COMPILER DESIGN

UNIT-II in ONE SHOT

Syllabus

Basic Parsing Techniques: Parsers, Shift reduce parsing, operator precedence parsing, top down parsing, predictive parsers Automatic Construction of efficient Parsers: LR parsers, the canonical Collection of LR(0) items, constructing SLR parsing tables, constructing Canonical LR parsing tables, Constructing LALR parsing tables, using ambiguous grammars, an automatic parser generator, implementation of LR parsing tables.

Parsing

Parsing (also known as syntax analysis) is the process of analyzing a sequence of tokens, produced by the lexical analyzer, to determine whether it conforms to the grammatical rules of a programming language.

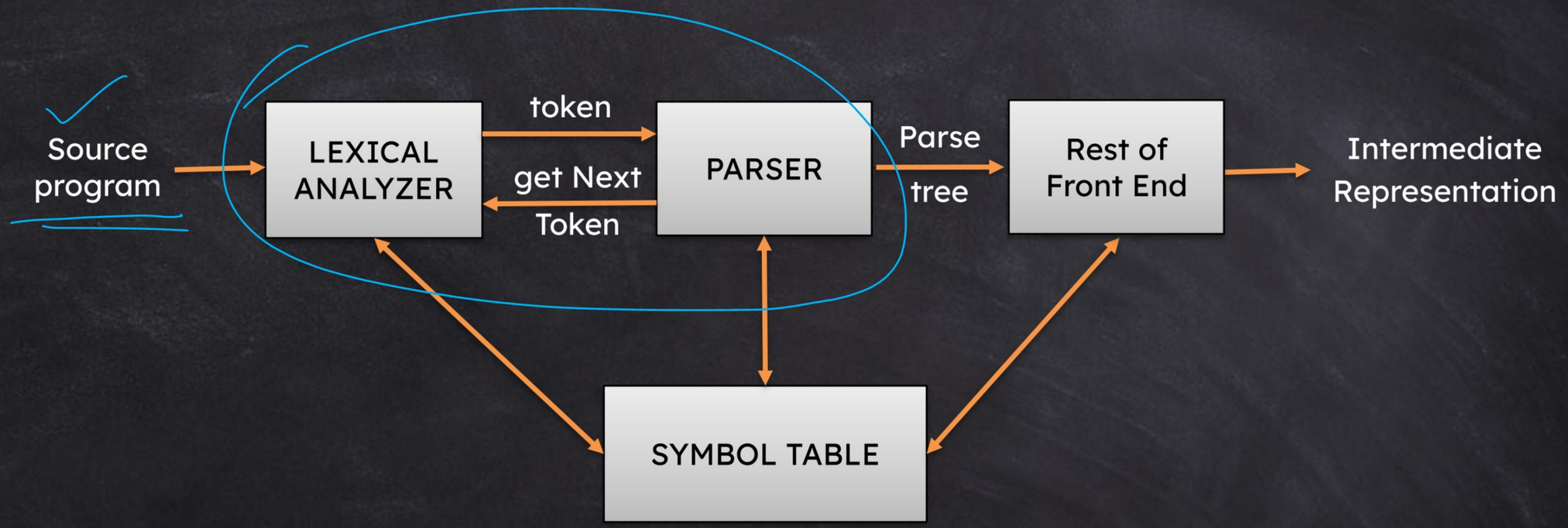
The parser organizes these tokens into a structured representation called a parse tree or syntax tree, which represents the syntactic structure of the source code according to the language's grammar (typically a CFG).

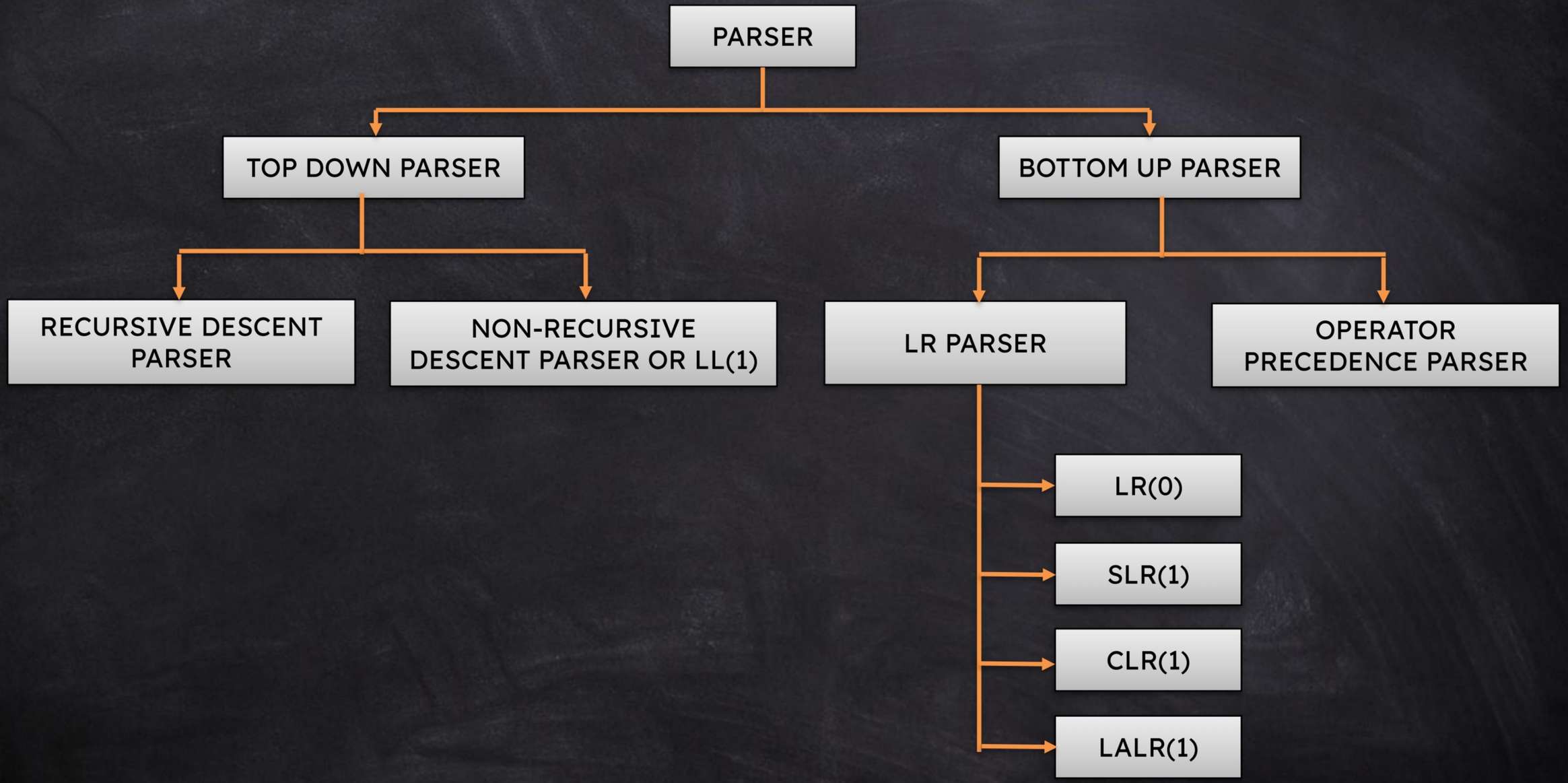
Parsing ensures that the code is syntactically correct and provides a foundation for further analysis, such as semantic analysis and code generation.

Key Aspects of Parsing

Input: A sequence of tokens from the lexical analyzer.
Output: A parse tree or syntax tree, or an error if the input doesn't follow the grammar.
Grammar: Usually a context-free grammar (CFG) that defines the valid syntax of the programming language.

Role of Parser




PARSER
TOP DOWN PARSER
BOTTOM UP PARSER
RECURSIVE DESCENT PARSER
NON-RECURSIVE DESCENT PARSER OR LL(1)
LR PARSER
OPERATOR PRECEDENCE PARSER
LR(0)
SLR(1)
CLR(1)
LALR(1)

Top-Down Parsing

Top-Down Parsing is a parsing strategy that begins with the start symbol of a grammar and attempts to derive the input string by applying production rules from the top (start symbol) to the bottom (terminals).
It tries to build the parse tree from the root to the leaves, matching the input tokens from left to right.

TDP uses leftmost derivation to build parse tree.

$E \rightarrow E + E \mid E * E \mid a \mid b \mid c$

Input: a+b*c

a+b*c

E

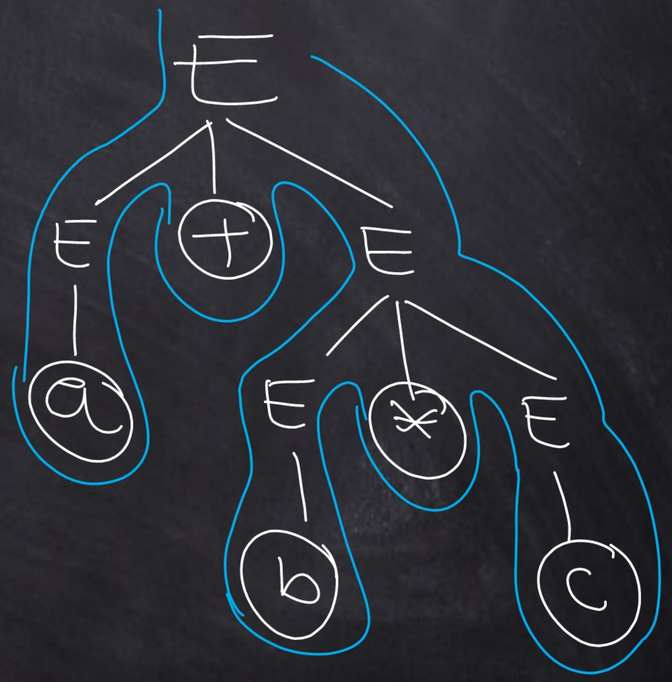
$\rightarrow E + E$ [E → E + E]

$\rightarrow a + E$ [E → a]

$\rightarrow a + E * E$ [E → E * E]

$\rightarrow a + b * E$ [E → b]

$\rightarrow a + b * c$ [E → c]



Bottom-Up Parsing

Bottom-Up Parsing is a parsing strategy that begins with the input string (terminals) and attempts to reconstruct the start symbol of the grammar by applying production rules in reverse, from bottom (terminals) to top (start symbol).

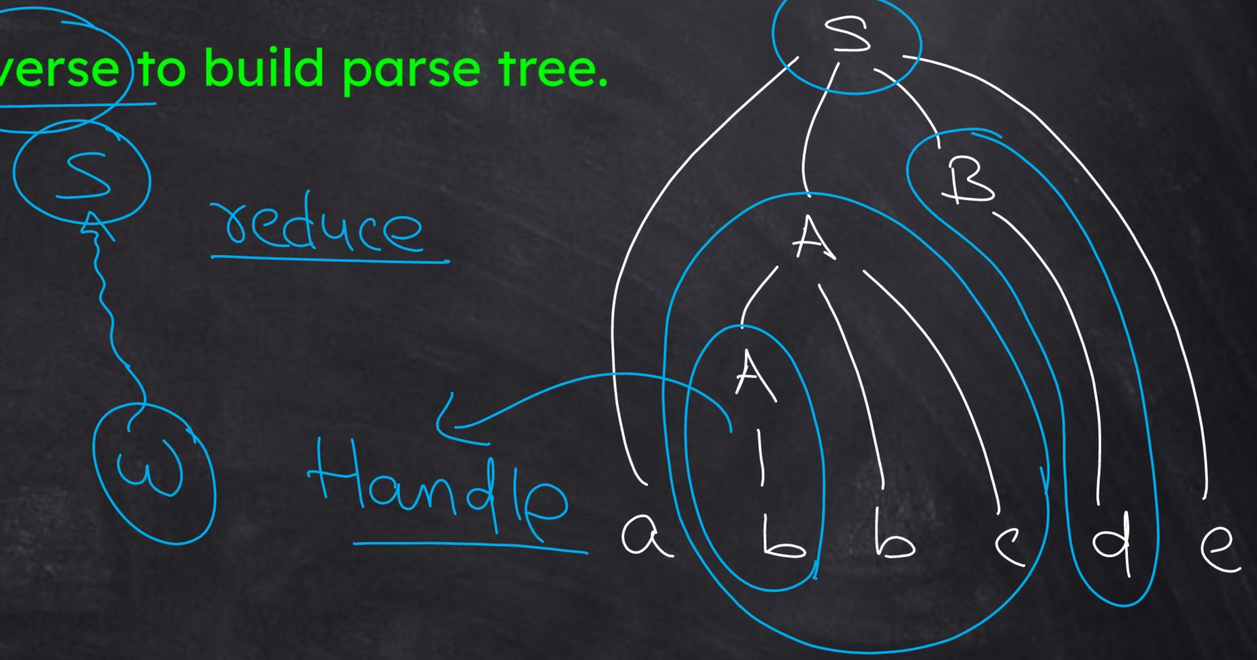
BUP uses rightmost derivation in reverse to build parse tree.

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

Input: abbcde



Recursive Descent Parser

It is a top-down parsing method and uses recursive functions to check whether the input string follows the grammar.
Starts from the start symbol and keeps expanding non-terminals using grammar rules until the input matches.



- Begin with the start symbol.
- For every non-terminal, call a function to expand it using rules.
- Compare terminals with the input.
- Use recursion for non-terminals when required.
- If all rules match → accept; otherwise → error.

For a recursive descent parser, the grammar must be:

- Free of Left Recursion
- Left Factored (No common prefix)

Recursive Descent Parser

$E \rightarrow i\ E'$

$E' \rightarrow +\ i\ E' \mid \varepsilon$

```
void main() {
  E();
  if (input == $)
    parsing successful!
  else
    parsing failed!
}
```

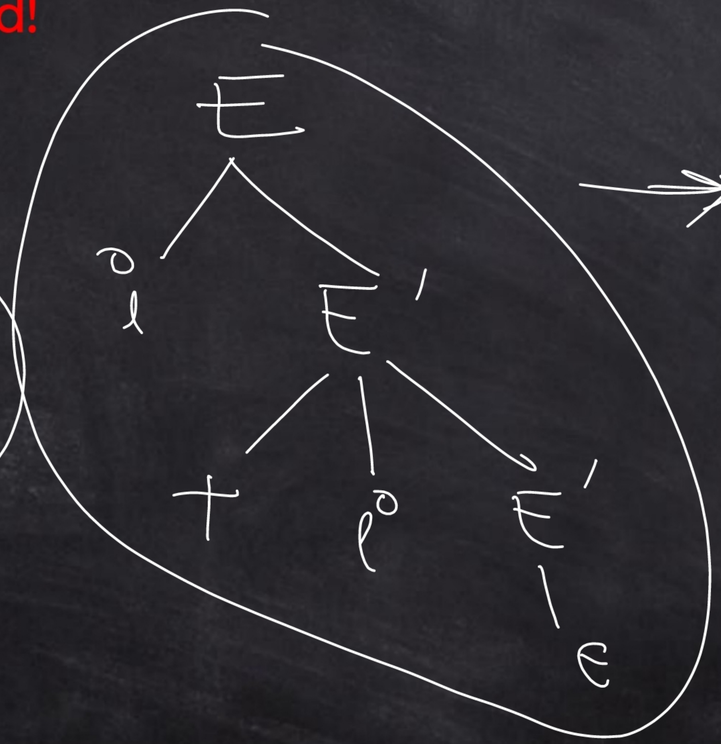
```
void E() {
if (input == i)
    input++;
 E'();
}
```

```
void E'() {
if (input == +){
    input++;
if (input == i)
    input++;
   E'();
}
else
    epsilon
}
```



string = i + i $

input



Recursive Descent Parser

Advantages:

Simple to implement – Easy to write and understand (functions map directly to grammar rules).

Intuitive – Closely follows the structure of grammar, making it easier to debug.

Modular – Each non-terminal has its own function, so the code is well-structured.

Disadvantages:

Not suitable for all grammars – Fails if the grammar has left recursion.

May require backtracking – Can be inefficient if multiple rules must be tried.

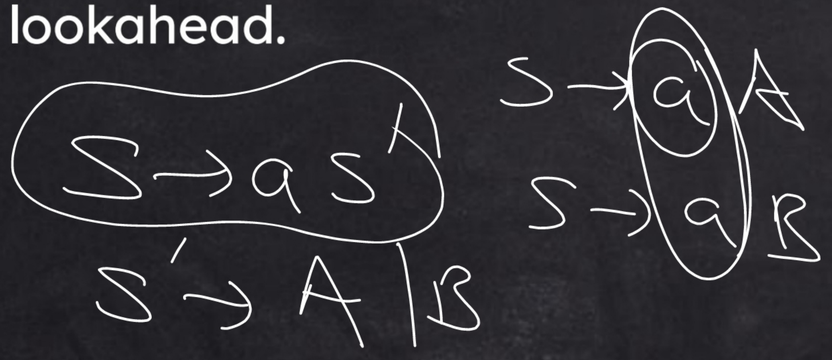
Inefficient for complex grammars – Parsing time can increase due to backtracking.

