, an error indication.

Method Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the program
□

```

let  $a$  be the first symbol of  $w\$$ ;
while(1) { /* repeat forever */
    let  $s$  be the state on top of the stack;
    if ( ACTION[ $s, a$ ] = shift  $t$  ) {
        push  $t$  onto the stack;
        let  $a$  be the next input symbol;
    } else if ( ACTION[ $s, a$ ] = reduce  $A \rightarrow \beta$  ) {
        pop  $|\beta|$  symbols off the stack;
        let state  $t$  now be on top of the stack;
        push GOTO[ $t, A$ ] onto the stack;
        output the production  $A \rightarrow \beta$ ;
    } else if ( ACTION[ $s, a$ ] = accept ) break; /* parsing is done */
    else call error-recovery routine;
}

```




How to construct a conflict-free parsing table?

- ① SLR
- ② LR(1)
- ③ LALR
- ④ Using Ambiguities



5.2.1 Simple LR

LR(0) Item:

Seu_zzz@seu.edu.cn

Algorithm : Constructing an SLR-parsing table.

INPUT: An augmented grammar G .

OUTPUT: The SLR-parsing table functions ACTION and GOTO for G .

METHOD:

1. Construct $C = \{I_0, I_1, \dots, I_n\}$, the collection of sets of LR(0) items for G .

2. State i is constructed from I_i . The parsing actions for state i are determined as follows:

(a) If $[A \rightarrow \alpha \bullet a \beta]$ is in I_i , and $\text{GOTO}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to "shift j ". Here a must be a terminal.

(b) If $[A \rightarrow \alpha \bullet]$ is in I_i then set $\text{ACTION}[i, a]$ to "reduce $A \rightarrow \alpha$ " for all a in $\text{FOLLOW}(A)$; here A may not be S' .

(c) If $[S' \rightarrow S \bullet]$ is in I_i , then set $\text{ACTION}[i, \$]$ to "accept."

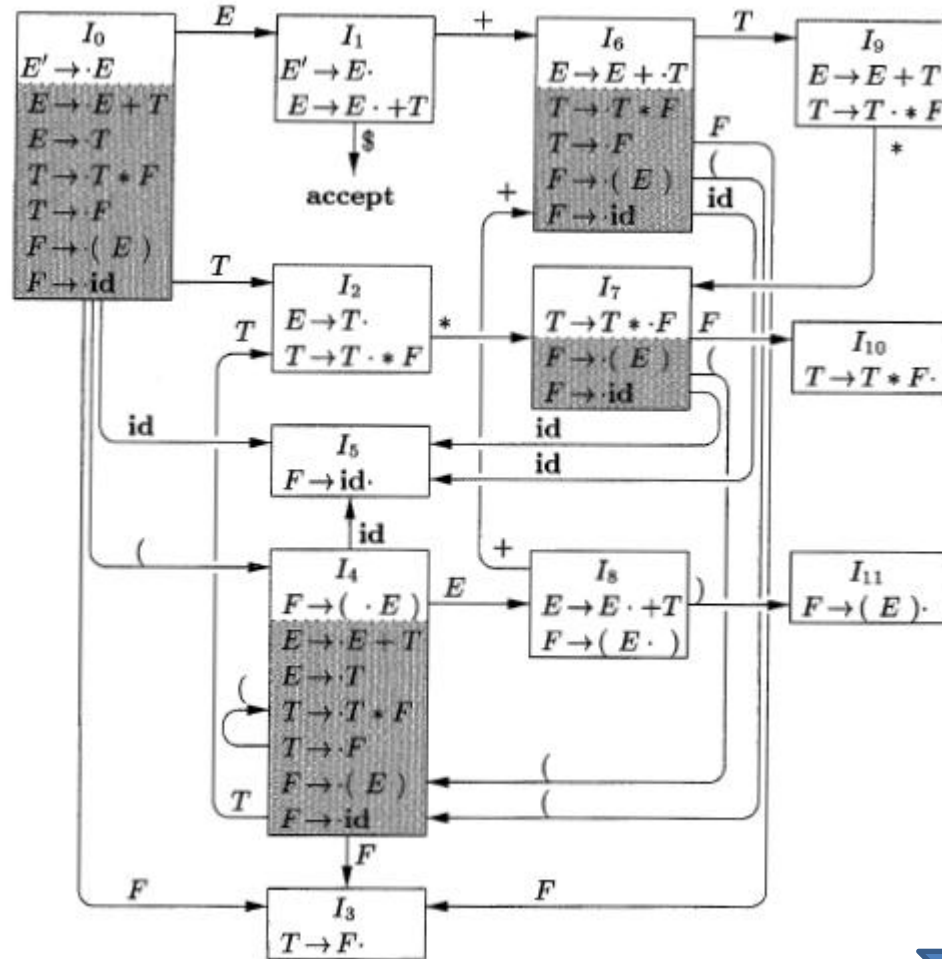
If any conflicting actions result from the above rules, we say the grammar is not SLR(1). The algorithm fails to produce a parser in this case.

3. The goto transitions for state i are constructed for all nonterminals A using the rule: If $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}[i, A] = j$.
4. All entries not defined by rules (2) and (3) are made "error."
5. The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow \bullet S]$.



Example.

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





STATE	ACTION						GOTO		
	id	+	*	()	\$	<i>E</i>	<i>T</i>	<i>F</i>
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Procedure of Using SLR

Step 1. Eliminating Left Recursions

Step 2. Life Maximal Common Leftmost Factors

Step 3. Construct SLR parsing table.

Step 4. If no errors in the parsing table then the grammar is a SLR(1) grammar, then goto 5, otherwise, break.

Example. The following grammar is not SLR(1).

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

ACTION[2, =] ?

$$\begin{aligned} I_0: \quad &S' \rightarrow \cdot S \\ &S \rightarrow \cdot L = R \\ &S \rightarrow \cdot R \\ &L \rightarrow \cdot * R \\ &L \rightarrow \cdot \text{id} \\ &R \rightarrow \cdot L \end{aligned}$$

$$I_5: \quad L \rightarrow \text{id} \cdot$$

$$\begin{aligned} I_6: \quad &S \rightarrow L = \cdot R \\ &R \rightarrow \cdot L \\ &L \rightarrow \cdot * R \\ &L \rightarrow \cdot \text{id} \end{aligned}$$

