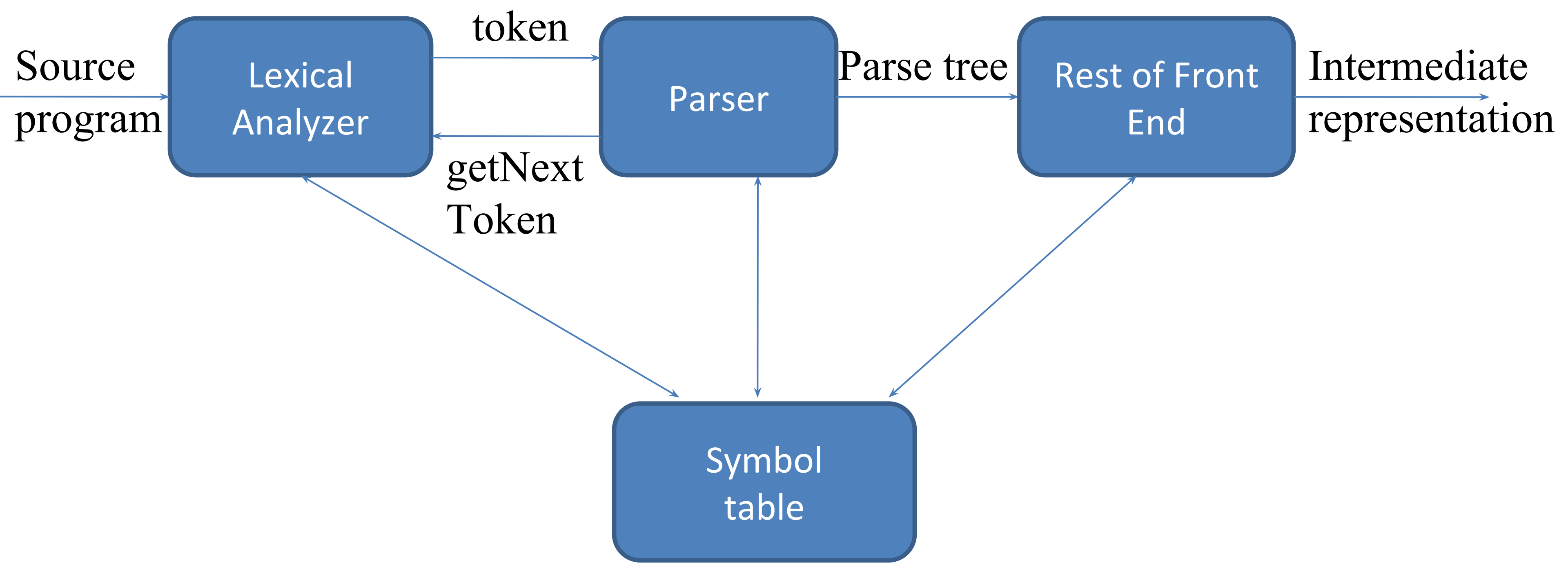