Left Factoring (No common prefix)

Top-down parsers read input from left to right and decide which production to use based on the next input token.

If multiple productions start with the same sequence of symbols (prefix), the parser cannot decide (non-determinism) which one to choose without looking further ahead — causing a parsing conflict.

Left factoring rewrites the grammar so that this decision can be made with one token of lookahead.



$S \rightarrow aA \mid aB$

$S \rightarrow a\,S'$
$S' \rightarrow A \mid B$

Left Factoring (No common prefix)

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

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

$A \rightarrow aAB \mid aBc \mid aAc$

$A \rightarrow a\,A'$
$A' \rightarrow A\,A'' \mid B\,c$
$A'' \rightarrow B \mid c$

$S \rightarrow iEtSS' \mid a$
$S' \rightarrow \in \mid eS$
$E \rightarrow b$

$A \rightarrow aA'$
$A' \rightarrow AB \mid Bc \mid Ac$
$A' \rightarrow AA'' \mid Bc$
$A'' \rightarrow B \mid c$

Left Factoring (No common prefix)

$S \rightarrow bSSaaS \mid bSSaSb \mid bSb \mid a$

$S \rightarrow b\,S\,S' \mid a$
$S' \rightarrow S\,a\,S'' \mid b$
$S'' \rightarrow a\,S \mid S\,b$

$S \rightarrow a \mid ab \mid abc \mid abcd$

$S \rightarrow a\,S'$
$S' \rightarrow \varepsilon \mid b\,S''$
$S'' \rightarrow \varepsilon \mid c\,S'''$
$S''' \rightarrow \varepsilon \mid d$

$S \rightarrow bSS' \mid a$
$S' \rightarrow SaaS \mid SaSb \mid b$
$S' \rightarrow SaS'' \mid b$
$S'' \rightarrow aS \mid Sb$
$S'' \rightarrow \epsilon \mid cS'''$
$S''' \rightarrow \epsilon \mid d$
$S \rightarrow aS'$
$S' \rightarrow \epsilon \mid b \mid bc \mid bcd$
$S' \rightarrow \epsilon \mid bS''$
$S'' \rightarrow \epsilon \mid c \mid cd$

First and Follow

FIRST(α) is the set of terminals that can appear as the first symbol in string derived from α, plus ε (empty string) if α can derive ε.



Where: α can be a single grammar symbol (terminal or nonterminal) or a sequence of grammar symbols.

FOLLOW(A) for a nonterminal A, to be the set of terminals that can appear immediately to the right of A in some sentential form.

Note: Before calculating the FIRST and FOLLOW functions, ensure that the grammar is free from left recursion. If left recursion is present, eliminate it first.

FIRST(α)

Rule 1: If X is a terminal a, then FIRST(X) = { a }.

f(a) = a
f(aab) = a
f(aBc) = a

Rule 2: If there is a production A → ε, then add ε to FIRST(A).

Rule 3: If A → Y1 Y2 … Yk is a production:
Add every terminal in FIRST(Y1) except ε to FIRST(A).
If ε ∈ FIRST(Y1), then also add terminals from FIRST(Y2) (except ε), and so on.
If all Y1..Yk can derive ε, then add ε to FIRST(A).

S → ε

S → ABC
A → aab | ε
B → bab | ε
C → cab | ε

= {a, b, c, ε}
= {a, ε}
= {b, ε}
= {c, ε}

f(S) = F(ABC)
=



FOLLOW(A)

Rule 1: Place $ in FOLLOW(S), where S is the start symbol, and $ is the input right endmarker.

Rule 2: If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST (β) except ε is in FOLLOW(B).

Rule 3: If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$, where FIRST (β) contains ε, then everything in FOLLOW (A) is in FOLLOW(B)

Note: ε will never appear in the follow function of a non-terminal

$S \rightarrow ABC$	FIRST(S) = {a, b, c, ε}
$A \rightarrow aab \mid \varepsilon$	FIRST(A) = {a, ε}
$B \rightarrow bab \mid \varepsilon$	FIRST(B) = {b, ε}
$C \rightarrow cab \mid \varepsilon$	FIRST(C) = {c, ε}

f(S) = { $ }
f(A) = { b, c, $ }
f(B) = { c, $ }
f(C) = { $ }

First and Follow

S → a B D h
B → c C
C → b C | ε
D → E F
E → g | ε
F → f | ε



FIRST(S) = { a }	FOLLOW(S) = { $ }
FIRST(B) = { c }	FOLLOW(B) = { g, f, h }
FIRST(C) = { b, ε }	FOLLOW(C) = { g, f, h }
FIRST(D) = { g, f, ε }	FOLLOW(D) = { h }
FIRST(E) = { g, ε }	FOLLOW(E) = { f, h }
FIRST(F) = { f, ε }	FOLLOW(F) = { h }

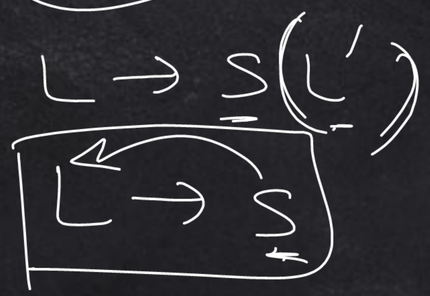


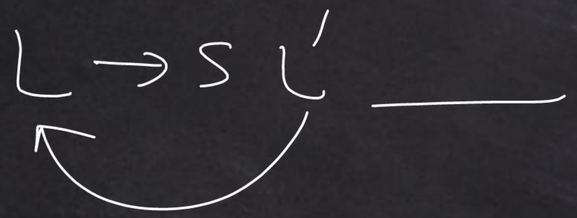
First and Follow

$S \rightarrow (L) \mid a$

$L \rightarrow S\ L'$

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



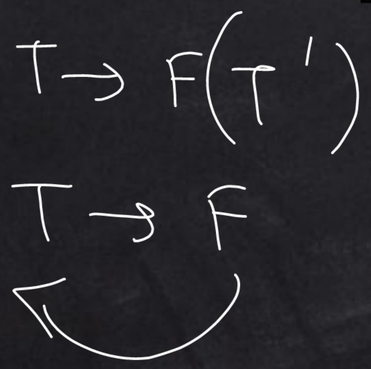


FIRST(S) = { (, a }	FOLLOW(S) = { $, ',' ,) }
FIRST(L) = { (, a }	FOLLOW(L) = {) }
FIRST(L') = { ',' , ε }	FOLLOW(L') = {) }

First and Follow

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

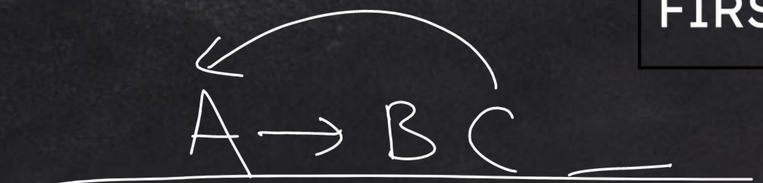
FIRST(E) = { (, id }	FOLLOW(E) = { $,) }
FIRST(E′) = { + , ε }	FOLLOW(E′) = { $,) }
FIRST(T) = { (, id }	FOLLOW(T) = { + , $,) }
FIRST(T′) = { * , ε }	FOLLOW(T′) = { + , $,) }
FIRST(F) = { (, id }	FOLLOW(F) = { * , + , $,) }



First and Follow

$S \rightarrow A\ C\ B \mid C\ b\ B \mid B\ a$
$A \rightarrow d\ a \mid B\ C$
$B \rightarrow g \mid \varepsilon$
$C \rightarrow h \mid \varepsilon$

FIRST(S) = { d, g, h, ε, b, a }	FOLLOW(S) = { $ }
FIRST(A) = { d, g, h, ε }	FOLLOW(A) = { h, g, $ }
FIRST(B) = { g, ε }	FOLLOW(B) = { $, a, h, g }
FIRST(C) = { h, ε }	FOLLOW(C) = { b, h, g, $ }







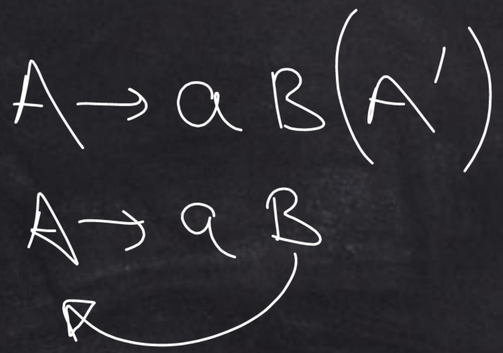
First and Follow

$S \rightarrow A$
$A \rightarrow a\ B \mid A\ d$
$B \rightarrow b$
$C \rightarrow g$

FIRST(S) = { a }	FOLLOW(S) = { $ }
FIRST(A) = { a }	FOLLOW(A) = { $ }
FIRST(A′) = { d, ε }	FOLLOW(A′) = { $ }
FIRST(B) = { b }	FOLLOW(B) = { d, $ }
FIRST(C) = { g }	FOLLOW(C) = ø or { }

Remove left recursion

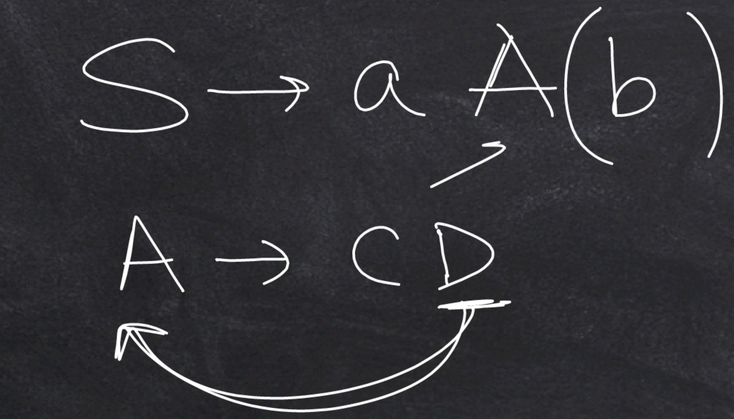
$S \rightarrow A$
$A \rightarrow a\ B\ A'$
$A' \rightarrow d\ A' \mid \varepsilon$
$B \rightarrow b$
$C \rightarrow g$



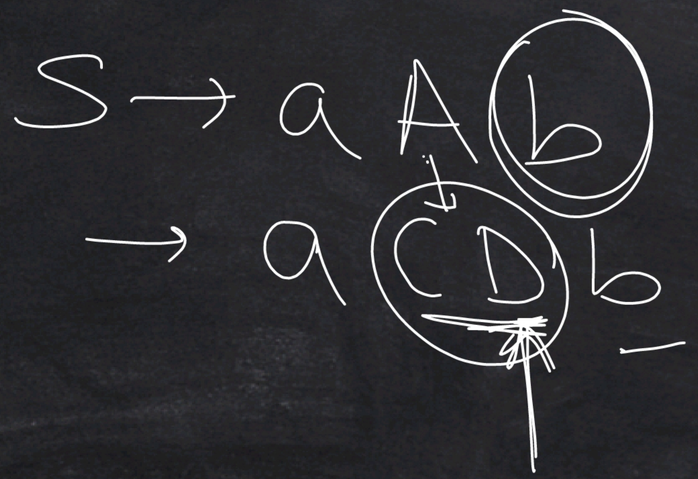
First and Follow

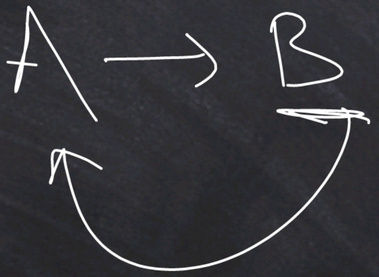
$S \rightarrow A\ a\ A\ b \mid B\ b\ B\ a$
$A \rightarrow \varepsilon$
$B \rightarrow \varepsilon$

FIRST(S) = { a, b }	FOLLOW(S) = { $ }
FIRST(A) = { ε }	FOLLOW(A) = { a, b }
FIRST(B) = { ε }	FOLLOW(B) = { a, b }

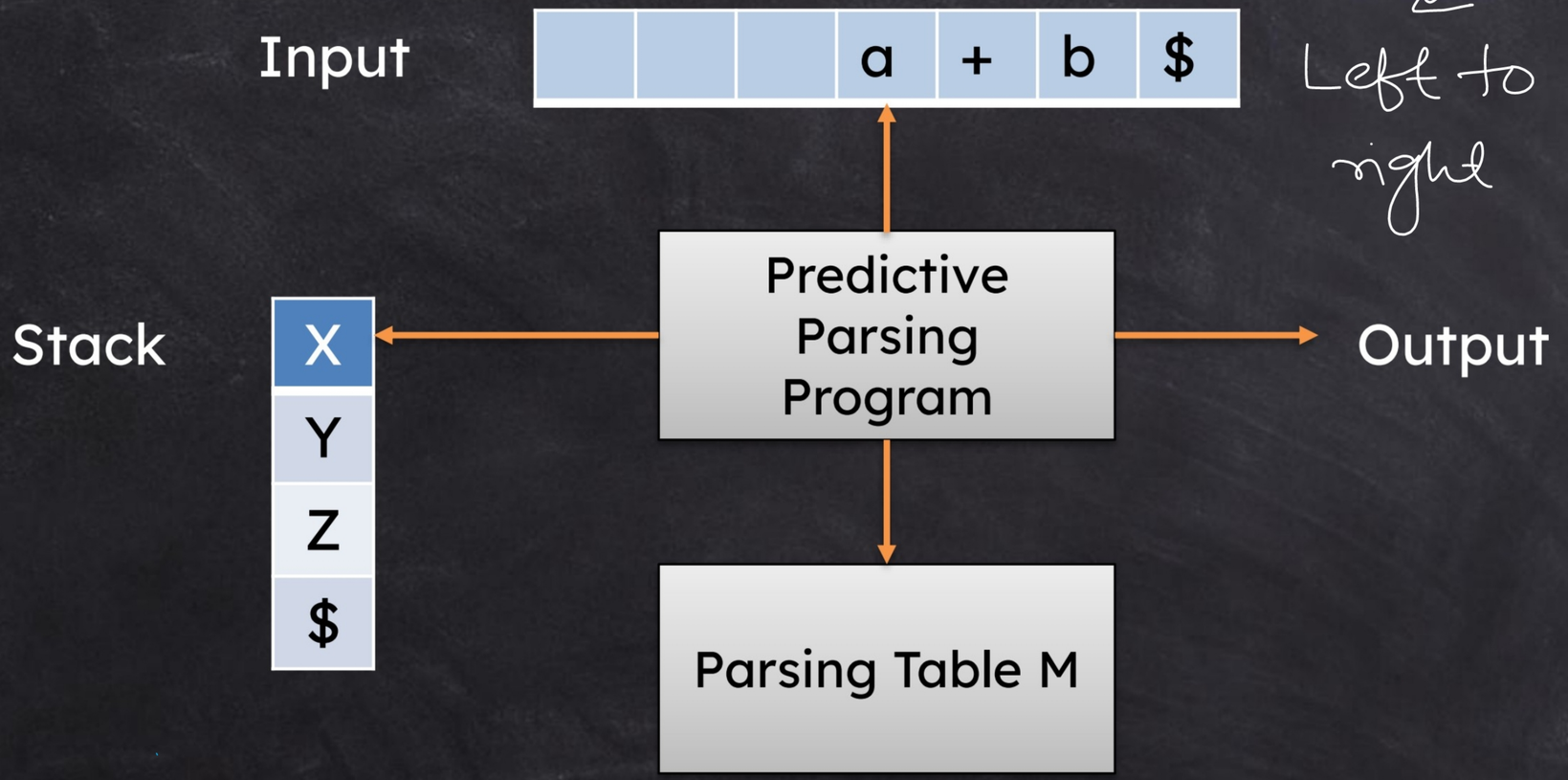

S → a A (b)
A → C D


A → B C ( )

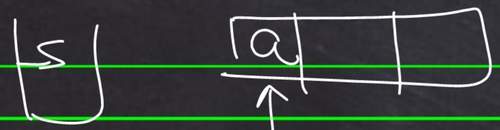

S → a A b
→ a C D b


A → B

Non Recursive Descent Parser or LL(1)



LL(1) parser consists of four components: an input buffer, a stack, a parsing table, and an output stream.
The input buffer holds the string to be parsed and appends a special end marker $ to indicate the end of the input.
The stack stores grammar symbols (non-terminal & terminal), with $ placed at the bottom, indicating the bottom of the stack.
At the beginning, the stack contains the start symbol of the grammar placed on top of $.