$$I_1: \quad S' \rightarrow S \cdot$$

$$I_7: \quad L \rightarrow *R \cdot$$

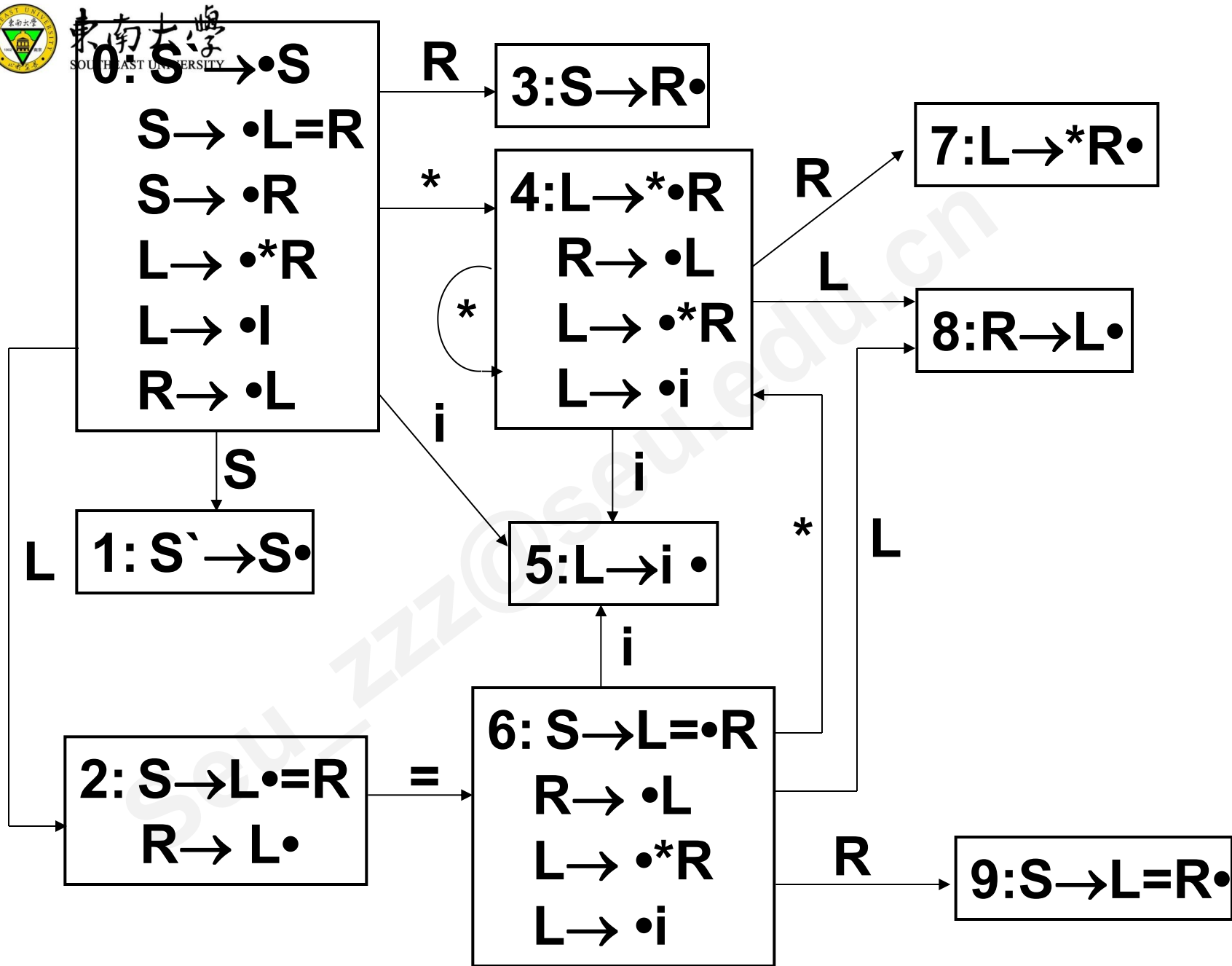
$$\begin{aligned} I_2: \quad &S \rightarrow L \cdot = R \\ &R \rightarrow L \cdot \end{aligned}$$

$$I_8: \quad R \rightarrow L \cdot$$

$$I_9: \quad S \rightarrow L = R \cdot$$

$$I_3: \quad S \rightarrow R \cdot$$

$$\begin{aligned} I_4: \quad &L \rightarrow * \cdot R \\ &R \rightarrow \cdot L \\ &L \rightarrow \cdot * R \\ &L \rightarrow \cdot \text{id} \end{aligned}$$



state	ACTION				GOTO		
	=	i	*	#	S	L	R
0		S ₅	S ₄		1	2	3
1				acc			
2	S ₆ / r ₆			r ₆			
3				r ₃			
4		S ₅	S ₄			8	7
5	r ₅			r ₅			
6		S ₅	S ₄			8	9
7	r ₄			r ₄			
8	r ₆			r ₆			
9				r ₂			

5.2.1 LR(1)

Construction of the sets of LR(1) items

Input. An augmented grammar G'

Output. The sets of LR(1) items that are the set of items valid for one or more viable prefixes of G' .

Method. The procedures closure and goto and the main routine items for constructing the sets of items.

```

function closure(I);
{ do { for (each item  $(A \rightarrow \alpha \bullet B \beta, a)$  in I,
           each production  $B \rightarrow \gamma$  in  $G'$ ,
           and each terminal b in  $\text{FIRST}(\beta a)$ 
           such that  $(B \rightarrow \bullet \gamma, b)$  is not in I )
      add  $(B \rightarrow \bullet \gamma, b)$  to I;
    } while there is still new items add to I;
  return I
}

```

```
function goto(I, X);  
{ let J be the set of items  $(A \rightarrow \alpha X \bullet \beta, a)$  such  
  that  $(A \rightarrow \alpha \bullet X \beta, a)$  is in I ;  
  
  return closure(J)  
}
```

Void items (G');

$\{C = \{\text{closure}(\{ (S' \rightarrow \bullet S, \#) \})\};$

do { for (each set of items I in C and each
grammar symbol X

such that

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

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

} while there is still new items add to C ;

}

Construction of the canonical LR parsing table

- **Input.** An augmented grammar G'
- **Output.** The canonical LR parsing table functions action and goto for G'
- **Method.**
 - (1) Construct $C=\{I_0, I_1, \dots, I_n\}$, the collection of sets of **LR(1)** items for G' .
 - (2) State i is constructed from I_i . The parsing **actions** for state i are determined as follows:

- a) If $[A \rightarrow \alpha \bullet a \beta, b]$ is in I_i and $\text{goto}(I_i, a) = I_j$, then set $\text{ACTION}[i, a] = \text{"Shift } j\text{"}$, here a must be a terminal.
- b) If $[A \rightarrow \alpha \bullet, a] \in I_i$, $A \neq S'$, then set $\text{ACTION}[i, a] = r_j$; j is the No. of production $A \rightarrow \alpha$.
- c) If $[S' \rightarrow \bullet S, \#]$ is in I_i , then set $\text{ACTION}[i, \#]$ to "accept"

- (3) The goto transitions for state i are determined as follows: if $\text{goto}(I_i, A) = I_j$, then $\text{goto}[i, A] = j$.
- (4) All entries not defined by rules 2 and 3 are made “error”
- (5) The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow \bullet S, \#]$.
- If any conflicting actions are generated by the above rules, we say the grammar is not LR(1).