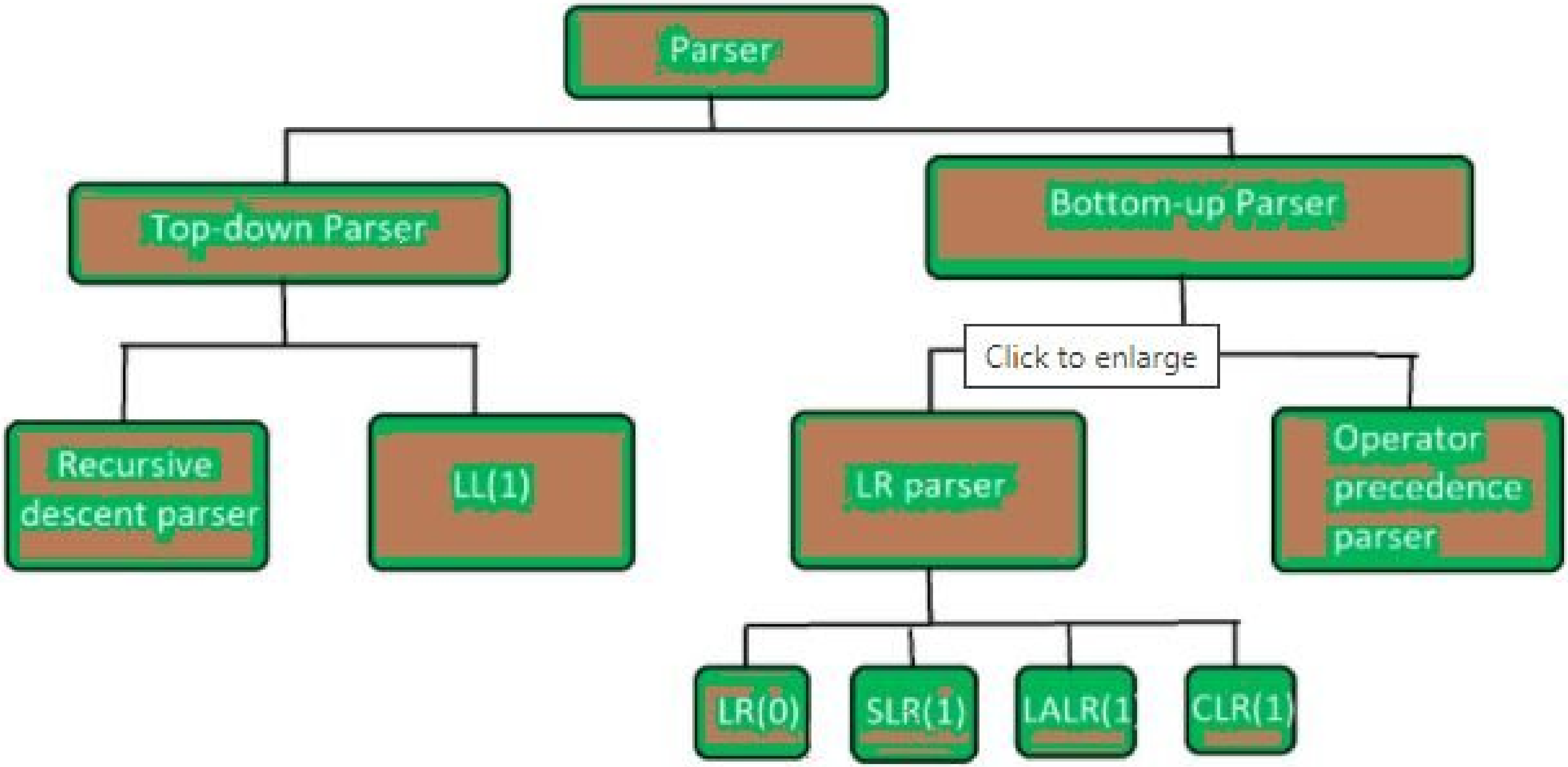
UNIT-IV Parsers