The parsing table is a two-dimensional array M[A, a], where:
A is a non-terminal,
a is a terminal symbol or the end marker $.
During parsing, the program checks:
X is the symbol at the top of the stack,
a is the current input symbol.

	id	$
E	E → T E′	
E′		E′→ε

Limitations of LL(1) Grammar

Note 1: No Left Recursion

LL(1) grammar must not have left recursion.

Note 2: No Ambiguity

LL(1) grammar must not be ambiguous. 

Note 3: No Common Prefix

LL(1) grammar must not have common prefixes in productions

A grammar cannot be LL(1) if it has Left Recursion, Ambiguity, or Common Prefix in its productions.

LL(1) Parsing Table Construction

Fill the Table rows = nonterminals, columns = terminals ∪ {$}.

For every production $A \to \alpha$:

1. For each $a \in FIRST(\alpha)$ with $a \neq \epsilon$ put $A \to \alpha A$ in M[A,a].
2. If $\epsilon \in FIRST(\alpha)$: for each $b \in FOLLOW(A)$, put $A \to \alpha$ in M[A,b].

If, after performing the above, there is no production at all in M[A,a], then set M[A,a] to error.

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

FOLLOW(S) = { $, ',' ,) }
FOLLOW(L) = {) }
FOLLOW(L') = {) }

$L' \rightarrow \epsilon$
$=$
follow(L')

	(	)	a	,	$
S	$S \rightarrow (L)$	X	$S \rightarrow a$	X	
L	$L \rightarrow S\ L'$		$L \rightarrow S\ L'$		
L'		$L' \rightarrow \varepsilon$		$L' \rightarrow , S\ L'$	

Initialize:

Push $ (end marker) and start symbol S on stack.

Set input pointer to first symbol of input string.

Repeat until stack top = $:

Let X = top(stack) and a = current input symbol.

Case 1: If X is a terminal or $:

- If X = a, then pop stack and advance input pointer.
- Else, **error** (mismatch).

Case 2: If X is a non-terminal:

- If M[X, a] contains a production X → Y1 Y2 … Yk:
  - Pop X.
  - Push Yk … Y2 Y1 onto stack (in reverse order).
  - Output the production used.
- Else, **error** (parsing table entry empty).

If both stack and input are $:

- **Accept.**

Stack	Input	Action
$ S	(a,a)$	S→(L)
$) L (	(a,a)$	pop
$) L	a,a)$	L→S L′
$) L′ S	a,a)$	S→a
$) L′ a	a,a)$	pop
$) L′	,a)$	L′→, S L′
$) L′ S ,	,a)$	pop
$) L′ S	a)$	S→a
$) L′ a	a)$	pop
$) L′	)$	L′→ε
$)	)$	pop
$	$	accepted

LL(1) Parsing Table Construction

$E \rightarrow T\ E'$

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

$T \rightarrow F\ T'$

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

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

FOLLOW(E) = { $,) }
FOLLOW(E′) = { $,) }
FOLLOW(T) = { + , $,) }
FOLLOW(T′) = { + , $,) }
FOLLOW(F) = { * , + , $,) }

$E' \rightarrow \epsilon$

	id	+	*	(	)	$
E	E → T E′			E → T E′		
E′	—	E′→+T E′			E′→ε	E′→ε
T	T → F T′			T → F T′		
T′		T′→ε	T′→*F T′		T′→ε	T′→ε
F	F → id			F → (E)		

Initialize:

Push $ (end marker) and start symbol S on stack.

Set input pointer to first symbol of input string.

Repeat until stack top = $:

Let X = top(stack) and a = current input symbol.

Case 1: If X is a terminal or $:

- If X = a, then pop stack and advance input pointer.
- Else, **error** (mismatch).

Case 2: If X is a non-terminal:

- If M[X, a] contains a production X → Y1 Y2 … Yk:
  - Pop X.
  - Push Yk … Y2 Y1 onto stack (in reverse order).
  - Output the production used.
- Else, **error** (parsing table entry empty).

If both stack and input are $:

- **Accept.**

Stack	Input	Output
$E	id+id*id$	E → T E′
$E′ T	id+id*id$	T → F T′
$E′ T′ F	id+id*id$	F → id
$E′ T′ id	id+id*id$	pop
$E′ T′	+id*id$	T′ → ε
$E′	+id*id$	E′ → + T E′
$E′ T+	+id*id$	pop
$E′ T	id*id$	T → F T′
$E′ T′ F	id*id$	F → id
$E′ T′ id	id*id$	pop
$E′ T′	*id$	T′ → * F T′
$E′ T′ F *	*id$	pop
$E′ T′ F	id$	F → id
$E′ T′ id	id$	pop
$E′ T′	$	T′ → ε
$E′	$	E′ → ε
$	$	Accepted

$S \rightarrow a\ B\ D\ h$

$B \rightarrow c\ C$

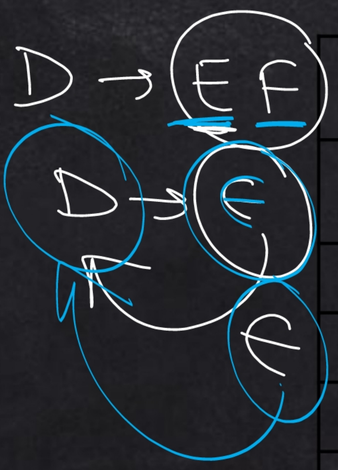
$C \rightarrow b\ C \mid \varepsilon$

$D \rightarrow E\ F$

$E \rightarrow g \mid \varepsilon$

$F \rightarrow f \mid \varepsilon$

FOLLOW (S) ={$}
FOLLOW (B) = { g, f, h }
FOLLOW (C) = { g, f, h }
FOLLOW (D) = {h}
FOLLOW (E) = {f, h}
FOLLOW (F) = {h}



	a	b	c	f	g	h	$
S	$S \rightarrow a\ B\ D\ h$						
B			$B \rightarrow c\ C$				
C		$C \rightarrow b\ C$		$C \rightarrow \varepsilon$	$C \rightarrow \varepsilon$	$C \rightarrow \varepsilon$	
D				$D \rightarrow E\ F$	$D \rightarrow E\ F$	$D \rightarrow E\ F$	
E				$E \rightarrow \varepsilon$	$E \rightarrow g$	$E \rightarrow \varepsilon$	
F				$F \rightarrow f$		$F \rightarrow \varepsilon$	

Stack	Input	Output
$S	acbh$	S → a B D h
$hDBa	acbh$	pop
$hDB	cbh$	B → c C
$hDCc	cbh$	pop
$hDC	bh$	C → b C
$hDCb	bh$	pop
$hDC	h$	C → ε
$hD	h$	D → E F
$hFE	h$	E → ε
$hF	h$	F → ε
$h	h$	pop
$	$	Accept

	a	b	c	f	g	h	$
S	S → a B D h						
B			B → c C				
C		C → b C		C → ε	C → ε	C → ε	
D				D → E F	D → E F	D → E F	
E				E → ε	E → g	E → ε	
F				F → f		F → ε	

Practice Problems

$S \rightarrow A\ C\ B \mid C\ b\ B \mid B\ a$
$A \rightarrow d\ a \mid B\ C$
$B \rightarrow g \mid \varepsilon$
$C \rightarrow h \mid \varepsilon$



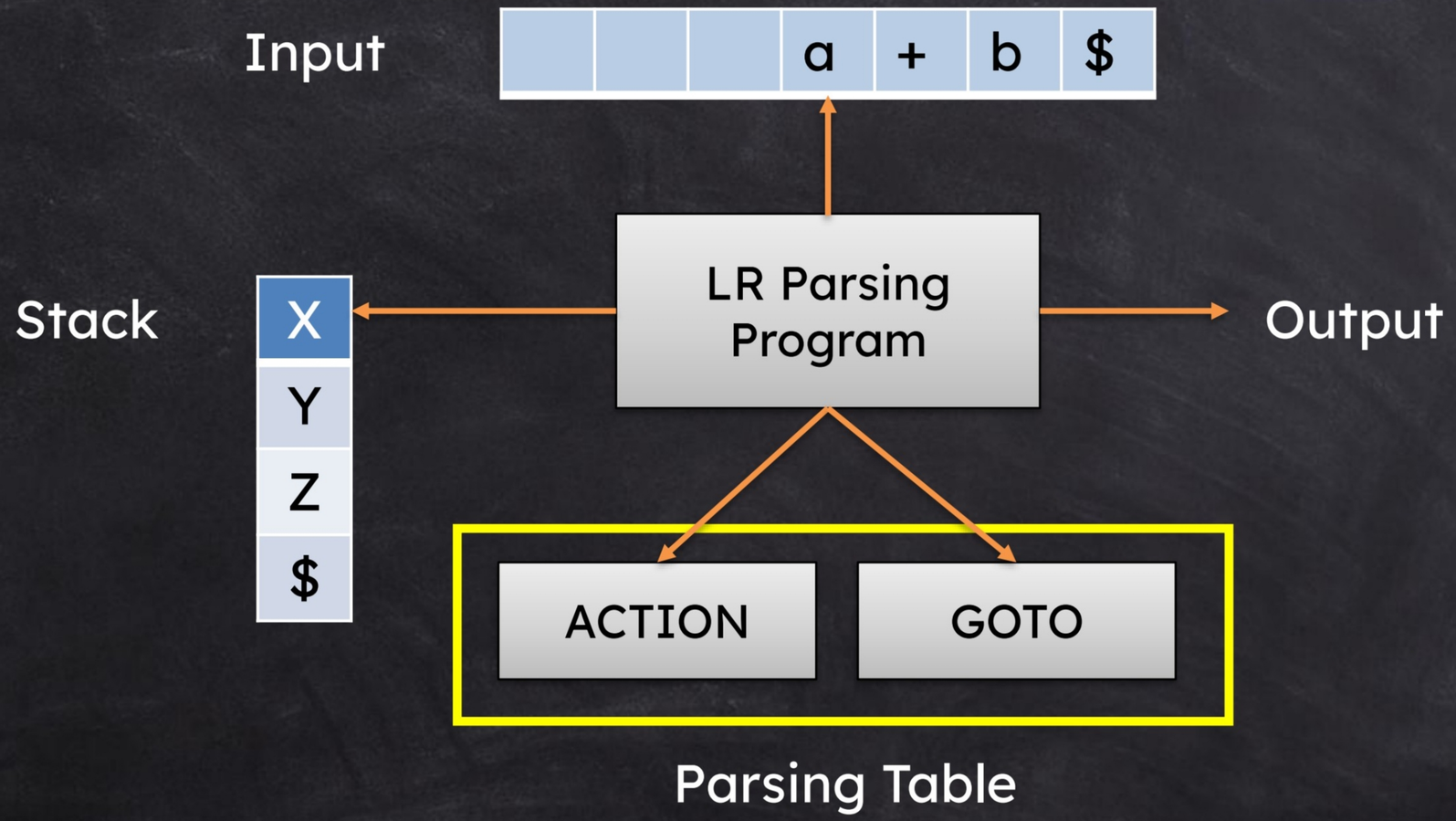
$S \rightarrow A$
$A \rightarrow a\ B\ A'$
$A' \rightarrow d\ A' \mid \varepsilon$
$B \rightarrow b$
$C \rightarrow g$



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



LR Parser


Input
a
+
b
$
Stack
X
Y
Z
$
LR Parsing Program
Output
ACTION
GOTO
Parsing Table

The input buffer holds the string to be parsed and appends a special end marker $ to indicate the end of the input.

Stack stores Grammar symbols (terminals + non-terminals) and States (from the parsing DFA).

The parsing table has two parts:

ACTION Table:

- Tells the parser whether to Shift, Reduce, Accept, or Error.
- Rows → states.
- Columns → terminal symbols.

GOTO Table:

- Used after a Reduce.
- Tells which state to move to for a given non-terminal.
- Rows → states.
- Columns → non-terminals.

LR(0) item

An LR(0) item is a production rule with a dot (•) showing how much of the RHS has been seen.

$E \rightarrow \bullet\ E + T$ (before parsing any symbol)
$E \rightarrow E \bullet + T$ (after parsing E)
$E \rightarrow E + \bullet\ T$ (after parsing E +)
$E \rightarrow E + T \bullet$ (after parsing E + T)

The dot (•) tells us the parser's progress in recognizing that production.

Closure Operation: If you have an item with a dot (•) before a non-terminal, Closure expands it by adding all productions of that non-terminal (with dot (•) at the start).

👉 Closure keeps adding items until no more new items can be added.

Grammar

$E \rightarrow E + T \mid T$ (1) (2)

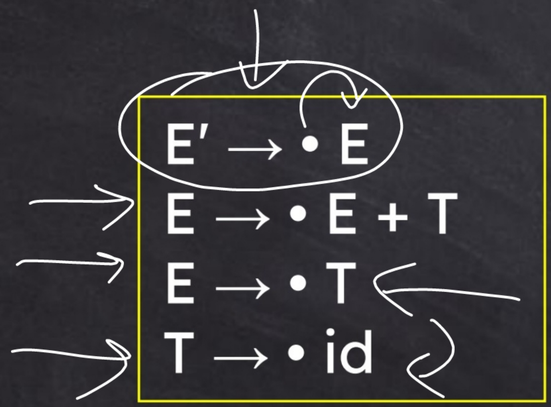
$T \rightarrow id$ (3)

Augmented Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow id$



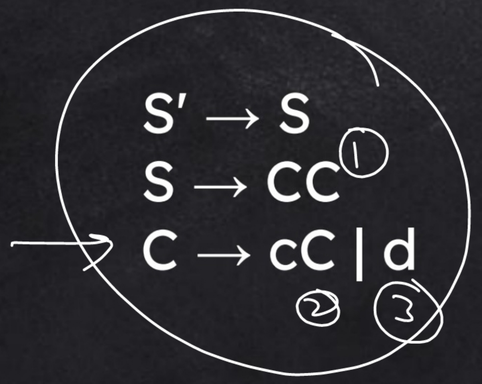
Goto Operation: The Goto(I, X) function means:

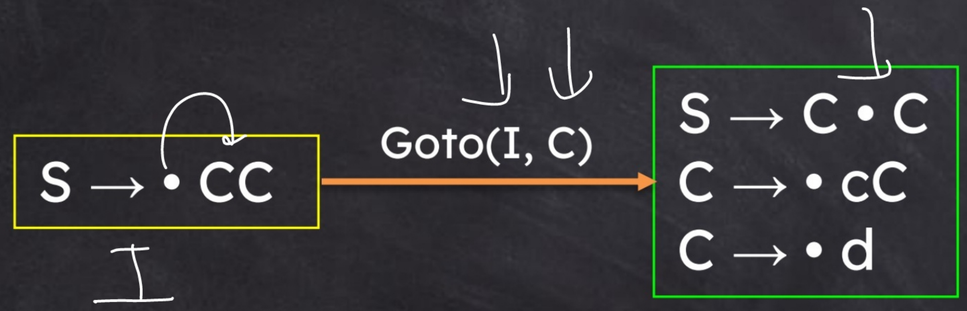
From a set of items I

If the dot (•) is before symbol X,

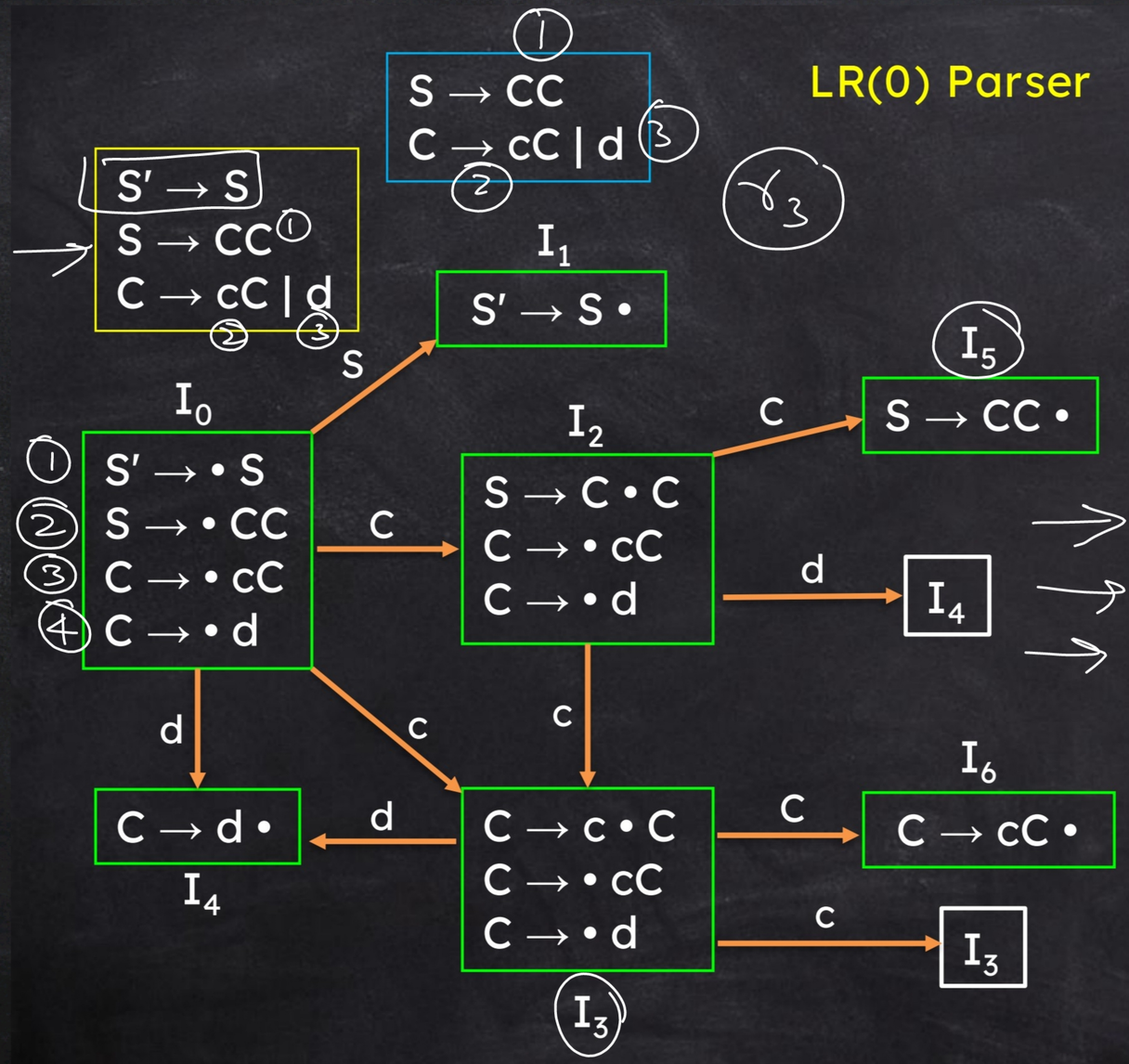
Move the dot past X,

Take the Closure of the result.





LR(0) Parser



State	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

LR(0) Parser

(1) $S \rightarrow CC$

(2) (3) $C \rightarrow cC \mid d$

$S \rightarrow CC$

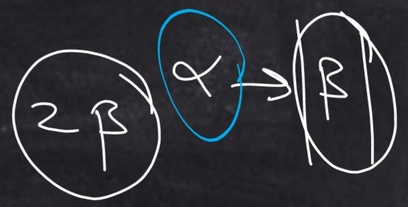
Shift ⇒ push

reduce ⇒ pop.

$C \rightarrow d$

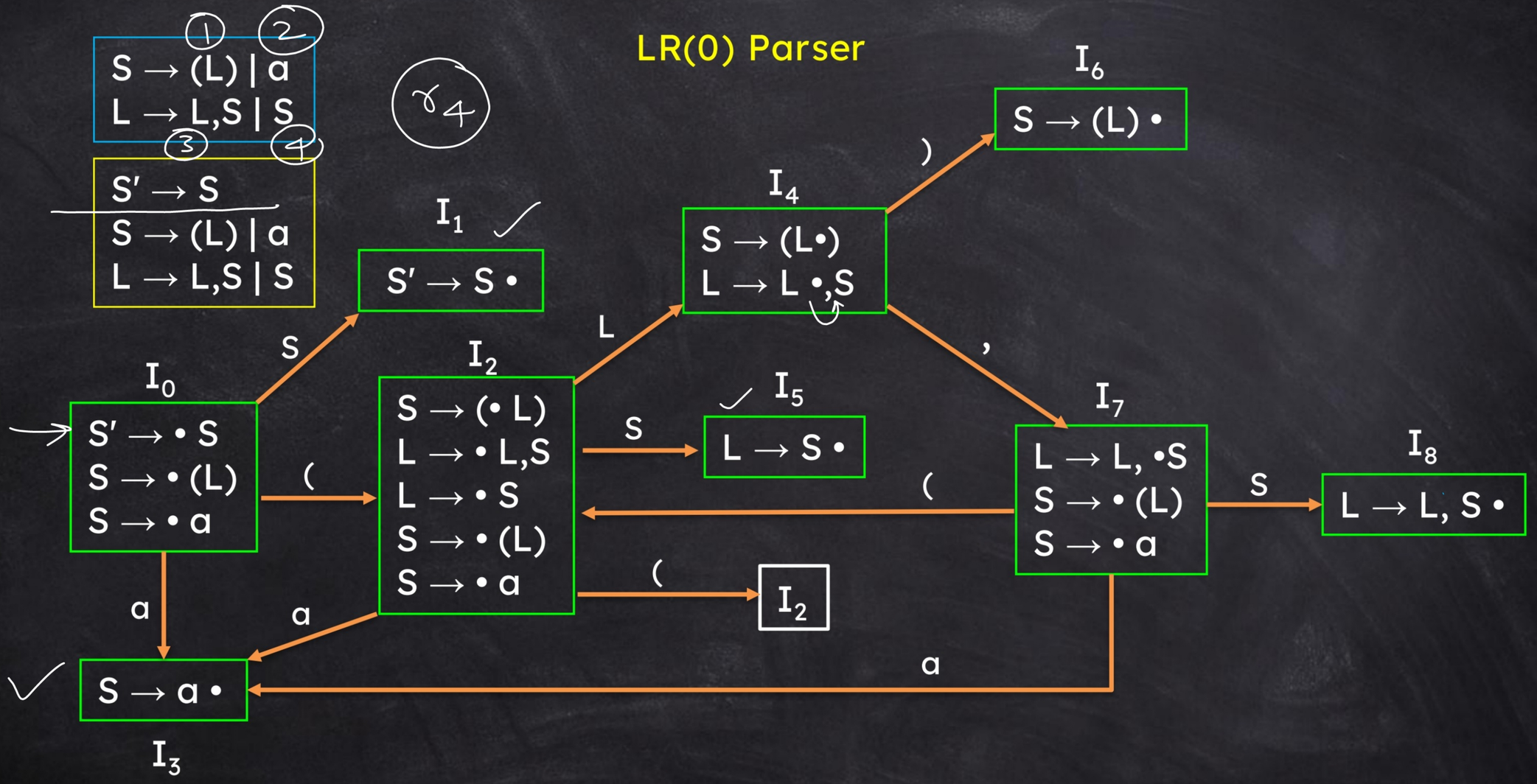
Stack	Input	Action
0	d d $	Shift d
0 d 4	d $	Reduce C → d
0 C 2	d $	Shift d
0 C 2 d 4	$	Reduce C → d
0 C 2 C 5	$	Reduce S → C C
0 S 1	$	Accept

State	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		



Input = cdd$

LR(0) Parser



1 2
S → (L) | a
L → L,S | S
3 4
S′ → S
S → (L) | a
L → L,S | S
r4
I0
S′ → • S
S → • (L)
S → • a
I1
S′ → S •
I2
S → (• L)
L → • L,S
L → • S
S → • (L)
S → • a
I3
S → a •
I4
S → (L•)
L → L •,S
I5
L → S •
I6
S → (L) •
I7
L → L, •S
S → • (L)
S → • a
I8
L → L, S •
S
(
a
L
)
,

LR(0) Parser

State	ACTION					GOTO	
	(	)	,	a	$	S	L
0	s2			s3		1	
1					acc		
2	s2			s3		5	4
3	r2	r2	r2	r2	r2		
4		s6	s7				
5	r4	r4	r4	r4	r4		
6	r1	r1	r1	r1	r1		
7	s2			s3		8	
8	r3	r3	r3	r3	r3		

LR(0) Parser

Stack	Input	Action
0	(a , a) $	shift (| goto 2 (s2)
0 (2	a , a) $	shift a | goto 3 (s3)
0 (2 a 3	, a) $	reduce r2: S → a (pop a & 3 | goto 5)
0 (2 S 5	, a) $	reduce r4: L → S (pop S & 5 | goto 4)
0 (2 L 4	, a) $	shift , | goto 7 (s7)
0 (2 L 4 , 7	a) $	shift a | goto 3 (s3)
0 (2 L 4 , 7 a 3	) $	reduce r2: S → a (pop a & 3 | goto 8)
0 (2 L 4 , 7 S 8	) $	reduce r3: L → L , S (pop L 4 , 7 S 8 | goto 4)
0 (2 L 4	) $	shift) | goto 6 (s6)
0 (2 L 4) 6	$	reduce r1: S → (L) (pop (2 L 4) 6 | goto 1)
0 S 1	$	accept

LR(0) Parser

An LR(0) parser is a type of bottom-up parser (shift-reduce parser) used to analyze whether a given input string can be generated by a grammar.

It is the simplest LR parser and is based on LR(0) items, where the “0” means zero lookahead (it makes parsing decisions without looking at the next input symbol).

Drawbacks of LR(0) Parser

LR(0) parsers do not use any lookahead symbol, that means when the parser is in a state with an item like $A \rightarrow \alpha \bullet$, it must reduce on all inputs — often causing shift/reduce or reduce/reduce conflicts.

More powerful variants like SLR(1), LR(1), and LALR(1) are used in practice.

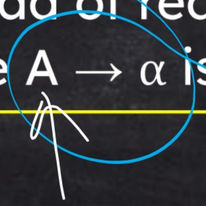
Although LR(0) tables are smaller than LR(1), their parsing power is too weak.

SLR(1) Parser

SLR = Simple LR (or SLR(1)).
The “1” means it uses 1 lookahead symbol to make parsing decisions.
It is an improvement over LR(0):
LR(0) uses no lookahead.
SLR(1) uses the FOLLOW set of non-terminals to decide reductions.

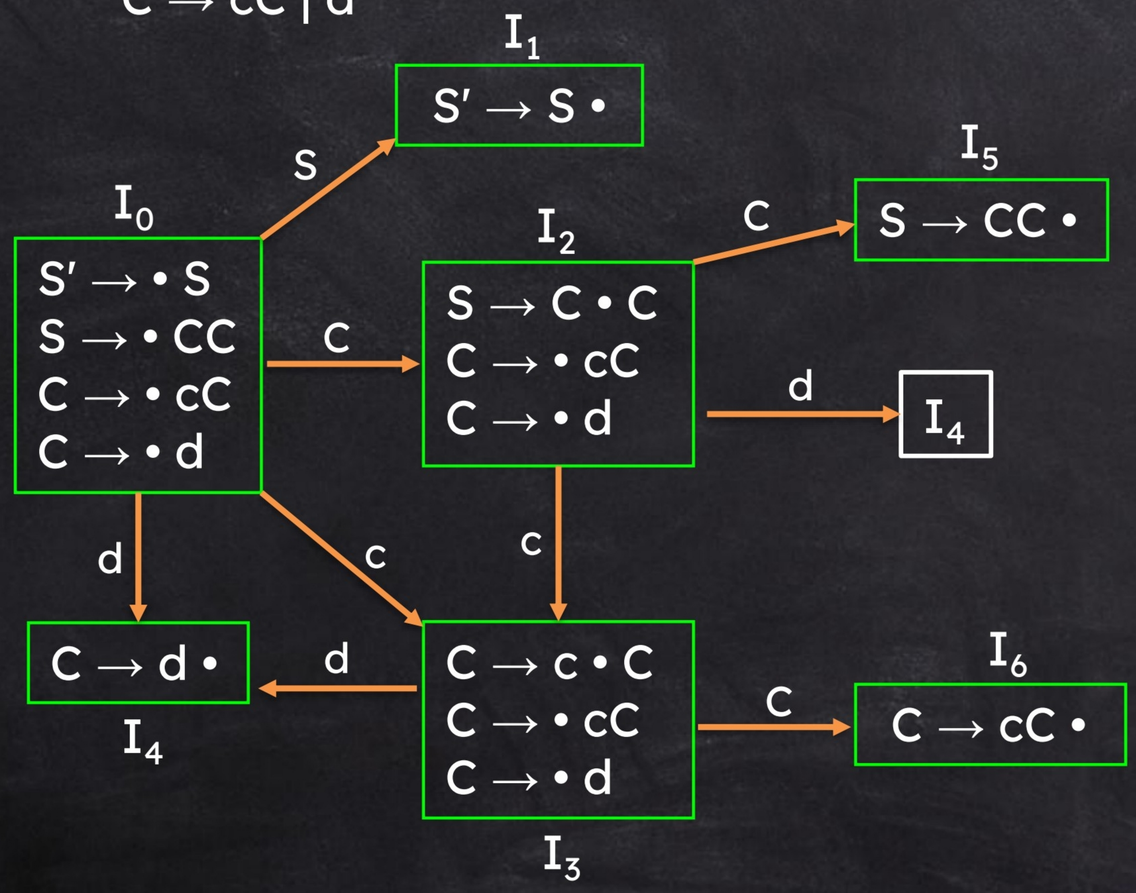
How SLR(1) Works

1. Construct LR(0) Parsing Table (states/items as in LR(0)).
2. Build ACTION & GOTO tables as in LR(0).
   For shift entries: same as LR(0).
   For reduce entries: instead of reducing on all inputs, reduce only on tokens in FOLLOW(A) (where $A \rightarrow \alpha$ is the reduction rule).



SLR(1) Parser

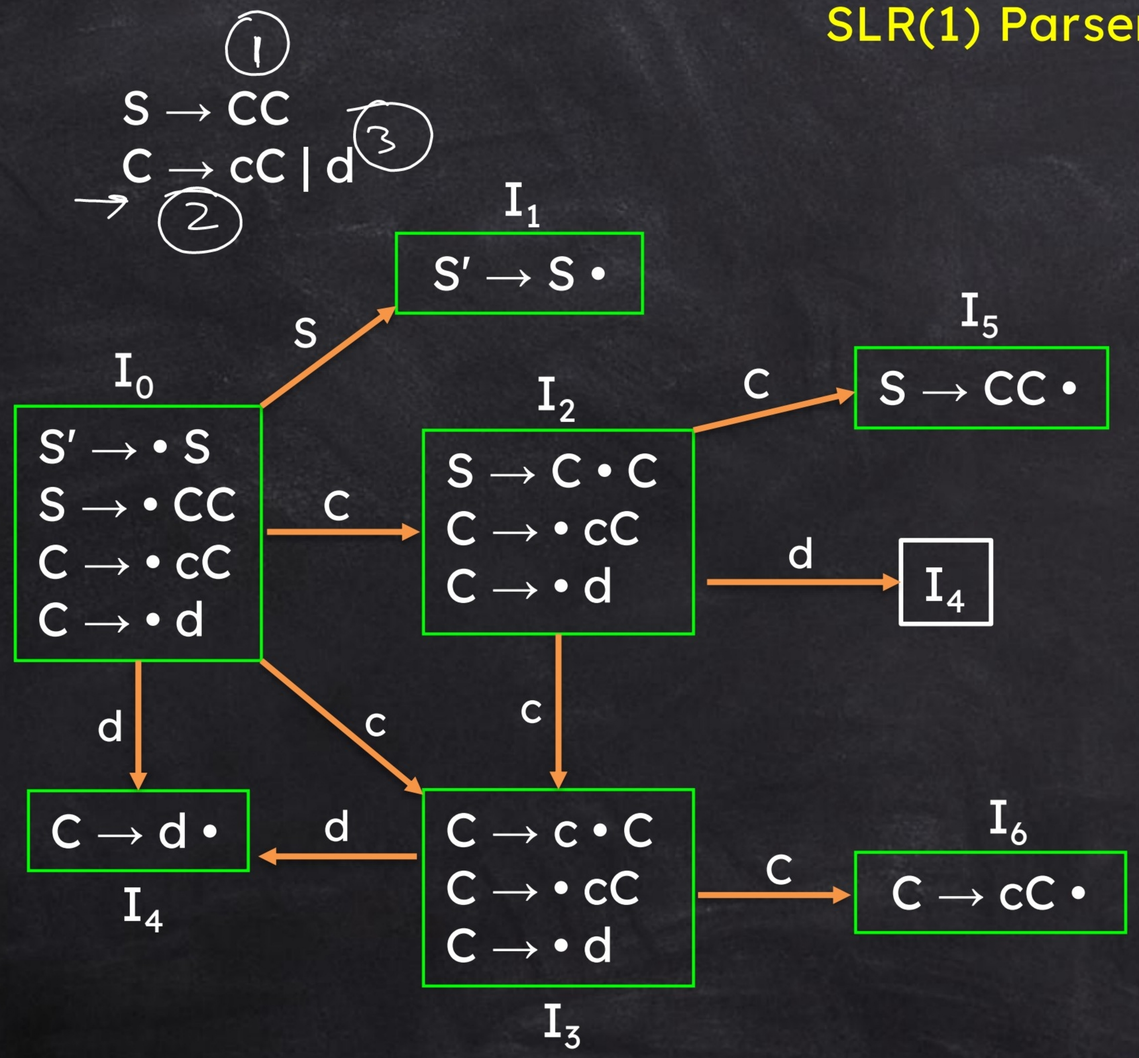
$S \rightarrow CC$
$C \rightarrow cC \mid d$