Example. Compute the items for the following grammar:

$$1. S' \rightarrow S \quad 2. S \rightarrow CC$$

$$3. C \rightarrow cC|d$$

Answer: the initial set of items is I_0 :

I_0
$S' \rightarrow \bullet S, \#$
$S \rightarrow \bullet CC, \#$
$C \rightarrow \bullet cC, c d$
$C \rightarrow \bullet d, c d$



10: $S' \rightarrow \bullet S, \#$

16: $C \rightarrow c \bullet C, \#$

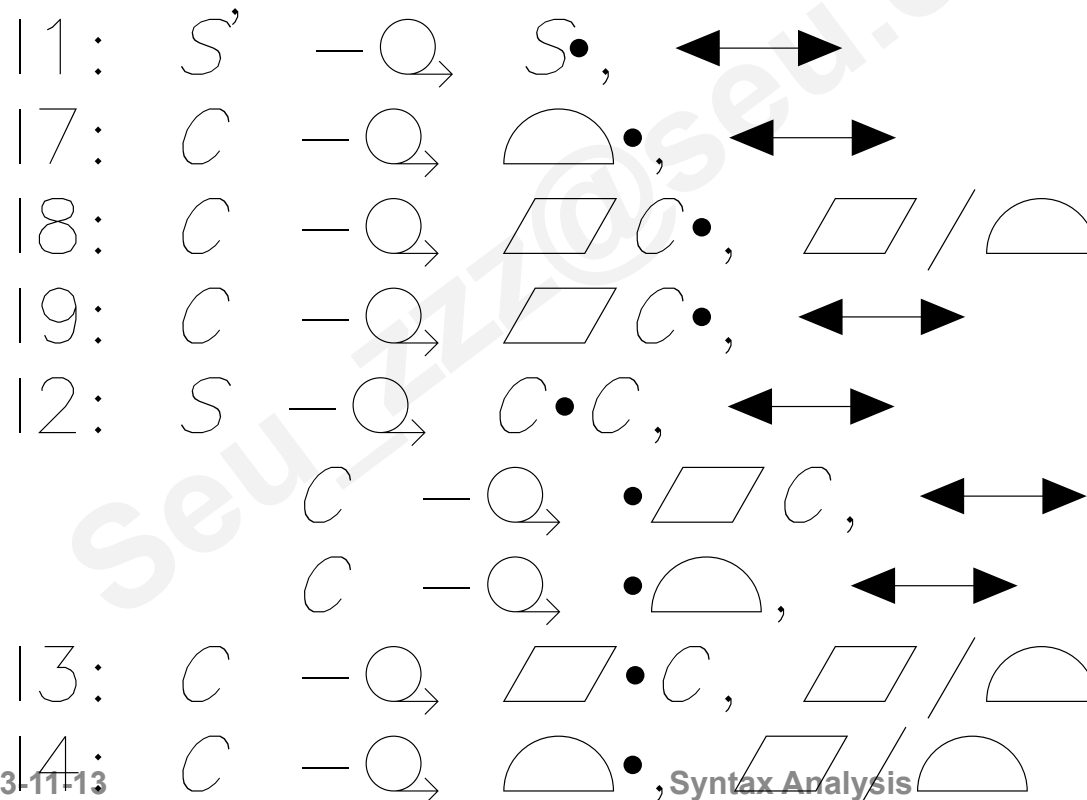
$S \rightarrow \bullet CC, \#$

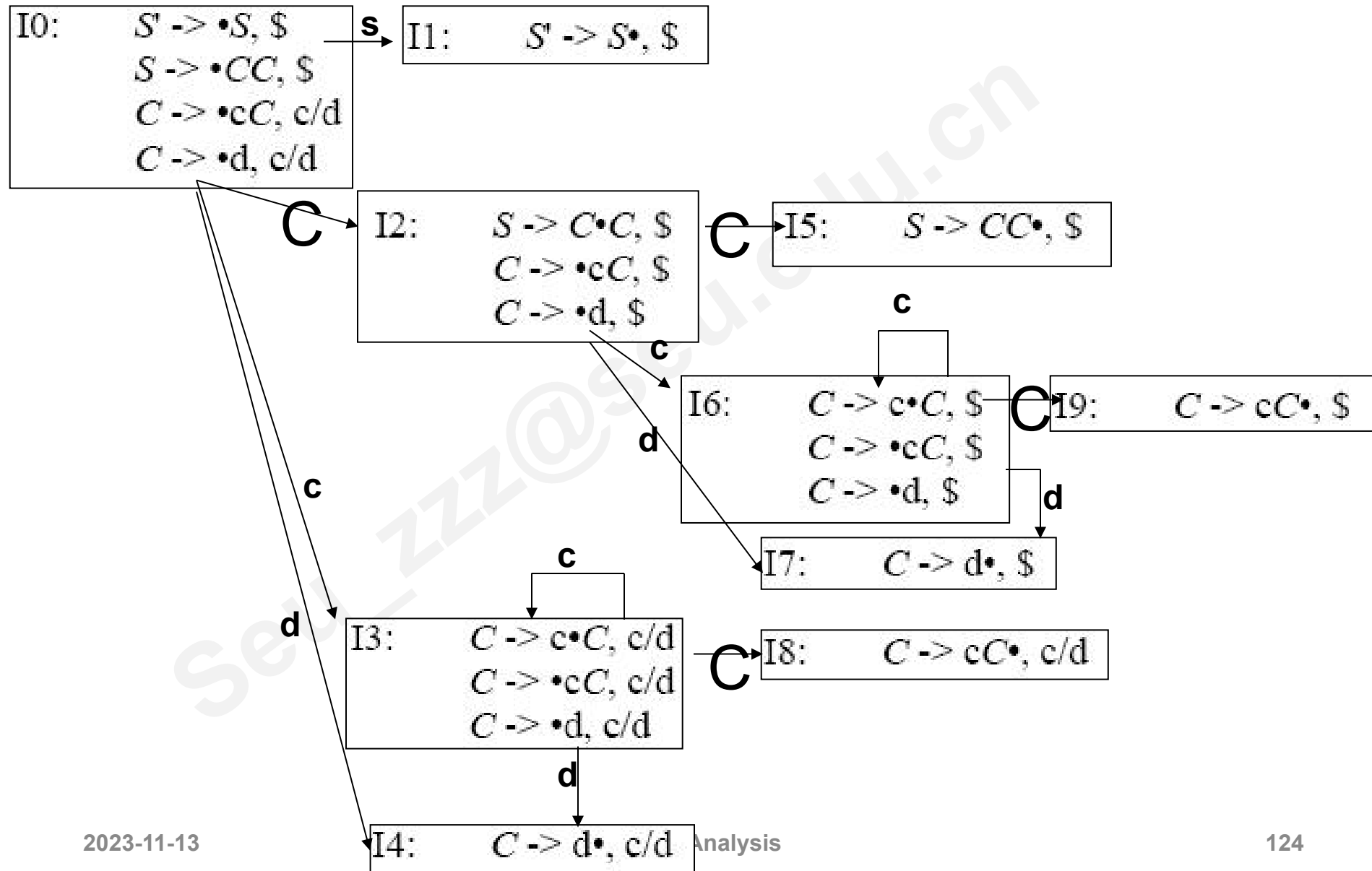
$C \rightarrow \bullet cC, \#$

$C \rightarrow \bullet cC, c/d$

$C \rightarrow \bullet d, \#$

$C \rightarrow \bullet d, c/d$







state	Action			goto	
	c	d	#	S	C
0	S ₃	S ₄		1	2
1			acc		
2	S ₆	S ₇			5
3	S ₃	S ₄			8
4	r ₃	r ₃			
5			r ₁		
6	S ₆	S ₇			9
7			r ₃		
8	r ₂	r ₂			
9			r ₂		



5.2.1 LALR

Basic idea

Merge the set of LR(1) items having the same core

- (1) When merging, the GOTO sub-table can be merged without any conflict, because GOTO function **just relies on the core**
- (2) When merging, the ACTION sub-table can also be merged without any conflicts, but it may occur the case of merging of error and shift/reduce actions. We assume non-error actions
- (3) After the set of LR(1) items are merged, an error may be caught lately, but the error will eventually be caught, in fact, it will be caught before any more input symbols are shifted.