- Role of parsers, Classification of Parsers: Top down parsers- recursive descent parser and predictive parser (LL parser), Bottom up Parsers – Shift Reduce parser, LR parser.
- YACC specification and Automatic construction of Parser (YACC)

The role of parser



Types of Parsers



Parser

- Takes token string as input and with the help of existing grammar, converts it into the corresponding parse tree
- Also called as ‘Syntax Analyser’

Top down- Parser

- Parser is mainly classified into 2 types
- **Top-down** Parser, and **Bottom-up** Parser

Top-down Parser:

- It starts from the **start symbol** and ends on the **terminals**
- It uses **left most derivation**.

Top down- Parser

- Classified into 2 types
  - Recursive descent parser
  - Non-recursive descent parser

Recursive descent parser

It is also known as Brute force parser or the backtracking parser.

Non-recursive descent parser

- It is also known as **LL(1) parser** or **predictive parser** or without backtracking parser
- It uses parsing table to generate the parse tree without need of backtracking
- **Predictive parser** is a recursive descent parser – with capability to predict which production is to be used to replace the input string.
- The predictive parser does not suffer from backtracking.

Top-down Parsing

- Input string : a + b * c
- S-> E
- E->E+T
- E->E*T
- E->T
- T->id

Sample - Top-down Parsing

- S->**E**
- ->**E**+T {using 3}
- ->**E**+T*T {using 2}
- ->**T**+ T*T {using 4}
- ->id+**T***T {using 5}
- ->a+id***T** {using 5}
- -> a+b*id {using 5}
- ->a+b*c {input string}

- Input string : a * b + c

CFG Grammar Rules

(1) S-> E

(2) E->E+T

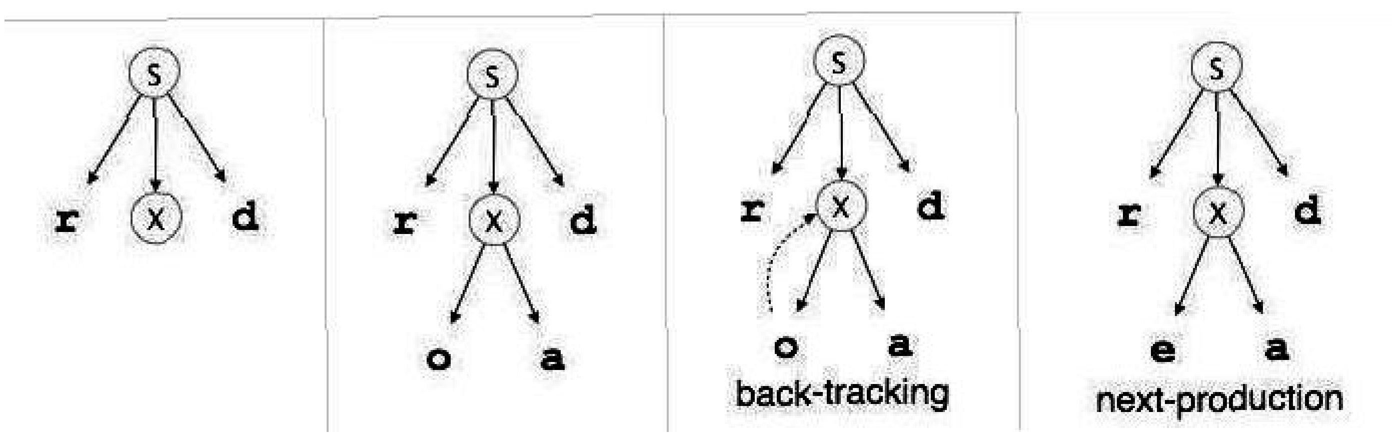
(3) E->E*T

(4) E->T

(5) T->id

Top down parsing (Backtracking)

- S->rXd | rZd
- X->oa | ea
- Z->ai
- String: read



Bottom-up Parser

- Parser which generates the parse tree for the given input string with the help of grammar productions by compressing the non-terminals
- Starts from **non-terminals** and ends on the **start symbol**
- It uses **reverse of** the **right most derivation**
- It is divided into 2 types: **LR parser**, and **Operator precedence parser**

Bottom-up Parsing

Input string : a + b * c

- S-> E
- E->E+T
- E->E*T
- E->T
- T->id

Bottom-up Parsing

- **a** + b * c {using reverse 5}
- **T**+b*c {using reverse 4}
- E+**b***c {using reverse 5}
- **E+T***c {using reverse 2}
- E***c** {using reverse 5}
- **E*T** {using reverse 3}
- E {using reverse 1}
- S

Input string : a + b * c

1) S-> E
2) E->E+T
3) E->E*T
4) E->T
5) T->id

LR parser:

LR parser is the bottom-up parser which generates the parse tree for the given string by using **unambiguous** grammar.