State	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

LR(0) Parsing Table

SLR(1) Parser

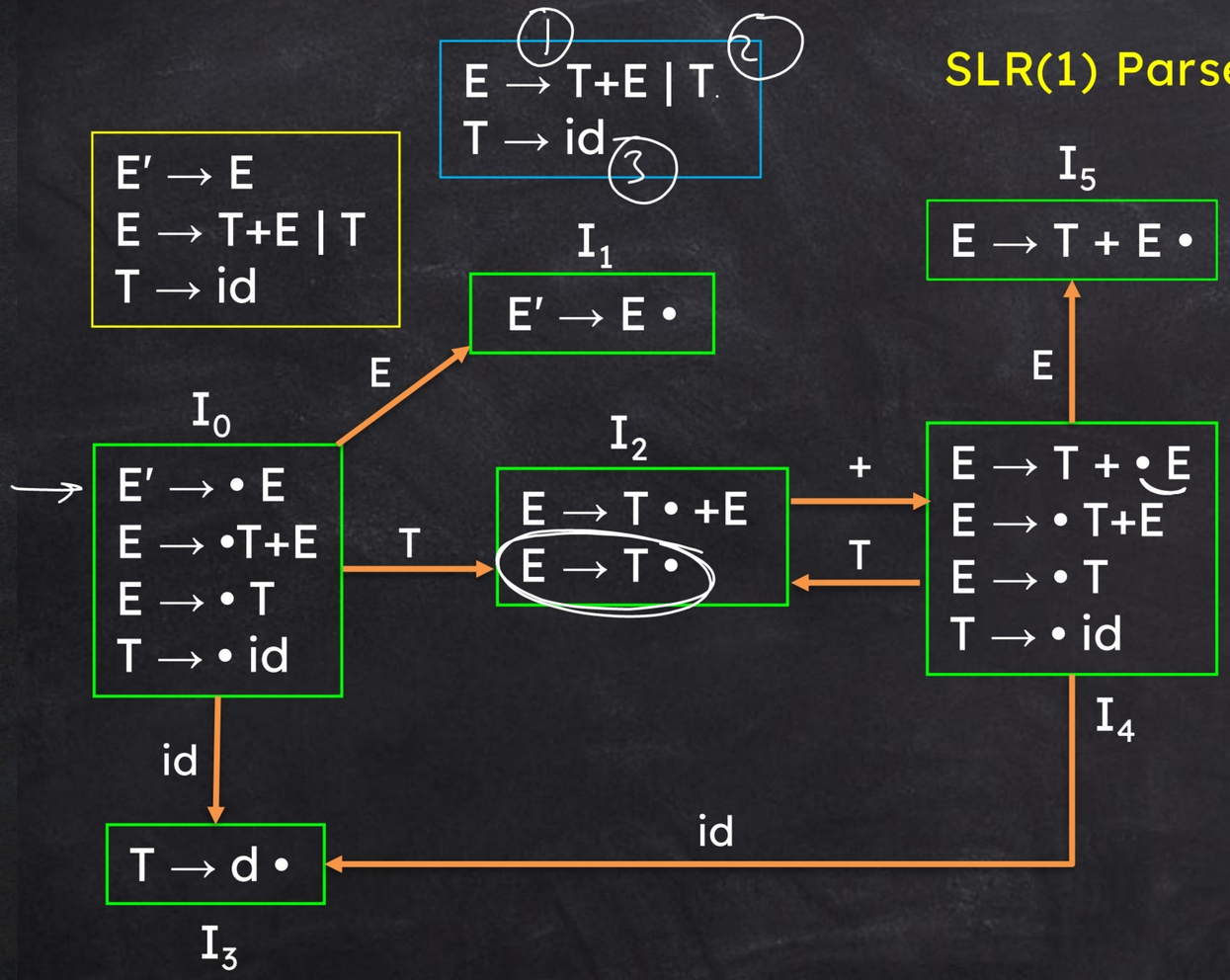


State	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5			r1		
6	r2	r2	r2		

SLR(1) Parsing Table

FOLLOW(S) = { $ }
FOLLOW(C) = { c, d, $ }

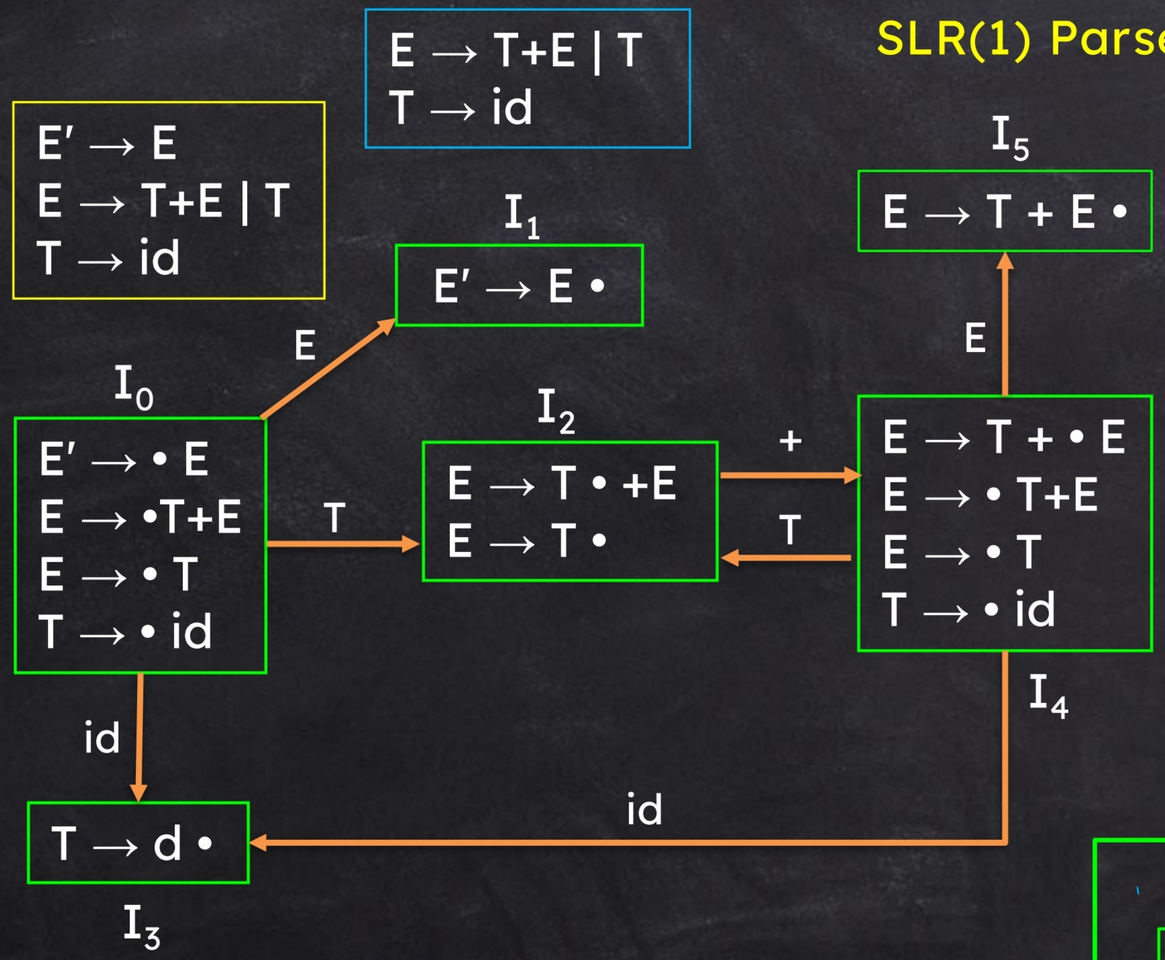
SLR(1) Parser



State	ACTION			GOTO	
	id	+	$	E	T
0	s3			1	2
1			acc		
2	r2	s4/r2	r2		
3	r3	r3	r3		
4	s3			5	2
5	r1	r1	r1		

LR(0) Parsing Table

SLR(1) Parser



State	ACTION			GOTO	
	id	+	$	E	T
0	s3			1	2
1			acc		
2		s4	r2		
3		r3	r3		
4	s3			5	2
5			r1		

SLR(1) Parsing Table

SLR(1)

LR(0)

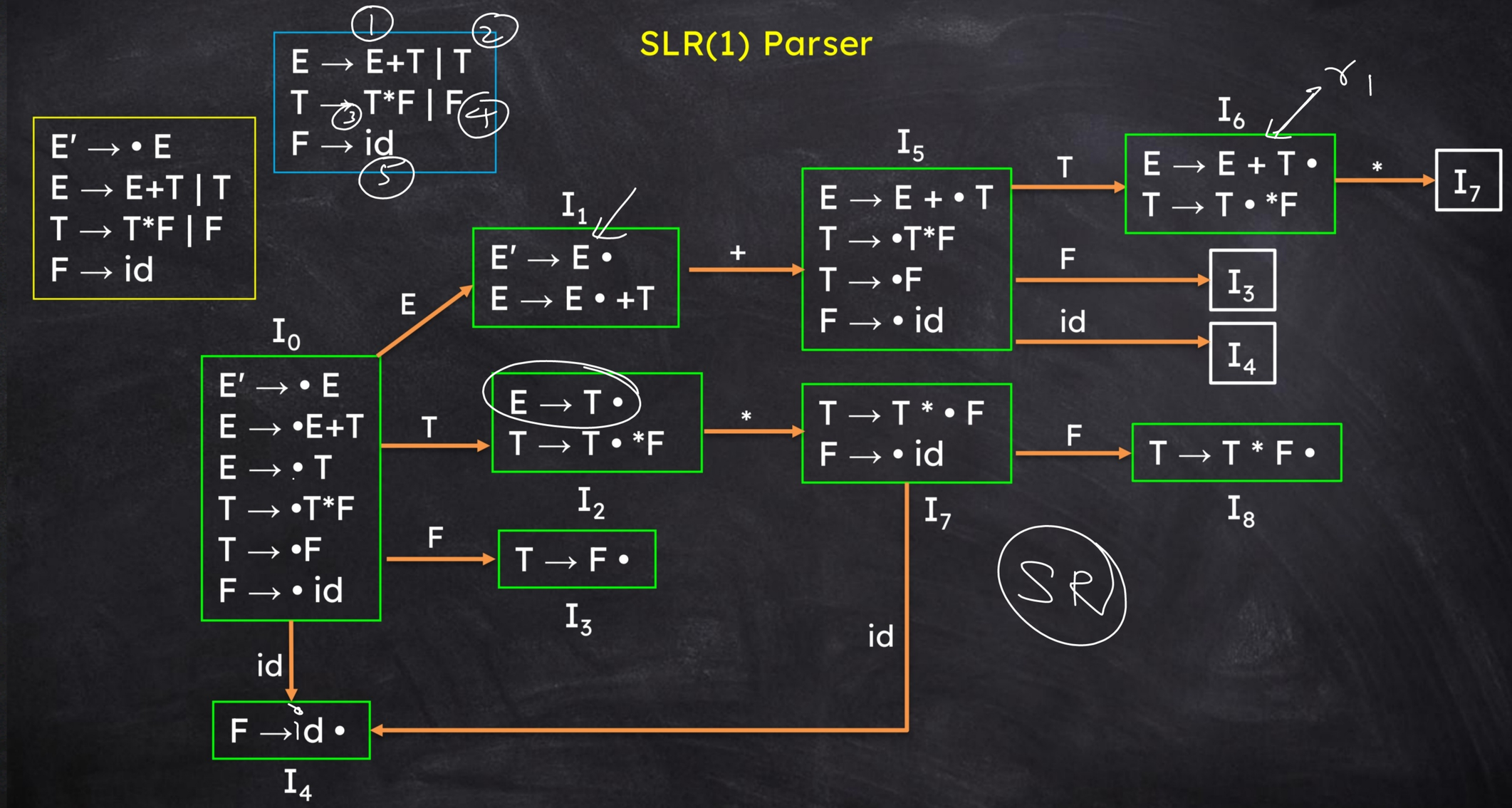
FOLLOW(E) = { $ }

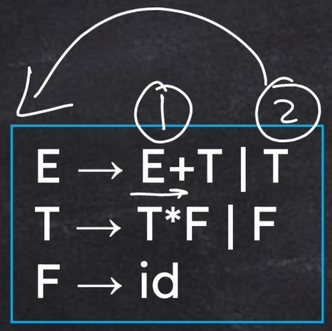
FOLLOW(T) = { +, $ }

SLR(1) Parser

Stack	Input	Action
0	id+id$	shift id | goto 3 (s3)
0 id 3	+id$	reduce r3: T → id (pop id & 3 | goto 2)
0 T 2	+id$	shift + | goto 4 (s4)
0 T 2 + 4	id$	shift id | goto 3 (s3)
0 T 2 + 4 id 3	$	reduce r3: T → id (pop id & 3 | goto 2)
0 T 2 + 4 T 2	$	reduce r2: E → T (pop T & 2 | goto 1)
0 T 2 + 4 E 5	$	reduce r1: E → E+T (pop T 2 + 4 E 5 | goto 2)
0 E 1	$	accept

SLR(1) Parser


E′ → • E
E → E+T | T
T → T*F | F
F → id
1
2
3
4
5
E → E+T | T
T → T*F | F
F → id
I0
E′ → • E
E → •E+T
E → • T
T → •T*F
T → •F
F → • id
E
T
F
id
I1
E′ → E •
E → E • +T
+
I2
E → T •
T → T • *F
*
I3
T → F •
I4
F → id •
I5
E → E + • T
T → •T*F
T → •F
F → • id
T
F
id
I6
E → E + T •
T → T • *F
*
I7
T → T * • F
F → • id
F
I8
T → T * F •
id
SR



SLR(1) Parser

	ACTION				GOTO		
	id	+	*	$	E	T	F
0	s4				1	2	3
1		s5		acc			
2	r2	r2	s7/r2	r2			
3	r4	r4	r4	r4			
4	r5	r5	r5	r5			
5	s4					7	3
6	r1	r1	s7/r1	r1			8
7	s4						
8	r3	r3	r3	r3			

LR(0) Parsing Table

SLR(1) Parser



	ACTION				GOTO		
	id	+	*	$	E	T	F
0	s4				1	2	3
1		s5		acc			
2		r2	s7	r2			
3		r4	r4	r4			
4		r5	r5	r5			
5	s4					7	3
6		r1	s7	r1			8
7	s4						
8		r3	r3	r3			

SLR(1) Parsing Table

FOLLOW(E) = {$, + } FOLLOW(T) = {$, +, * } FOLLOW(F) = { $, +, * }

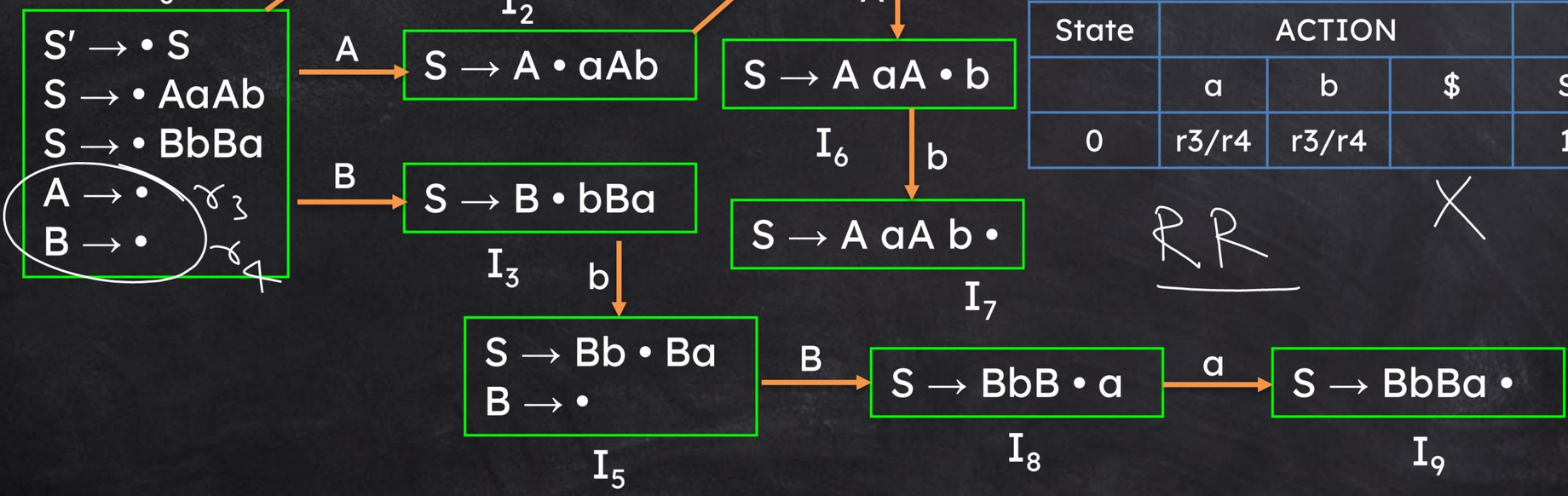
SLR(1) Parser

(1) (2)

S → AaAb | BbBa

A → ε (3)

B → ε (4)



LR(0)

State	ACTION			GOTO		
	a	b	$	S	A	B
0	r3/r4	r3/r4	r3/r4	1	2	3

SLR(1)

State	ACTION			GOTO		
	a	b	$	S	A	B
0	r3/r4	r3/r4		1	2	3

R R

X

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

CLR(1) Parser

An LR(1) item has the form:
$[A \rightarrow \alpha \bullet \beta, a]$

The Dot (•): This is a position marker. It represents the current state of parsing for this production.