(4)After merging, the conflict of reduce/reduce may be occurred.

2) The sets of LR(1) items having the same core

- The states which have the same items but the look-ahead symbols are different, then the states are having the same core.

Notes: We may merge these sets with common cores into one set of items.

18、 An easy, but space-consuming LALR table construction

- Input. An augmented grammar G'
- Output. The LALR parsing table functions action and goto for G'
- Method.
 - (1) Construct $C=\{I_0, I_1, \dots, I_n\}$, the collection of sets of LR(1) items.
 - (2) For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by their union.

- (3) Let $C' = \{J_0, J_1, \dots, J_m\}$ be the resulting sets of LR(1) items. The parsing actions for state I are constructed from J_i . If there is a parsing action conflict, the algorithm fails to produce a parser, and the grammar is not a LALR.
- (4) The goto table is constructed as follows.

- If J is the union of one or more sets of LR(1) items, that is, $J = I_1 \cup I_2 \cup \dots \cup I_k$ then the cores of $\text{goto}(I_1, X)$, $\text{goto}(I_2, X), \dots, \text{goto}(I_k, X)$ are the same, since I_1, I_2, \dots, I_n all have the same core. Let K be the union of all sets of items having the same core as $\text{goto}(I_1, X)$. then $\text{goto}(J, X) = K$.

If there is no parsing action conflicts , the given grammar is said to be an LALR(1) grammar



I0: $S' \rightarrow \bullet S, \$$
 $S \rightarrow \bullet CC, \$$
 $C \rightarrow \bullet cC, c/d$
 $C \rightarrow \bullet d, c/d$

I1: $S' \rightarrow S\bullet, \$$

I2: $S \rightarrow C\bullet C, \$$
 $C \rightarrow \bullet cC, \$$
 $C \rightarrow \bullet d, \$$

I3: $C \rightarrow c\bullet C, c/d$
 $C \rightarrow \bullet cC, c/d$
 $C \rightarrow \bullet d, c/d$

I4: $C \rightarrow d\bullet, c/d$

I5: $S \rightarrow CC\bullet, \$$

I6: $C \rightarrow c\bullet C, \$$
 $C \rightarrow \bullet cC, \$$
 $C \rightarrow \bullet d, \$$

I7: $C \rightarrow d\bullet, \$$

I8: $C \rightarrow cC\bullet, c/d$

I9: $C \rightarrow cC\bullet, \$$

state	Action			goto	
	c	d	#	S	C
0	S ₃	S ₄		1	2
1			acc		
2	S ₆	S ₇			5
3	S ₃	S ₄			8
4	r ₃	r ₃			
5			r ₁		
6	S ₆	S ₇			9
7			r ₃		
8	r ₂	r ₂			
9			r ₂		

State	action			goto	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Parsing string ccd

5.3 Using Ambiguities

Example 1. Using Precedence and Associativity to Resolve Parsing Action Conflicts

Grammar: $E \rightarrow E + E \mid E * E \mid (E) \mid i$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid i$

$i + i + i * i + i$

With LR idea, according other conditions, analyze ambiguity Grammar。 Steps:

- 1、 Construct LR(0) parsing table;
- 2、 if Conflicts happens, solve them with SLR
;
- 3、 The rest conflicts are solved by other conditions

E.g:

$E' \rightarrow E$

$E \rightarrow E + E \mid E * E \mid (E) \mid \mid$

1) LR(0) Parsing Table

2) SLR

E.G

$I_1: E' \rightarrow E \bullet$

$E \rightarrow E \bullet + E$

$E \rightarrow E \bullet * i$

Re—Shift conflict

3) Other conflicts

E.g: $I_7, E' \rightarrow E + E \bullet$

$E \rightarrow E \bullet + E$

$E \rightarrow E \bullet * E$

Re—Shift conflict



I0: $E' \rightarrow \bullet E$
 $E \rightarrow \bullet E + E$
 $E \rightarrow \bullet E * E$
 $E \rightarrow \bullet (E)$
 $E \rightarrow \bullet \text{id}$

I1: $E' \rightarrow E \bullet$
 $E \rightarrow E \bullet + E$
 $E \rightarrow E \bullet * E$

I2: $E \rightarrow (\bullet E)$
 $E \rightarrow \bullet E + E$
 $E \rightarrow \bullet E * E$
 $E \rightarrow \bullet (E)$
 $E \rightarrow \bullet \text{id}$

I3: $E \rightarrow \text{id} \bullet$

I4: $E \rightarrow E + \bullet E$
 $E \rightarrow \bullet E + E$
 $E \rightarrow \bullet E * E$
 $E \rightarrow \bullet (E)$
 $E \rightarrow \bullet \text{id}$

I5: $E \rightarrow E * \bullet E$
 $E \rightarrow \bullet E + E$
 $E \rightarrow \bullet E * E$
 $E \rightarrow \bullet (E)$
 $E \rightarrow \bullet \text{id}$

I6: $E \rightarrow (E \bullet)$
 $E \rightarrow E \bullet + E$
 $E \rightarrow E \bullet * E$

I7: $E \rightarrow E + E \bullet$
 $E \rightarrow E \bullet + E$
 $E \rightarrow E \bullet * E$

I8: $E \rightarrow E * E \bullet$
 $E \rightarrow E \bullet + E$
 $E \rightarrow E \bullet * E$

I9: $E \rightarrow (E) \bullet$



stat e	ACTION						GOTO
	i	+	*	()	#	S
0	S ₃			S ₂			1
1		S ₄	S ₅			acc	
2	S ₃			S ₂			6
3		r ₄	r ₄		r ₄	r ₄	
4	S ₃			S ₂			7
5	S ₃			S ₂			8
6		S ₄	S ₅		S ₉		
7		r ₁ /S ₄	S ₅ /r ₁		r ₁	r ₁	
8		r ₂ /S ₄	r ₂ /S ₅		r ₂	r ₂	
9		r ₃	r ₃		r ₃	r ₃	



For ACTION[7, *],

➤ **reduction or shift?**

➤ **“Shift” because “*” is superior**

For ACTION[7, +]

➤ **reduction or shift?**

➤ **“Shift” because the left “*” is superior**



状态	ACTION						GOTO
	i	+	*	()	#	S
0	S ₃			S ₂			1
1		S ₄	S ₅			acc	
2	S ₃			S ₂			6
3		r ₄	r ₄		r ₄	r ₄	
4	S ₃			S ₂			7
5	S ₃			S ₂			8
6		S ₄	S ₅		S ₉		
7		r ₁ (S ₄)	S ₅		r ₁	r ₁	
8		r ₂ (S ₄)	r ₂ (S ₅)		r ₂	r ₂	
9		r ₃	r ₃		r ₃	r ₃	

Example 2 The “Dangling-else” Ambiguity

Grammar:

$$S' \rightarrow S$$
$$S \rightarrow \text{if expr then stmt else stmt}$$
$$\quad \quad \quad | \text{if expr then stmt}$$
$$\quad \quad \quad | \text{other}$$
$$S' \rightarrow S$$
$$S \rightarrow iSeS \mid iS \mid a$$