It follows **reverse** of **right most derivation**

LR parser is of 4 types:

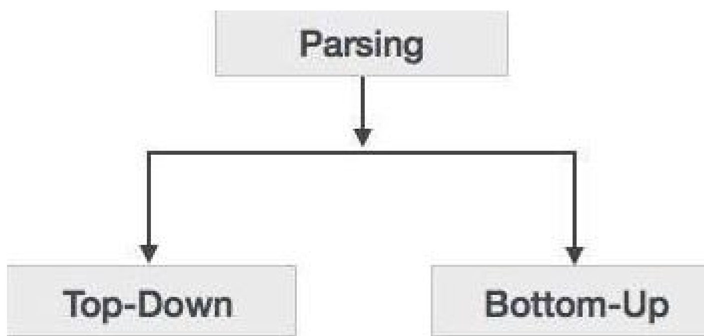
LR(0)

SLR(1)

LALR(1)

CLR(1)

Classification of Parsers



Operator precedence parser

- It generates the parse tree form given grammar and string

- The only conditions are –
  - RHS of production does not have 2 **consecutive non-terminals**
  - **RHS of production does not have epsilon**

LL(1) parsing

	FIRST	FOLLOW
S->ABCDE	FIRST(ABCDE)-FRIST(A)-{a,b ,c}	{$}
A->a|€	{a, €}	{b,c}
B->b |€	{b, €}	{c}
C->c	{c}	{d,e,$}
D-> d| €	{d, €}	{e,$}
E-> e | €	{e, €}	{$}

LL(1) parsing

	FIRST	FOLLOW
S->Bb | C d		
B-> aB | €		
C-> cC | €		

Parsing Table

	a	b	c	d	$
S	S->Bb	S->Bb	S->C d	S->C d	
B	B-> aB	B-> €			
C			C-> cC	C-> €	

It is a LL(1) grammar.

LL(1) parsing

	FIRST	FOLLOW
S->Bb | C d	{FIRST(B),FIRST(C)}-{a,b,c,d}	{$}
B-> aB | €	{a, €}	{b}
C-> cC | €	{c, €}	{d}

Parsing Table

	a	b	c	d	$
S	S->Bb	S->Bb	S->C d	S->C d	
B	B-> aB	B-> €			
C			C-> cC	C-> €	

It is a LL(1) grammar.

LL(1) parsing

	FIRST	FOLLOW
E-> TE’		
E’->+TE’ | €		
T-> FT’		
T’->* FT’ | €		
F-> id | (E)		

LL(1) parsing

	FIRST	FOLLOW
E-> TE’	{FIRST (T)}—{ id, (}	{ $,) }
E’->+TE’ | €	{+, €}	{$,) }
T-> FT’	{FIRST(F)}--{ id, (}	{+, $,) }
T’->* FT’ | €	{*, €}	{+, $,) }
F-> id | (E)	{ id , (}	{*, +, $,) }

Parsing Table

	Id	(	)	+	*	$
E						
E’						
T						
T’						
F						

It is a LL(1) grammar.

Parsing Table

	Id	(	)	+	*	$
E	E-> TE’	E-> TE’				
E’			E’-> €	E’->+TE’		E’-> €
T	T-> FT’	T-> FT’				
T’			T’-> €	T’-> €	T’->* FT’	T’-> €
F	F-> id	F-> (E)				

It is a LL(1) grammar.

	FIRST	FOLLOW
S->ACB|CbB| Ba		
A->da| BC		
B->g | €		
C->h |€		

	FIRST	FOLLOW
S->ACB|CbB| Ba	{FIRST(A),FIRST(C), FIRST(B)}-{d, g, h, €, b, a}	{$}
A->da| BC	{d,FIRST(B)}-{d, g, h, € }	{h,g,$}
B->g | €	{g , € }	{$, a , h, g,}
C->h |€	{h , €}	{g, $, b, h}

	FIRST	FOLLOW
S->aABb		
A->C| €		
B->d| €		

- S->aBDh
- B->cC
- C->bC| €
- D->EF
- E->g| €
- F-> f | €

- S->AaAb | BbBa
- A-> €
- B-> €

Find whether the grammar is LL(1) or not?

- S-> aSbS | bSaS | €

Find whether the grammar is LL(1) or not?