Everything to the left of the dot (α) has already been matched (seen and processed from the input stream).

Everything to the right of the dot (β) is expected to be seen next.

The Lookahead (a): This is a terminal symbol (or **$**, the end-of-input marker). It's a condition for the item. The item is only "active" or "valid" if the next token in the input stream is this specific token a.

LR(1) Items

Given a set of LR(1) items I, Closure(I) is constructed as:

1. Start with I: Put all items of into Closure(I).
2. Expansion Rule:
   If $[A \rightarrow \alpha \bullet B\ \beta, a] \in$ Closure(I) and $B \rightarrow \gamma$ is a production, then add:
   $[B \rightarrow \bullet\ \gamma, b]$
   for every terminal $b \in FIRST(\beta a)$.
3. Repeat until no new items can be added.

Grammar

$S \rightarrow CC$
$C \rightarrow cC \mid d$

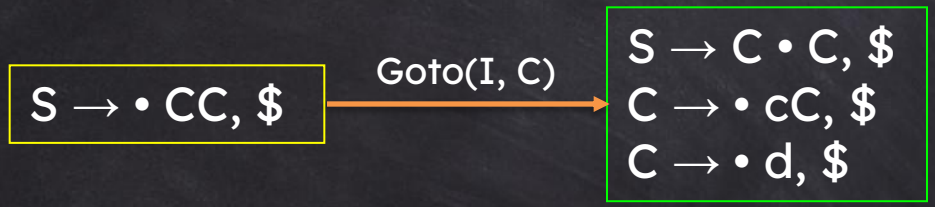
Augmented Grammar

$S' \rightarrow S$
$S \rightarrow CC$
$C \rightarrow cC \mid d$

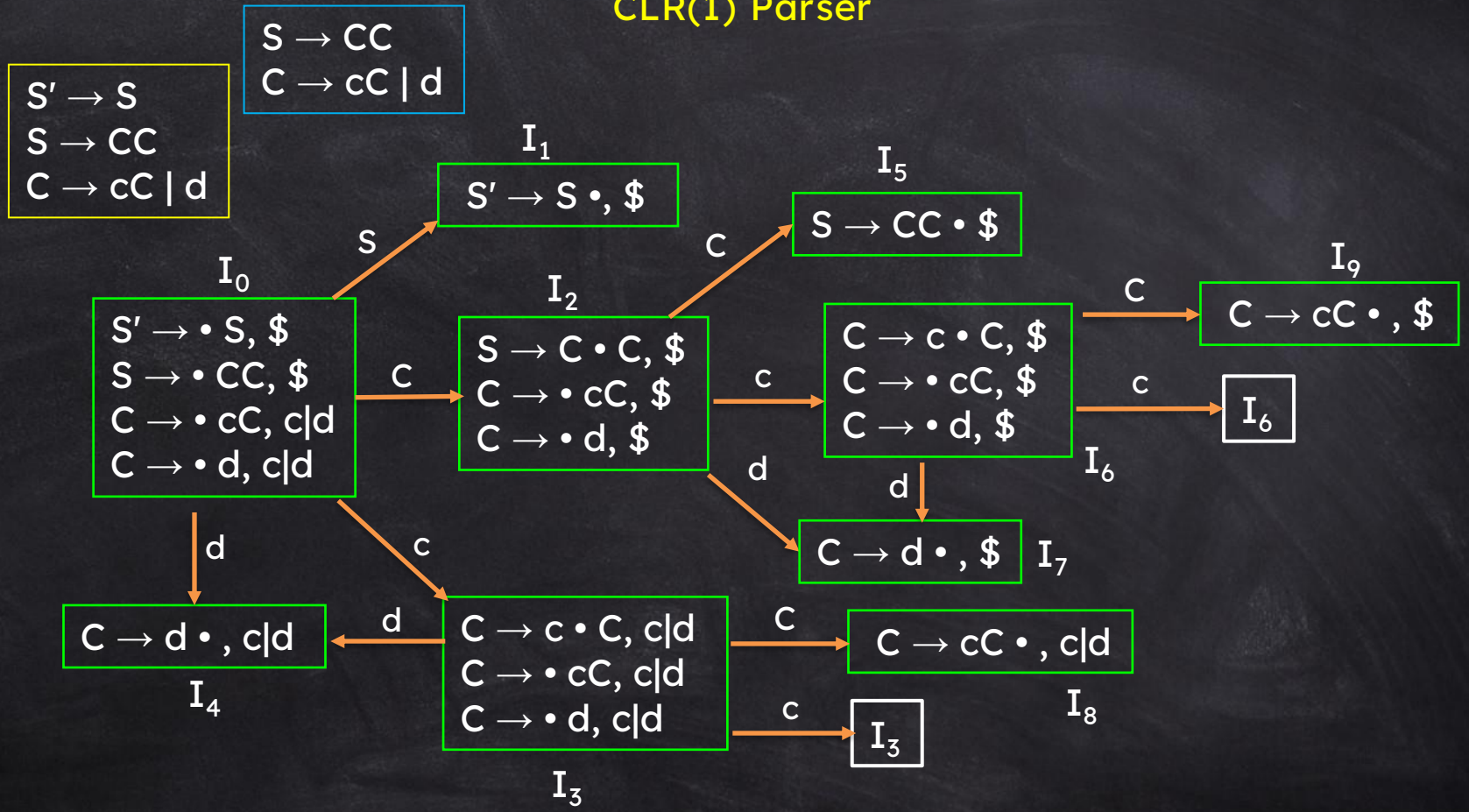
$S' \rightarrow \bullet\ S,\ \$$
$S \rightarrow \bullet\ CC,\ \$$
$C \rightarrow \bullet\ cC,\ c|d$
$C \rightarrow \bullet\ d,\ c|d$

Goto Operation: The Goto(I, X) function means:
From a set of items I
If the dot (•) is before symbol X,
Move the dot past X,
Take the Closure of the result.

$S' \rightarrow S$
$S \rightarrow CC$
$C \rightarrow cC \mid d$



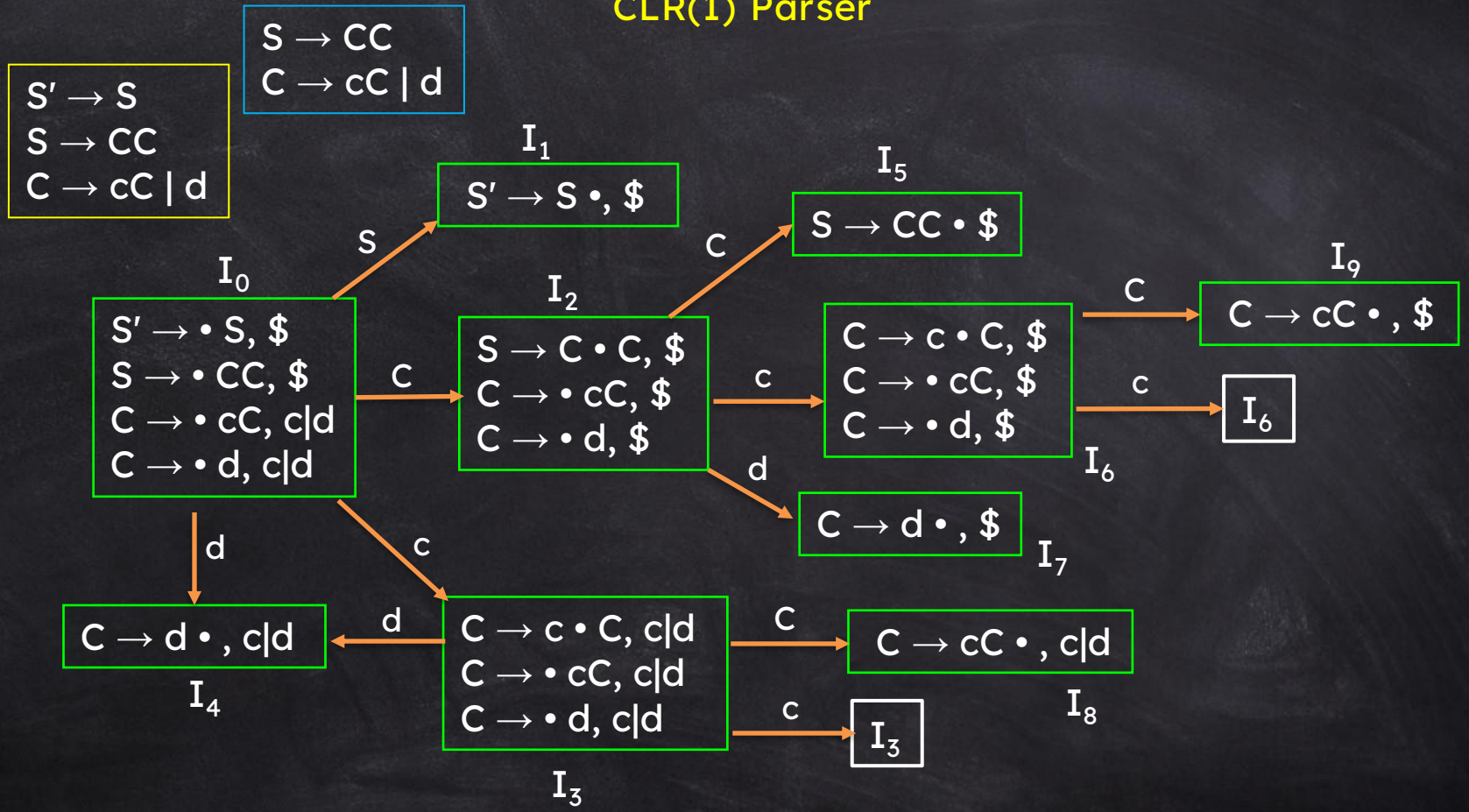
CLR(1) Parser


S′ → S
S → CC
C → cC | d
S → CC
C → cC | d
I0
S′ → • S, $
S → • CC, $
C → • cC, c|d
C → • d, c|d
I1
S′ → S •, $
I2
S → C • C, $
C → • cC, $
C → • d, $
I3
C → c • C, c|d
C → • cC, c|d
C → • d, c|d
I4
C → d • , c|d
I5
S → CC • $
I6
C → c • C, $
C → • cC, $
C → • d, $
I7
C → d • , $
I8
C → cC • , c|d
I9
C → cC • , $
S
C
c
d

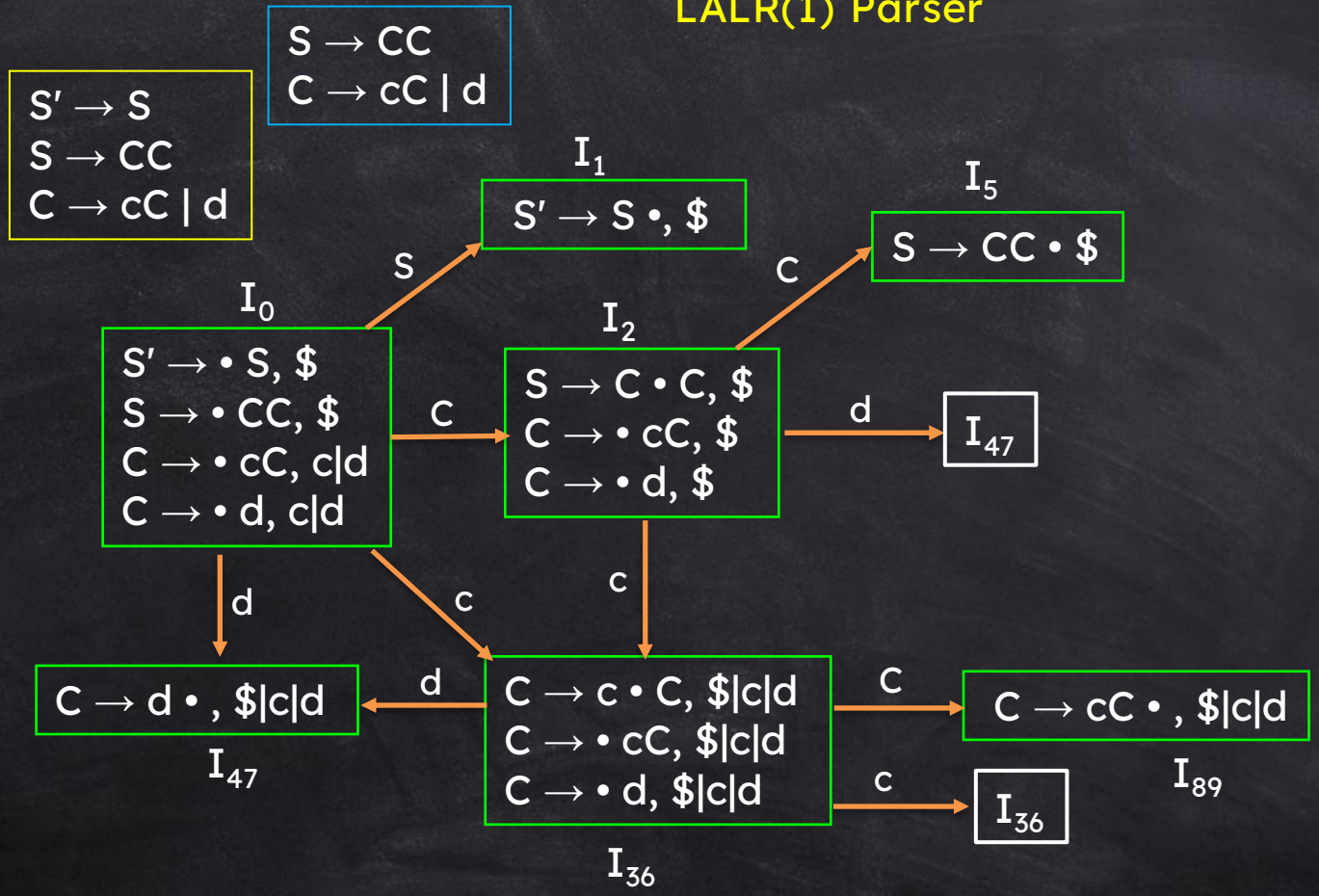
State	ACTION			GOTO	
	c	d	$	S	A
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4	r3		8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

CLR(1) Parsing Table

CLR(1) Parser


S′ → S
S → CC
C → cC | d
S → CC
C → cC | d
I0
S′ → • S, $
S → • CC, $
C → • cC, c|d
C → • d, c|d
I1
S′ → S •, $
I2
S → C • C, $
C → • cC, $
C → • d, $
I3
C → c • C, c|d
C → • cC, c|d
C → • d, c|d
I4
C → d • , c|d
I5
S → CC • $
I6
C → c • C, $
C → • cC, $
C → • d, $
I7
C → d • , $
I8
C → cC • , c|d
I9
C → cC • , $
S
C
c
d
I3
I6