I0: $S' \rightarrow \bullet S$
 $S \rightarrow \bullet iSeS$
 $S \rightarrow \bullet iS$
 $S \rightarrow \bullet a$

I1: $S' \rightarrow S \bullet$

I2: $S \rightarrow i \bullet SeS$
 $S \rightarrow i \bullet S$
 $S \rightarrow \bullet iSeS$
 $S \rightarrow \bullet iS$
 $S \rightarrow \bullet a$

I3: $S \rightarrow a \bullet$

I4: $S \rightarrow iS \bullet eS$
 $S \rightarrow iS \bullet$

I5: $S \rightarrow iSe \bullet S$
 $S \rightarrow \bullet iSeS$
 $S \rightarrow \bullet iS$
 $S \rightarrow \bullet a$

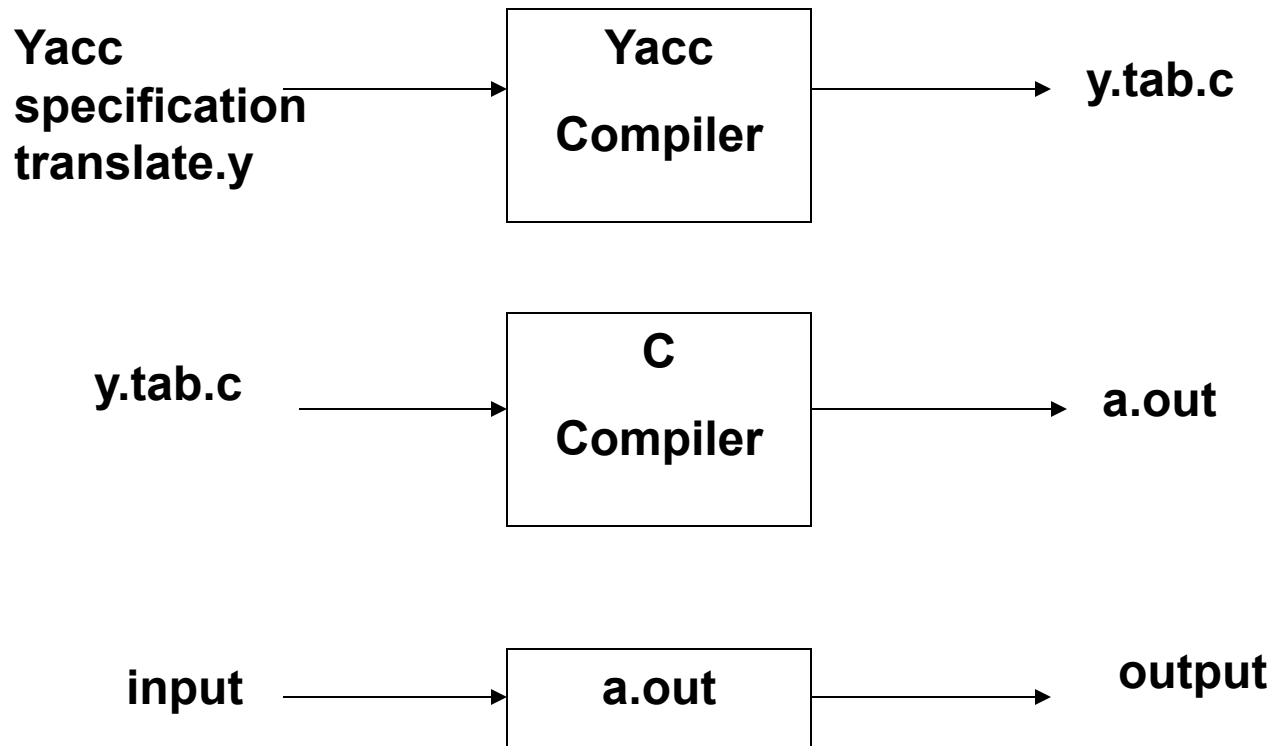
I6: $S \rightarrow iSeS \bullet$

state	ACTION				GOTO
	i	e	a	#	S
0	S ₂		S ₃		1
1				acc	
2	S ₂		S ₃		4
3	r ₄	r ₄	r ₄	r ₄	
4	r ₃	S ₅ /r ₃	r ₃		
5	S ₂		S ₃		6
6	r ₁	r ₁	r ₁		

stat e	ACTION				GOTO
	i	e	a	#	S
0	S ₂		S ₃		1
1				acc	
2	S ₂		S ₃		4
3	r ₄	r ₄	r ₄	r ₄	
4	r ₃	S ₅ /r ₃	r ₃		
5	S ₂		S ₃		6
6	r ₁	r ₁	r ₁		

6. YACC

1、Creating an input/output translator with Yacc



2、 Three parts of a Yacc source program

declaration

%%


translation rules

%%

supporting C-routines

Notes: The form of a translation rule is as followings:

<Left side>: <alt> {semantic action}



```
%token INTEGER
%%
```

```
S : E    {printf("%d\n", $1);};
```

```
E : T    {$$ = $1;}
  | E A T {switch ($2) {
              case '+': $$ = $1 + $3; break;
              case '-': $$ = $1 - $3; break; }
};
```

```
T : F    {$$ = $1;}
  | T M F {switch ($2) {
              case '*': $$ = $1 * $3; break;
              case '/': $$ = $1 / $3; break; }
};
```

```
F : '(' E ')'    {$$ = $2;}
  | INTEGER      {$$ = $1;}
  | '-'INTEGER    {$$ = -$2;}
  | '-'('E')     {$$ = -$3;};
```

```
A : '+'    {$$ = '+';}
  | '-'    {$$ = '-'};
```

```
M : '*'    {$$ = '*'}
  | '/'    {$$ = '/'};
```