- S->aABb
- A->c| €
- B->d | €

Find whether the grammar is LL(1) or not?

- S->A | a
- A-> a

Find whether the grammar is LL(1) or not?

- S->aB| €
- B->bC | €
- C-> cS | €

Find whether the grammar is LL(1) or not?

- S-AB
- A->a | €
- B-> b | €

- S->((S)) | €   (( ))$

Role of Parser in Compiler Design

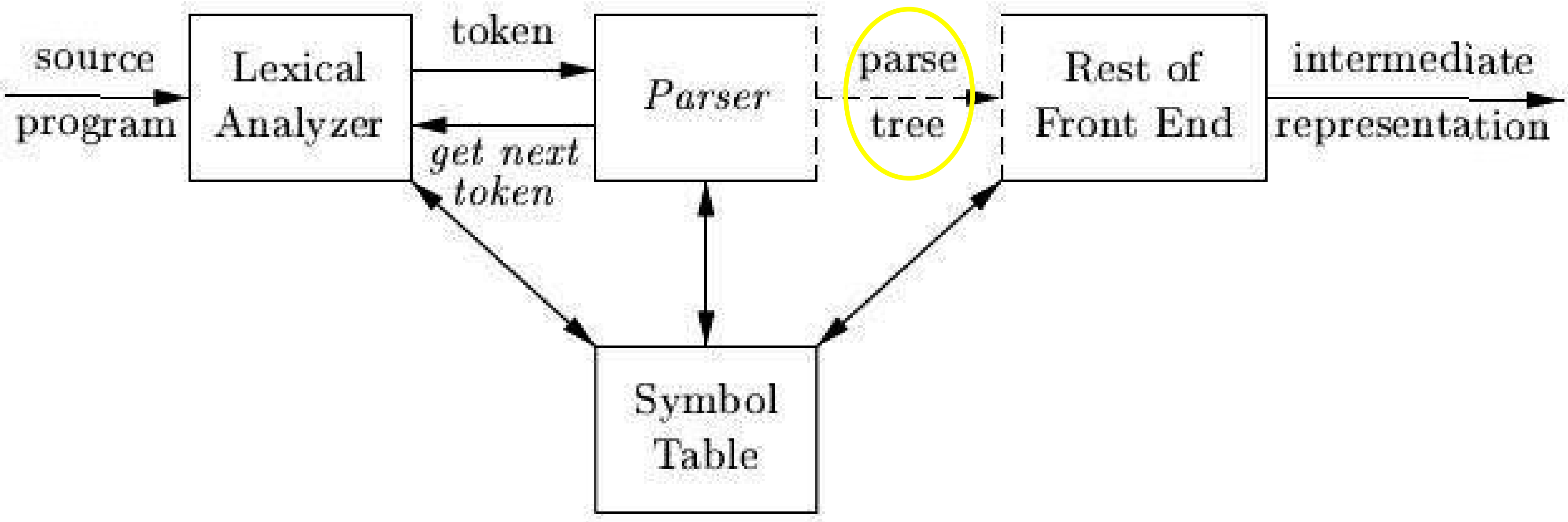


Figure : Position of parser in compiler model

Tasks of Parser

- **Obtain** string of Tokens from Lexical Analyzer
- **Group** the Tokens appearing in the input order, to identify larger structures in the program
- **Identify and report syntax errors** in the program
- **Recover from error**, and continue to process the rest of the input

Comparison with Lexical Analysis

Phase	Input	Output
Lexer	String of characters	String of tokens
Parser	String of tokens	Parse tree

Grammars and HLL

- Grammar gives a precise, yet easy-to-understand, **syntactic** specification of a programming language (HLL)
- From **certain classes** of grammars, an efficient parser can automatically be generated that determines the **syntactic structure** of a source program
- The structure imparted to a language by **a properly designed grammar** is useful for translating source programs into correct object code and for detecting errors

Context Free Grammars (CFG)

- The **syntax of a programming language (HLL)** is described by a context free grammar (CFG) in BNF (Backus-Naur Form) notation