LALR(1) Parser



State	ACTION			GOTO	
	a	b	$	S	A
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4	r3		8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

CLR(1) Parsing Table

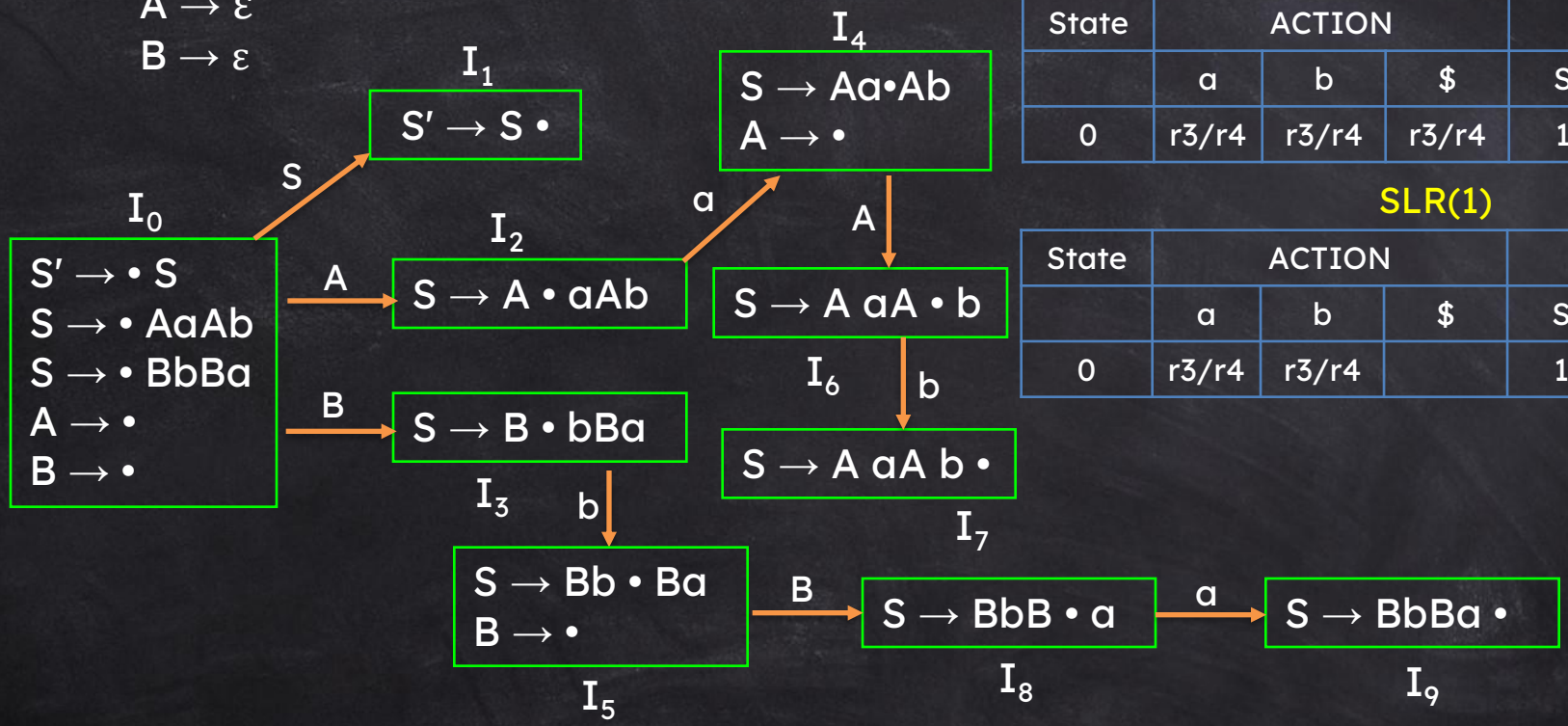
State	ACTION			GOTO	
	a	b	$	S	A
0	S36	S47		1	2
1			Accept		
2	S36	S47			5
36	S36	S47	r3		89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

LALR(1) Parsing Table

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$



LR(0)

State	ACTION			GOTO		
	a	b	$	S	A	B
0	r3/r4	r3/r4	r3/r4	1	2	3

SLR(1)

State	ACTION			GOTO		
	a	b	$	S	A	B
0	r3/r4	r3/r4		1	2	3

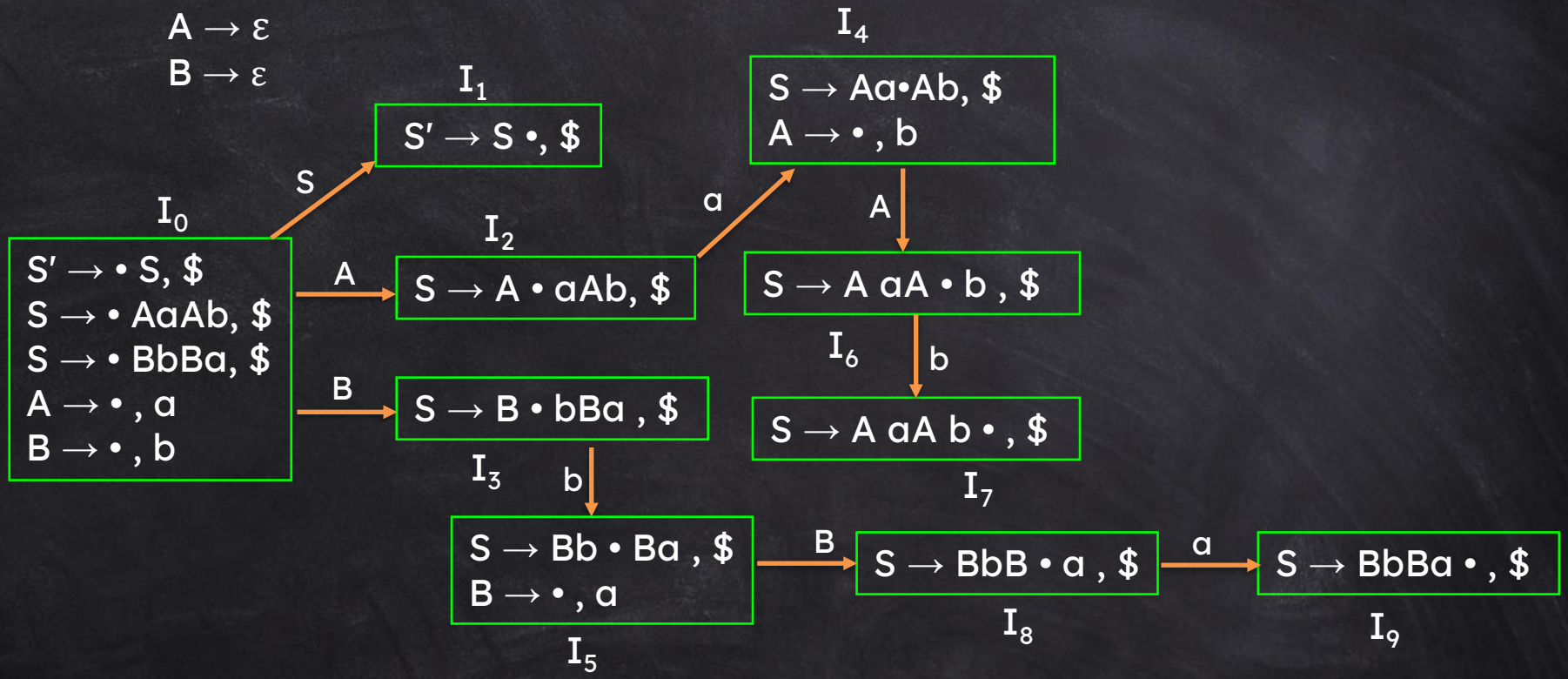
FOLLOW(S) = { $ } FOLLOW(A) = { a, b } FOLLOW(B) = { a, b }

CLR(1) Parser

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$



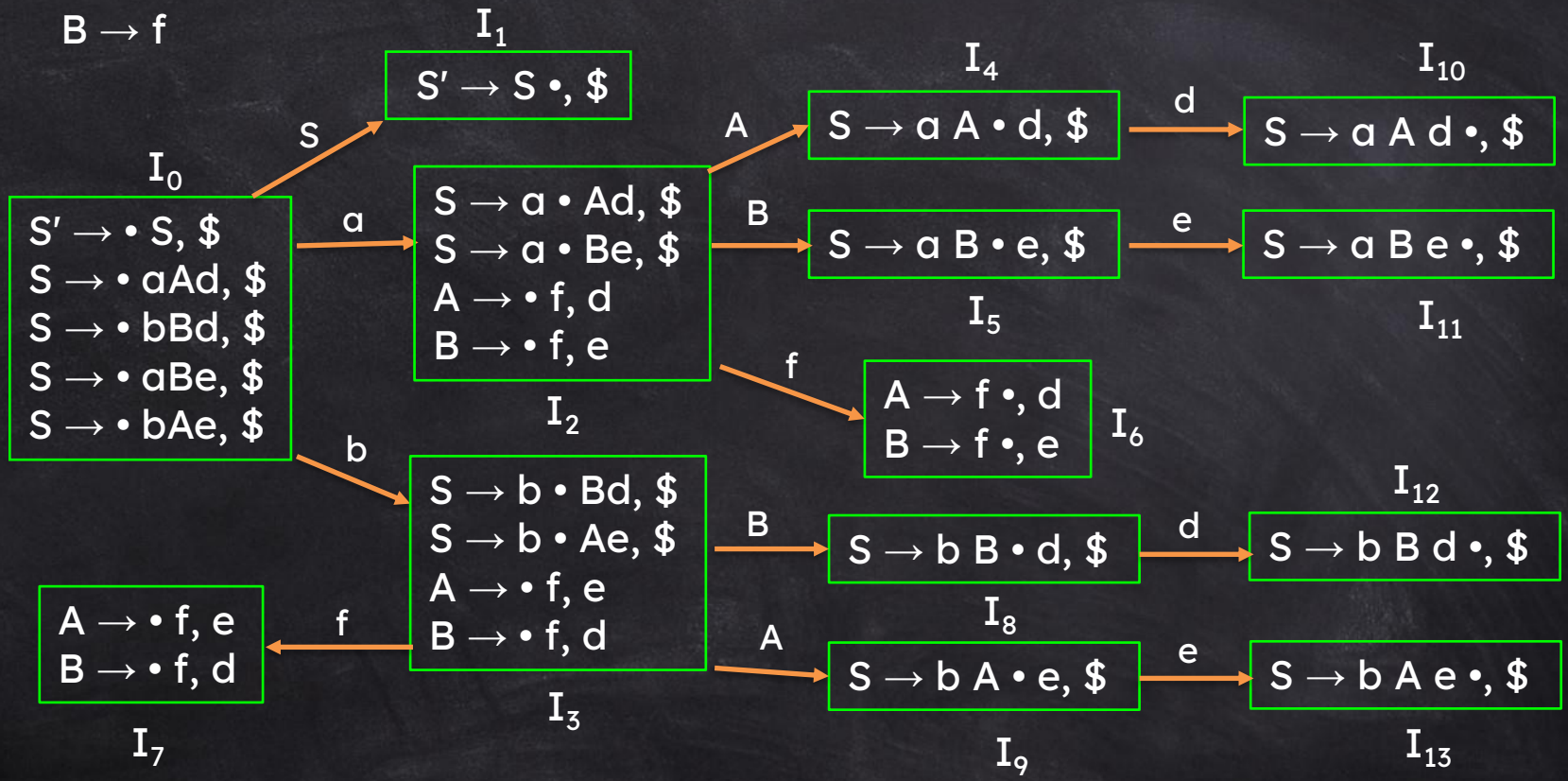
State	ACTION			GOTO		
	a	b	$	S	A	B
0	r3	r4		1	2	3
1			Accept			
2	S4					
3		S5				
4		r3			6	
5	r4					8
6	S7					
7			r1			
8	S9					
9			r2			

CLR(1) Parsing Table

LALR(1) Parsing Table

S → aAd | bBd | aBe | bAe
A → f
B → f

CLR(1) Parser



State	ACTION						GOTO		
	a	b	d	e	f	$	S	A	B
0	s2	s3					1		
1						acc			
2					s6			4	5
3					s7			9	8
4			s10						
5				s11					
6			r5	r6					
7			r6	r5					
8			s12						
9				s13					
10						r1			
11						r3			
12						r2			
13						r4			

CLR(1) Parsing Table

State	ACTION						GOTO		
	a	b	d	e	f	$	S	A	B
67			r5/r6	r6/r5					

LALR(1) Parsing Table

LR Parsing Algorithm

Let S be the state on TOS and a will be the look ahead symbol.

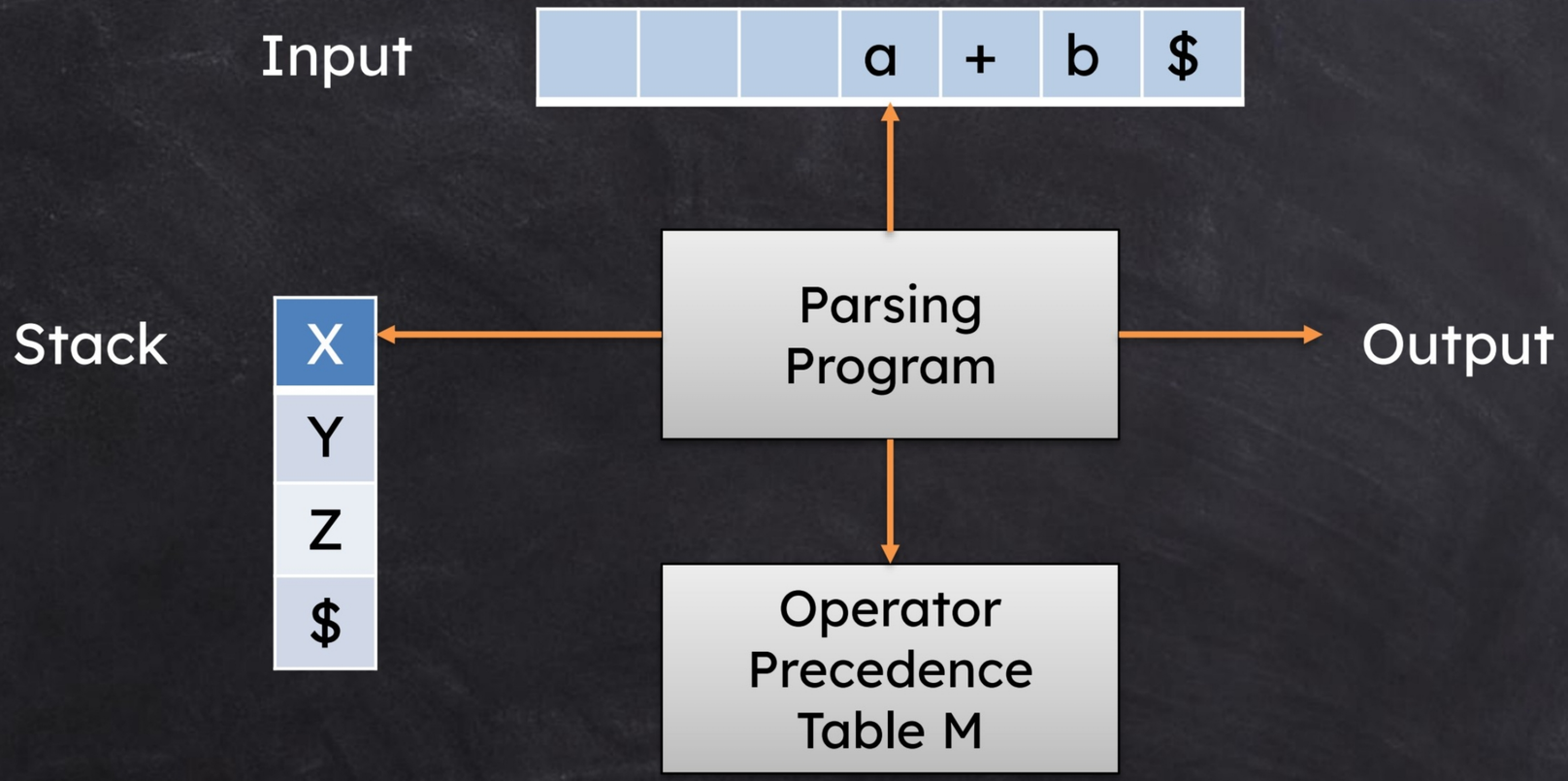
1. If ACTION[S, a] = S_i then shift(push) a into stack and increment the input pointer.