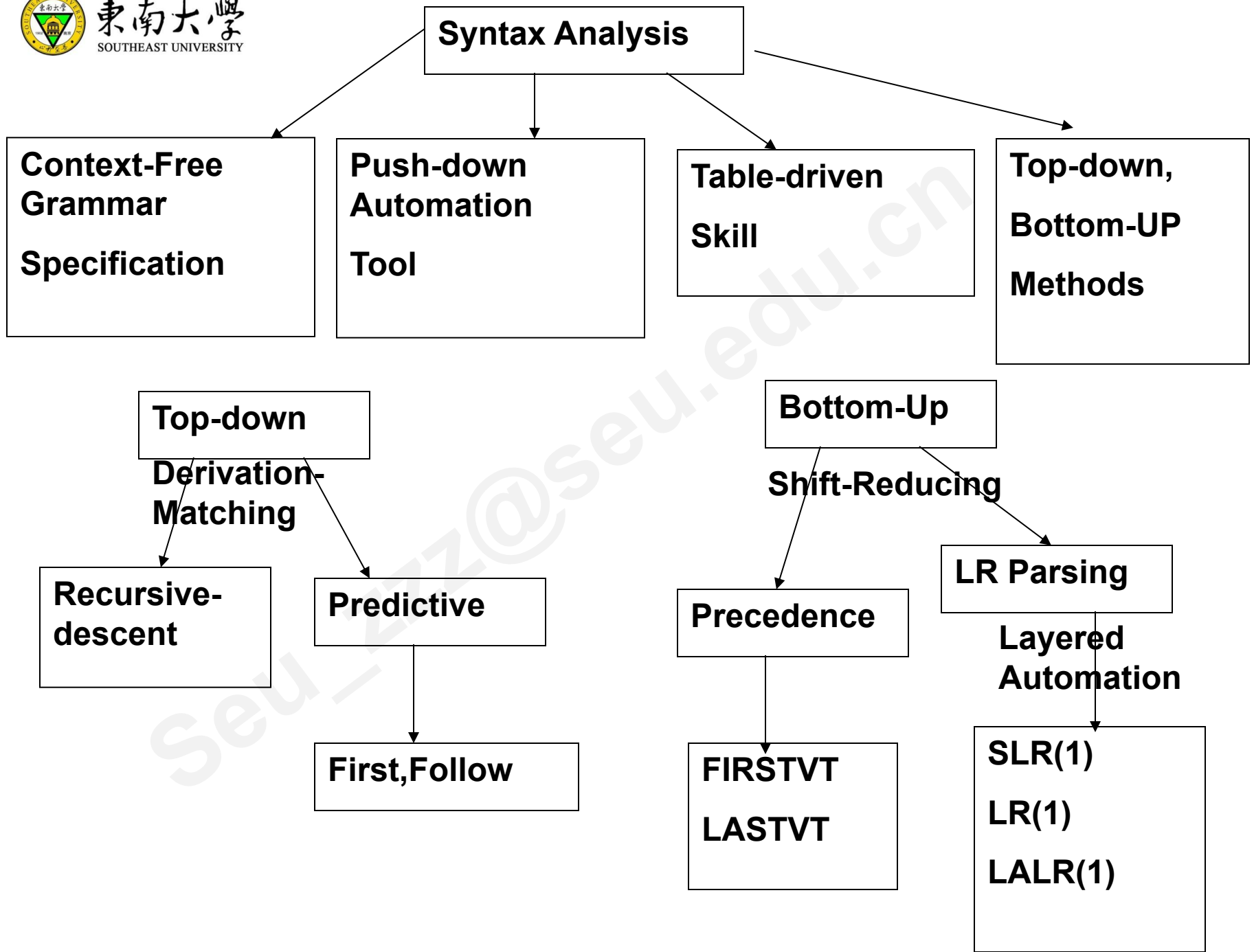
```
%%
```

```
main()
```

```
{
```

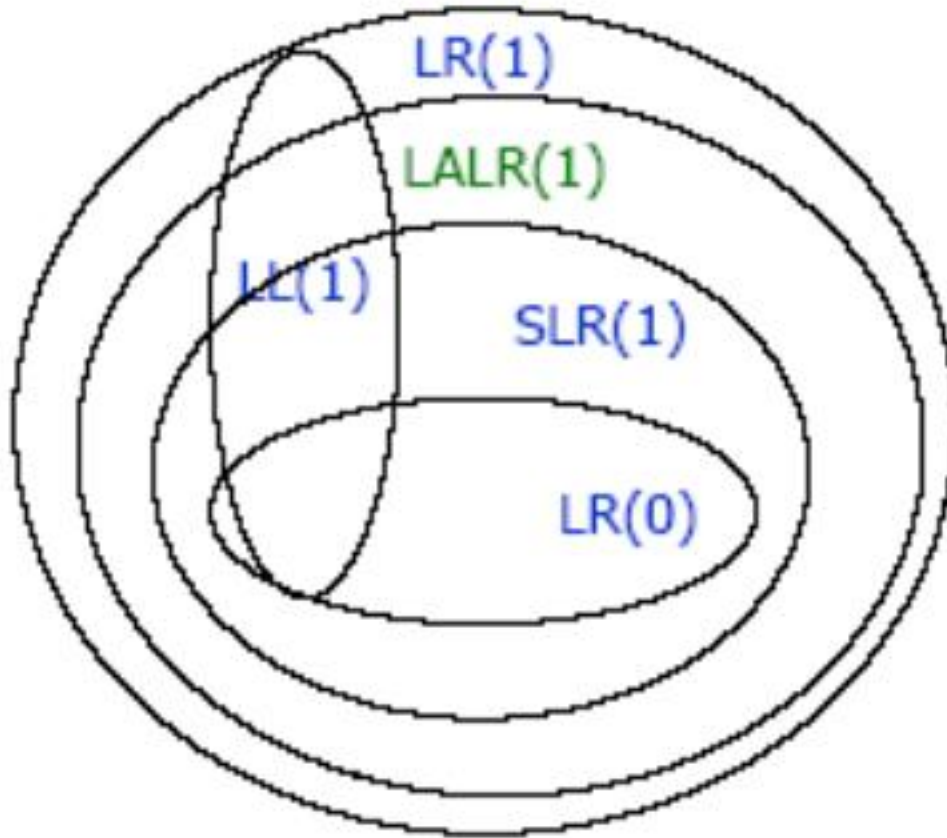
```
    if (!yyparse())
        printf("completed parse\n");
    else
        printf("illegal expression\n");
```

```
}
```





Classification of Grammars



$$LR(k) \subseteq LR(k+1)$$

$$LL(k) \subseteq LL(k+1)$$

$$LL(k) \subseteq LR(k)$$

$$LR(0) \subseteq SLR(1)$$

$$LALR(1) \subseteq LR(1)$$



Recursive Descent Analyses

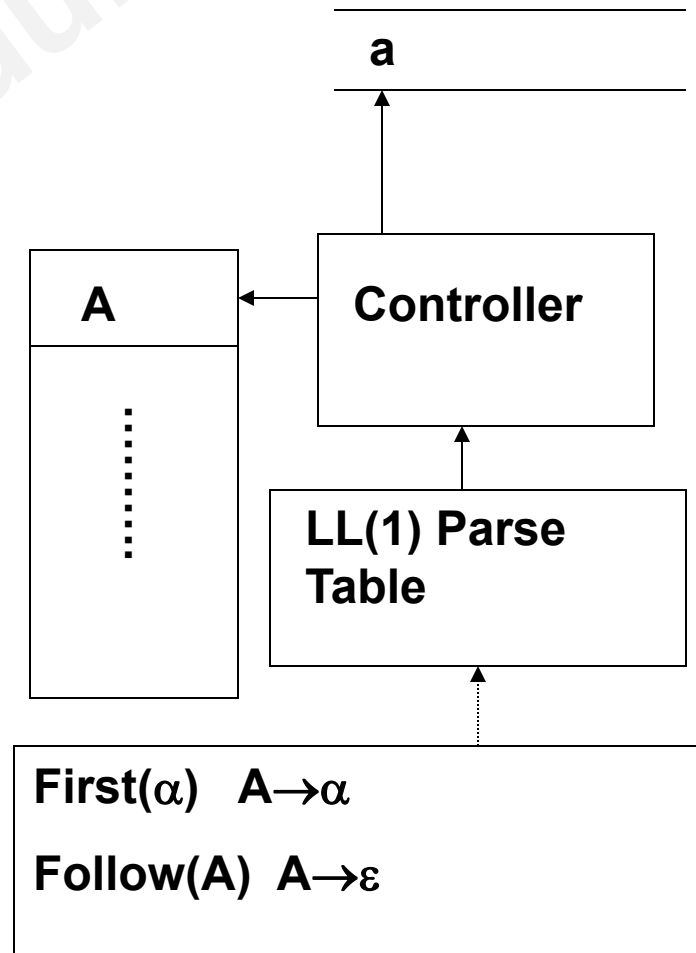
Advantages: Easy to write programs

Disadvantages: Backtracking, poor efficiency

Predictive Analyses : predict the production which is used when a non-terminated occurs on top of the analyses stack