2. If ACTION[S, a] = r_j where r_j is $\alpha \rightarrow \beta$ then reduce(pop) 2 $|\beta|$ and replace by α, if m is the state below α then push goto[m, α]

3. If ACTION[S, a] = accepted, then successful completion.

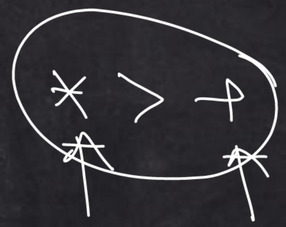
4. If ACTION[S, a] = Blank, then Error.

Operator-Precedence Parser



Operator-Precedence Parser consists of four components: an input buffer, a stack, an operator precedence table, and an output stream.
The input buffer holds the string to be parsed and appends a special end marker $ to indicate the end of the input.
The stack stores grammar symbols (terminals), with $ placed at the bottom, indicating the bottom of the stack.

The Operator Precedence Table is a two-dimensional array M[a, b]
During parsing, the program checks:
a is the symbol at the top of the stack,
b is the current input symbol.



INPUT BUFFER

STACK	id	+	*	$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	Accept

Operator Grammar

A grammar is an operator grammar if it satisfies the following condition:

For every production rule $A \rightarrow \alpha$

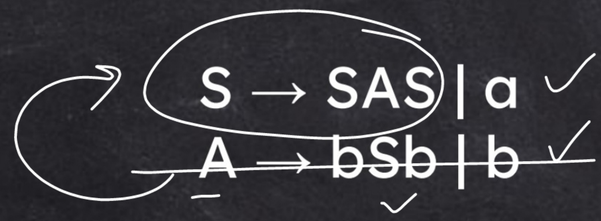
1. α must not contain consecutive non-terminals ($A \rightarrow BC$)
2. α must not contain epsilon ($A \rightarrow \varepsilon$)

Ex1: $E \rightarrow E+E \mid E*E \mid id$

Ex2:
$E \rightarrow AB$
$A \rightarrow a$
$B \rightarrow b$

Ex3:
$S \rightarrow SAS \mid a$
$A \rightarrow bSb \mid b$

Operator Grammar



S → SAS | a
A → bSb | b

S → SbSbS | SbS | a

S → SbSbS | SbS | a

E → EAE | (E) | -E | id
A → + | - | * | / | ↑

E → E+E | E-E | E*E | E/E | E↑E | (E) | -E | id

Operator Grammar

$P \rightarrow SR \mid S$
$R \rightarrow bSR \mid bS$
$S \rightarrow WbS \mid W$
$W \rightarrow L*W \mid L$
$L \rightarrow id$

$P \rightarrow SbsR$

$R \rightarrow bP$ | P

$P \rightarrow SR \mid S$
$R \rightarrow bP \mid bS$

$P \rightarrow SR$

$P \rightarrow SbP \mid SbS \mid S$
~~$R \rightarrow bP \mid bS$~~

$P \rightarrow SbP \mid SbS \mid S$
$S \rightarrow WbS \mid W$
$W \rightarrow L*W \mid L$
$L \rightarrow id$

Operator-Precedence Parsing

((a + b) + c)

(2 ↑ (3 ↑ 2))

Precedence Relations:

a < b: Operator a has lower precedence than b.

a = b: Operators have equal precedence.

a > b: Operator a has higher precedence than b.

Note 1: An identifier is always given the higher precedence than any other symbol

Note 2: $ symbol is always given the lower precedence

Note 3: When two operators have equal precedence, the order of evaluation is decided by their associativity.

If the operators are left-associative, evaluation proceeds from left to right.
If the operators are right-associative, evaluation proceeds from right to left.

Operator-Precedence Parsing

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

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

Terminal

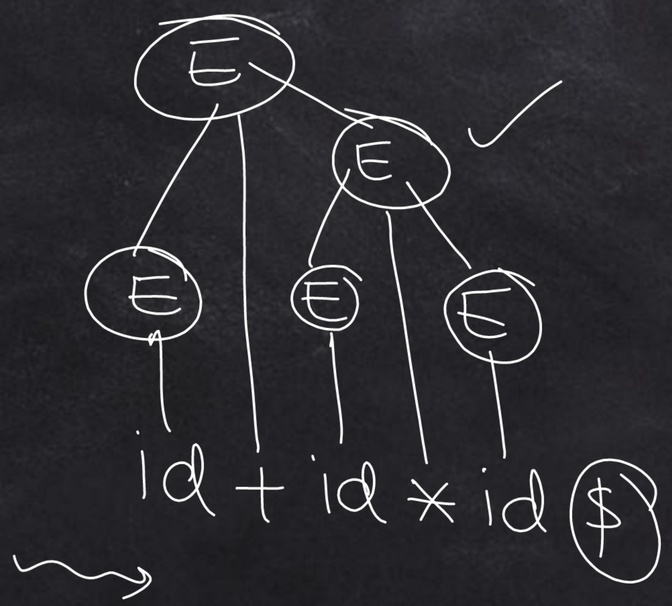
If input symbol > TOS
Action: Push (shift) the input symbol onto the stack.
If input symbol < TOS
Action: Pop (reduce) the stack until a lower-precedence symbol is found.

INPUT BUFFER

STACK

	id	+	*	$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	Accept

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



Stack	Input	Action	Relation
$	id+id*id$	push id	$ < id
$id	+id*id$	pop (reduce)	id > +
$	+id*id$	push +	$ < +
$+	id*id$	push id	+ < id
$+id	*id$	pop	id > *
$+	*id$	push *	+ < *
$+*	id$	push id	* < id
$+*id	$	pop	id > $
$+*	$	pop	* > $
$+	$	pop	+ > $
$	$	Accept	

Operator-Precedence Parsing

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid (E) \mid -E \mid E\uparrow E \mid id$

INPUT BUFFER



STACK

	+	-	*	/	↑	id	(	)	$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
↑	>	>	>	>	>	<	<	>	>
id	>	>	>	>	>	Error	Error	>	>
(	<	<	<	<	<	<	<	=	>
)	>	>	>	>	>			>	>
$	<	<	<	<	<	<	<		Accept

E → E+E | E-E | E*E | E/E | (E) | -E | E↑E | id



Stack	Input	Action	Relation
$	id+id/id$	push id	$ < id
$id	+id/id$	pop	id > +
$	+id/id$	push +	$ < +
$+	id/id$	push id	+ < id
$+id	/id$	pop	id > /
$+	/id$	push /	+ < /
$+/	id$	push id	/ < id
$+/id	$	pop	id > $
$+/	$	pop	/ > $
$+	$	pop	+ > $
$	$	Accept	

Operator-Precedence Parsing

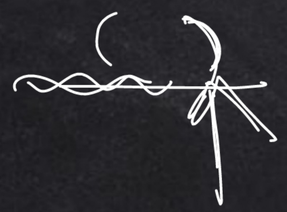


$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$



INPUT BUFFER

STACK





	+	*	(	)	id	$
+	>	<	<	>	<	>
*	>	>	<	>	<	>
(	<	<	<	=	<	
)	>	>		>		>
id	>	>		>		>
$	<	<	<		<	Accept

Operator-Precedence Parsing

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

Stack	Input	Action	Relation
$	(id+(id*id))$	push (	$ < (
$(	id+(id*id))$	push id	(< id
$(id	+(id*id))$	pop	id > +
$(	+(id*id))$	push +	(< +
$(+	(id*id))$	push (	+ < (
$(+(	id*id))$	push id	(< id
$(+(id	*id))$	Pop	id > *
$(+(	*id))$	push *	(< *
$(+(*	id))$	push id	* < id
$(+(*id	))$	pop	id >)
$(+(*	))$	pop	* >)
$(+(	))$	pop	(=)
$(+	)$	pop	+ >)
$(	)$	pop	(=)
$	$	Accept	

Operator-Precedence Parsing

P → SbP | SbS | S
S → WbS | W
W → L*W | L
L → id



INPUT BUFFER

STACK

	id	*	b	$
id	—	>	>	>
*	<	<	>	>
b	<	<	<	>
$	<	<	<	Accept

Advantages

- **Simple to implement:** Parsing can be done using a simple stack-based algorithm with precedence relations (<, >, =)
- **Efficient parsing:** Operator precedence parsing is generally faster than more complex parsers like LR(1) or SLR.
- **Handles ambiguity in expressions:** It naturally handles precedence and associativity of operators (like * before +, left-to-right for +, etc.).
- **Error detection:** Parsing errors are detected quickly when precedence relations are undefined.

Disadvantages

- **Limited class of grammars:** Only works for operator precedence grammars Cannot handle all CFGs
- **Not widely used in compilers:** Modern compilers prefer LR, SLR, LALR, LL parsers, which handle a much broader set of grammars.
- **Ambiguity in precedence relations:** If precedence relations between two terminals are not well-defined, the parser fails.

Precedence Function

In operator precedence parsing, instead of filling a full operator precedence table (<, >, =), we can assign numeric precedence values to terminals.
Two functions are defined:

- f (top-of-stack precedence function)
- g (input-symbol precedence function)

If a < b, then f (a) < g (b)
If a = b, then f (a) = g (b)
If a > b, then f (a) > g (b)

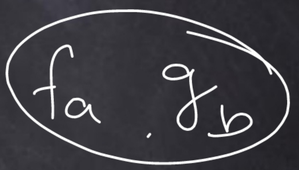
	id	+	*	$
f	4	2	4	0
g	5	1	3	0

Algorithm for Constructing Precedence Functions

For each terminal a or $, create the symbol f_a & g_a
Make a node for each symbol.



If a = b, then f_a & g_b are in same group or node.
If a = b & c = b, then f_a & f_c must be in same group or node.



(a) If a < b, Mark an edge from g_b to f_a.
(b) If a > b, Mark an edge from f_a to g_b.



If there are no cycles.



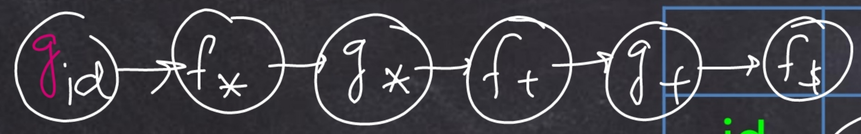
f_a = Length of longest path beginning at the group of f_a.
g_a = Length of the longest path from the group of g_a.

If the graph constructed has a cycle, then no precedence functions exist.

Precedence Function

$g_+ \leftarrow f_{id}$

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

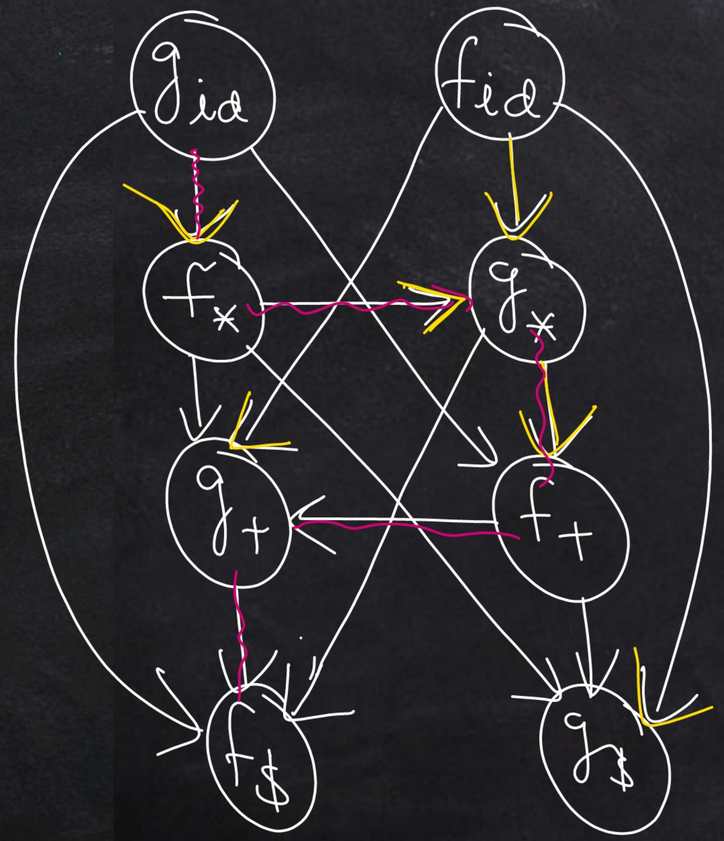


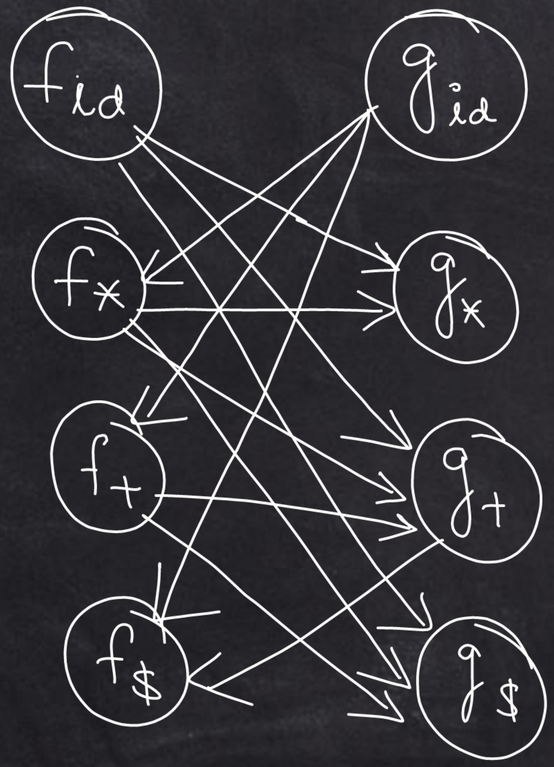
g / Input	id	+	*	$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	Accept

f

Stack

$g_+ \leftarrow f_{id}$

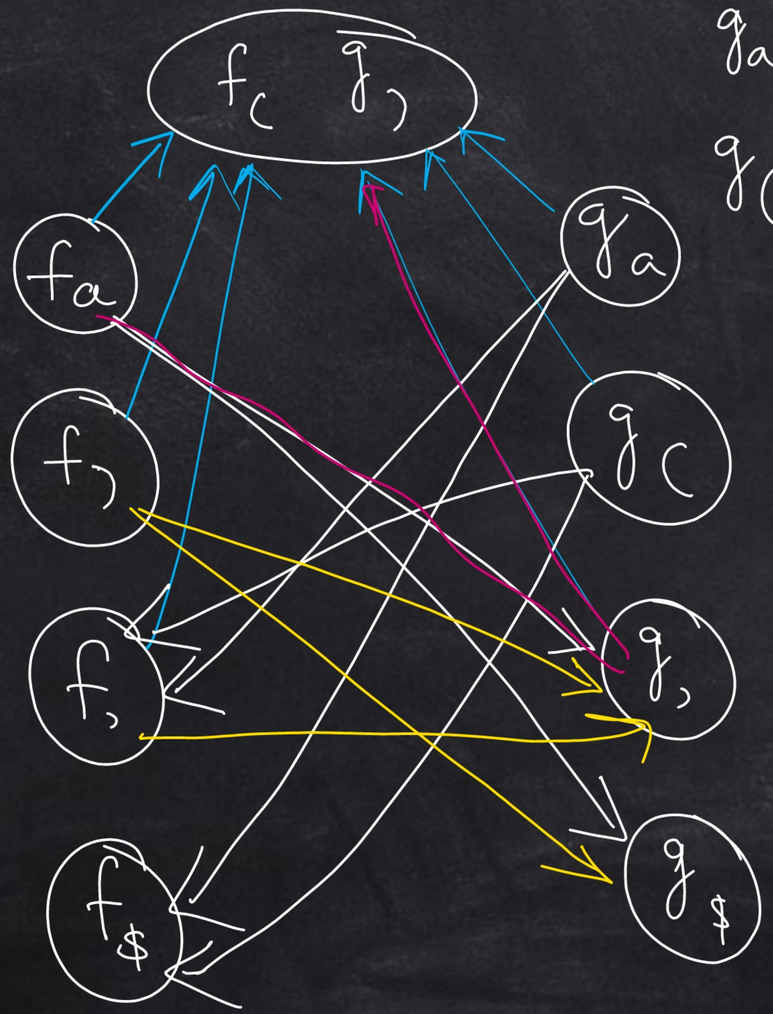




	id	+	*	$
f	4	2	4	0
g	5	1	3	0



Find the operator precedence function for the following precedence table



$g_a \rightarrow f_, \rightarrow g_,$

$g_(\rightarrow f_, \rightarrow g_,$

g

f

	a	(	)	,	$
a			>	>	>
(	<	<	=	<	
)				>	>
,	<		>		
$	<	<			

	a	(	)	,	$
f	2	0	2	2	0
g	3	3	0	1	0