Skills : **First**, **Follow**

Disadvantages: More pre-processes (Elimination of left recursions, Extracting maximum common left factors)





Bottom-up --- Operator Precedence Analyses

Skills : Shift- Reduce , FIRSTVT, LASTVT

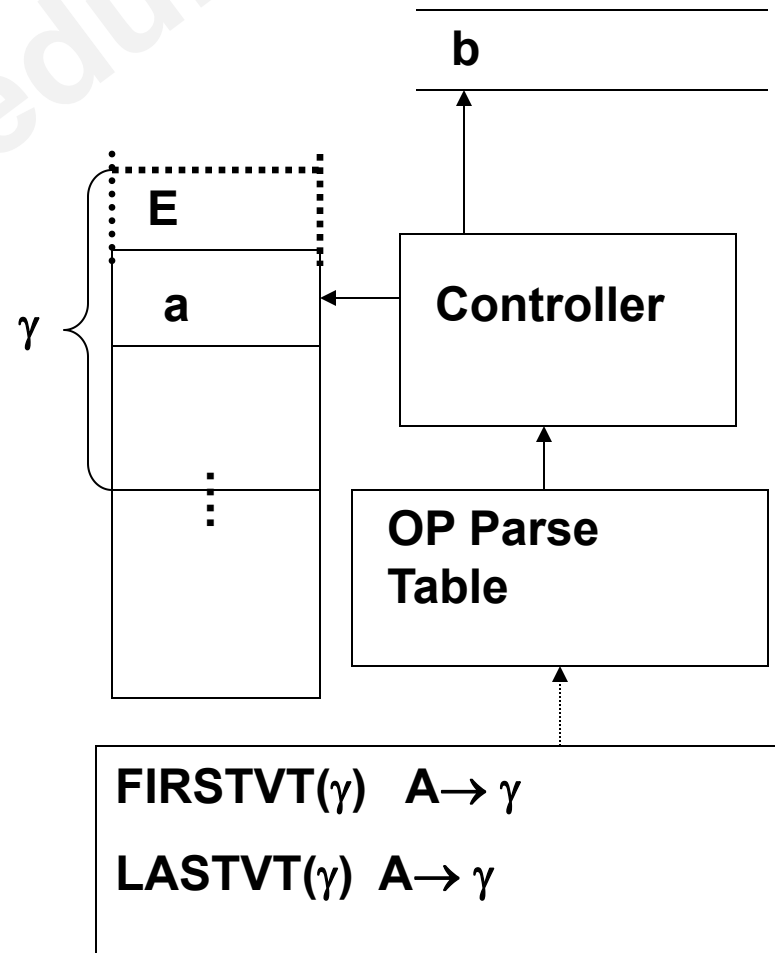
Disadvantages: Strict grammar limitation, poor reduce mechanism

Simple LR Analyses : based on determined FA, state stack and symbol stack (two stacks)

Skills : LR item and Follow(A)

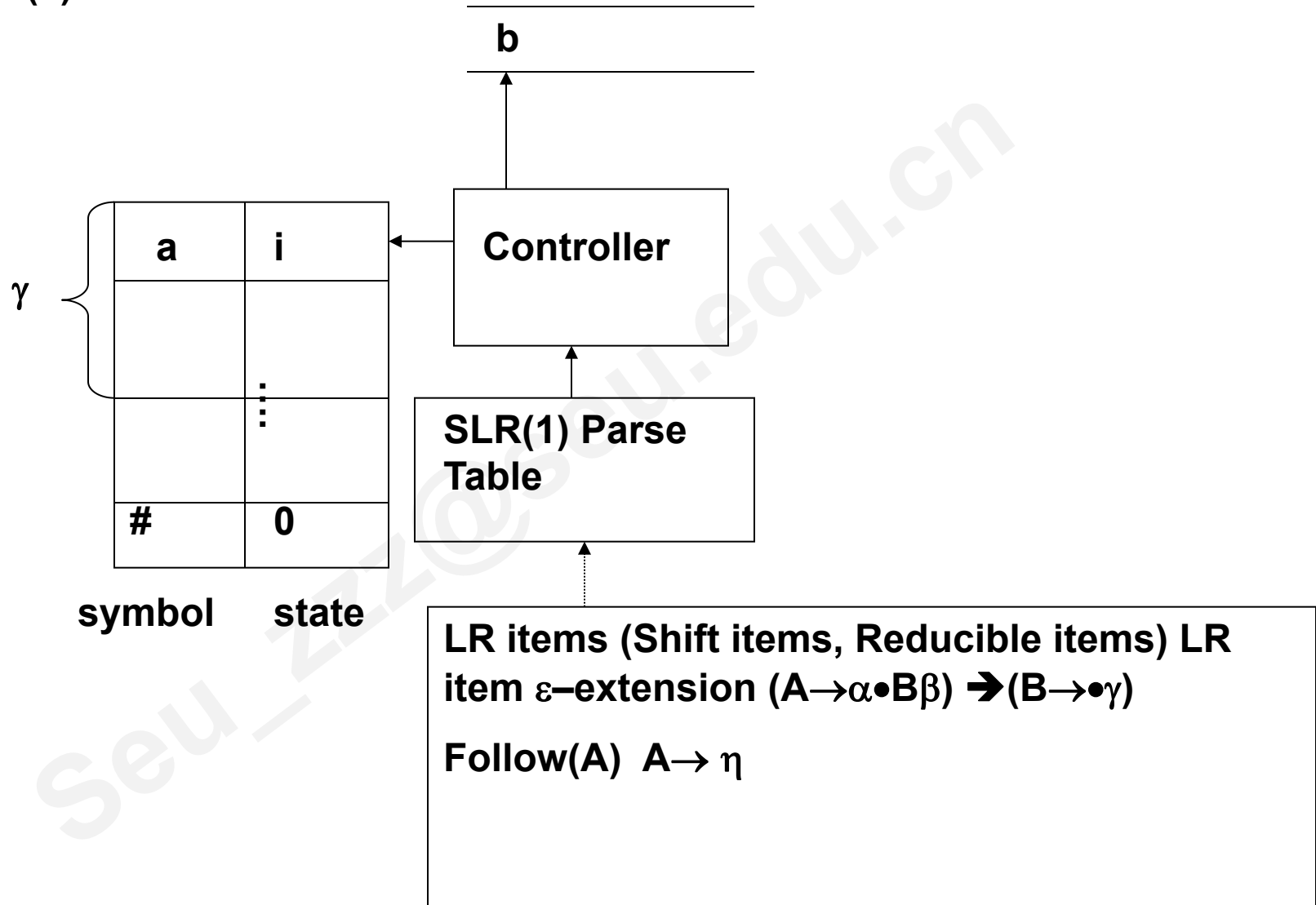
Disadvantages: cannot solve the problems of shift-reduce conflict and reduce-reduce conflict

LR(1) analyses





東南大學 SOUTHEAST UNIVERSITY SLR(1) Parser:





Canonical LR Analyses(LR(1))

SOUTHEAST UNIVERSITY

Skills : LR(1) item and Look-ahead symbol

Disadvantages: more states



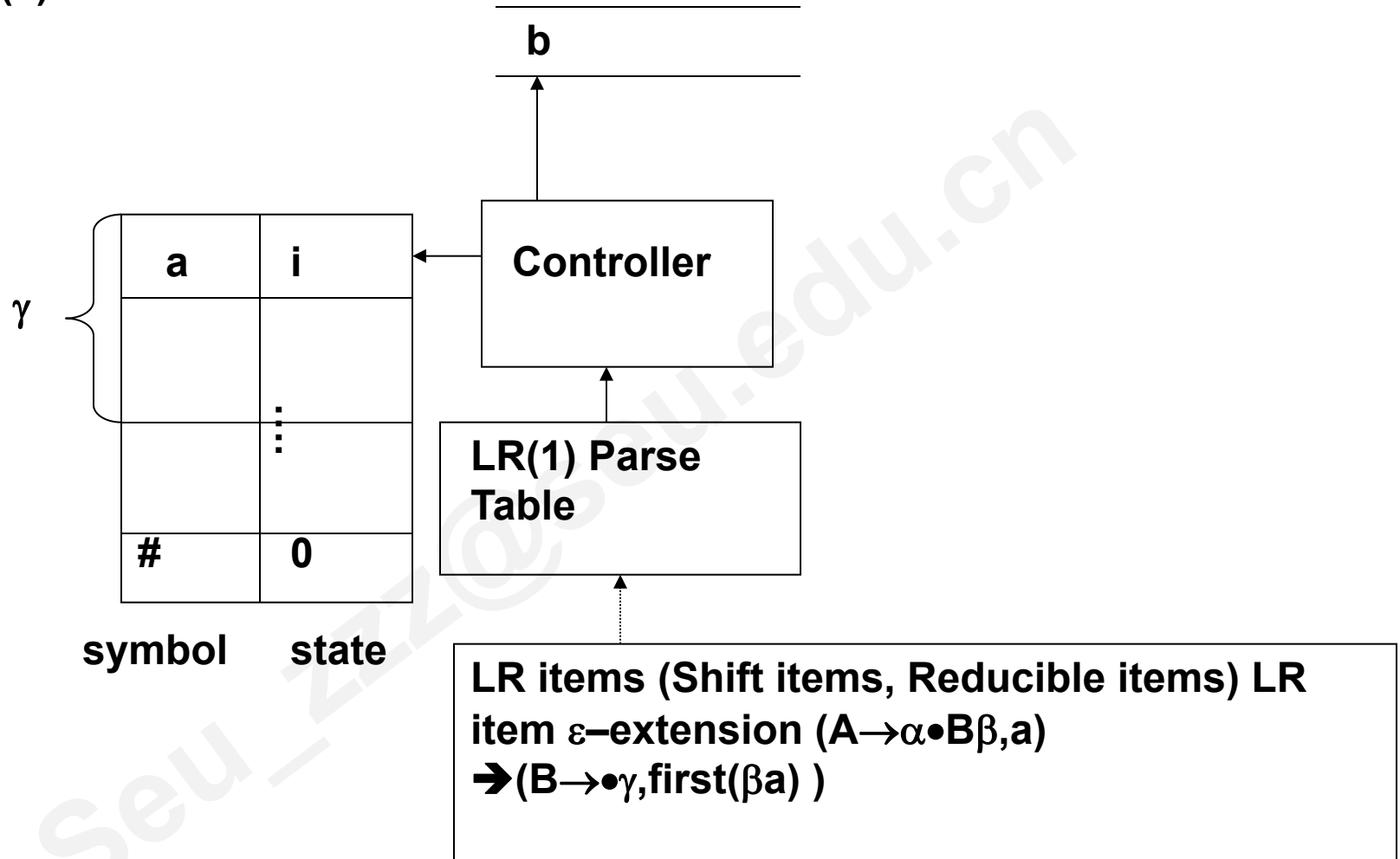
LALR(1)

Skills : Merge states with the same core

Disadvantages: maybe cause reduce-reduce conflict



東南大學 SOUTHEAST UNIVERSITY LR(1) Parser:





東南大學
SOUTHEAST UNIVERSITY

Generation of Parse Tree

Seu_zzz@seu.edu.cn



E.g. construct the **parse tree** for the string “i+i*i” under SLR(1) of the following grammar

0. $S' \rightarrow E$

1. $E \rightarrow E+T$

2. $E \rightarrow T$

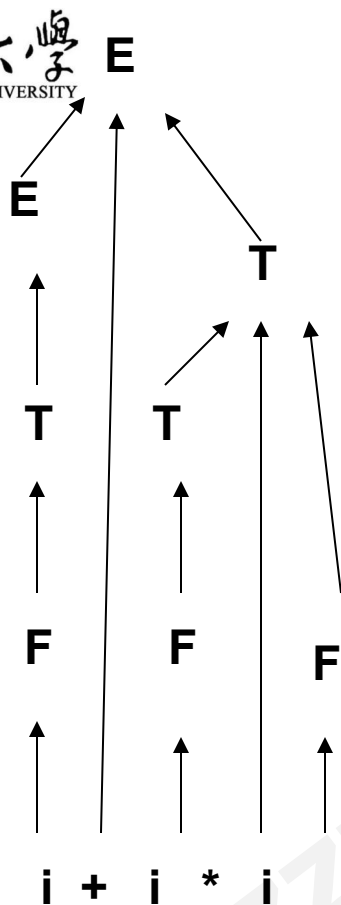
3. $T \rightarrow T * F$

4. $T \rightarrow F$

5. $F \rightarrow (E)$

6. $F \rightarrow i$

stat	ACTION						GOTO		
	i	+	*	()	#	E	T	F
0	S ₅			S ₄			1	2	3
1		S ₆				accept			
2		r ₂	S ₇		r ₂	r ₂			
3		r ₄	r ₄		r ₄	r ₄			
4	S ₅			S ₄			8	2	3
5		r ₆	r ₆		r ₆	r ₆			
6	S ₅			S ₄				9	3
7	S ₅			S ₄					10
8		S ₆			S ₁₁				
9		r ₁			r ₁	r ₁			
10		r ₃	r ₃		r ₃	r ₃			
11		r ₅	r ₅		r ₅	r ₅			



Assignments

- **Constructing the related LL(1) parsing table.**

$P \rightarrow b S d$

$S \rightarrow S ; A \mid A$

$A \rightarrow B \mid C$

$B \rightarrow a$

$C \rightarrow D \mid D e A$

$D \rightarrow E B$

$E \rightarrow i F t$

$F \rightarrow b$

- Please show that the following operator grammar is whether an operator precedence grammar by **constructing the related parsing table.**

$$S \rightarrow S ; G \mid G$$
$$G \rightarrow G(T) \mid H$$
$$H \rightarrow a \mid (S)$$
$$T \rightarrow T + S \mid S$$

- Please **construct** a LR(1) parsing table for the following **two** ambiguous grammar with the additional conditions:.

$S \rightarrow \text{if } S \text{ else } S \mid \text{if } S \mid S; S \mid a$ that **else** dangles with the closest previous unmatched **if** , ; has the property of left associative

$C \rightarrow C \text{ and } C \mid C \text{ or } C \mid b$ that **or** has higher precedence than that of **and**, **and** has the property of right associative, **or** has the property of right associative.