- CFG consists of **set of terminals**, **set of non terminals**, a **start symbol** and **set of productions**
  - **It has only one non-terminal symabolin the LHS of the production rule**

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

Sample Grammar

- Grammars with keywords are easier to parse, so (for study purpose) we look at **Grammars for EXPRESSIONS**, which present more of challenge, because of the associativity and precedence
- LR Grammar suitable for Bottom –up Parser (Not Suitable for Top-Down Parser, because of left recursion)

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

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

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

Derivation

- Meaning - Starting from the non-terminal start symbol of grammar, replacing stepwise, each nonterminal with the right-hand–side(RHS) of the rule, till no non-terminal symbol remains

Grammar, Derivation and Sentence

- The language generated by a grammar is its set of **sentences**
- A string of terminals **w** is in L(G), the language generated by G, if and only if **w** is a sentence of G
- **Sample Grammar G :**

$$\boxed{\begin{array}{lll} E & \rightarrow & E + T \mid E - T \mid T \\ T & \rightarrow & T * F \mid T / F \mid F \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}}$$

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

- **Derivation :** $\boxed{\text{"}E \text{ derives } -E.\text{"}}$ $\boxed{E * E \Rightarrow (E) * E \text{ or } E * E \Rightarrow E * (E).}$

derivation of $-(\mathbf{id})$ from E $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$

- **Sentence :**

The string $-(\mathbf{id} + \mathbf{id})$ is a sentence of grammar because there is a derivation

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\mathbf{id} + E) \Rightarrow -(\mathbf{id} + \mathbf{id})$$

Derivations, Sentence and Sentential form

$\alpha A\beta$, where α and β are arbitrary strings of grammar symbols

Suppose $A \rightarrow \gamma$ is a production.

The symbol $\Rightarrow$ means, "derives in one step."

$$\alpha A\beta \Rightarrow \alpha\gamma\beta$$

When a sequence of derivation steps $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$ rewrites α_1 to α_n, we say α_1 *derives* α_n

$\stackrel{+}{\Rightarrow}$ means, "derives in one or more steps."

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

- A sentence of G is a sentential form with **no nonterminals**
- A sentential form may contain both **terminals and nonterminals,** and may be empty.

Operand Associativity

- *[Apply to same operator]*
  - When an operand has operators to its left and right, conventions are needed for deciding **which operator applies to that operand**.
  - In most programming languages the four arithmetic operators, addition, subtraction, multiplication, and division are **left-associative**
    - By convention, 9+5+2 is equivalent to (9+5)+2 and 9-5-2 is equivalent to (9-5)-2.
  - Right Associative Operators
    - Exponentiation a**B**C -> a**(B**C)
    - Assignment operator = in C and its descendants is right-associative; that is, the expression a=b=c is treated in the same way as the expression a=(b=c).

Precedence of Operators

- Rules defining **the relative precedence** of operators are needed when **more than one kind** of operator is present.
- Sample expression : 9+5*2
  - Two possible Interpretations: (9+5)*2 or 9+(5*2)
  - In ordinary arithmetic, **multiplication and division** have **higher precedence** than addition and subtraction, so precedence leads to only one Interpretations: (9+5)*2 or **9+(5*2)** i.e. **9+(5*2)**

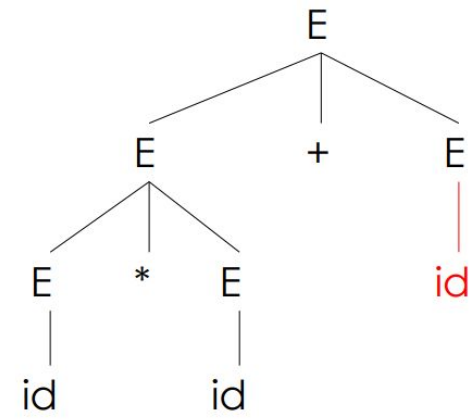
Terms and Concepts required:

2) Leftmost, Rightmost Derivation

Leftmost Derivation

- At each step replace each leftmost non-terminal

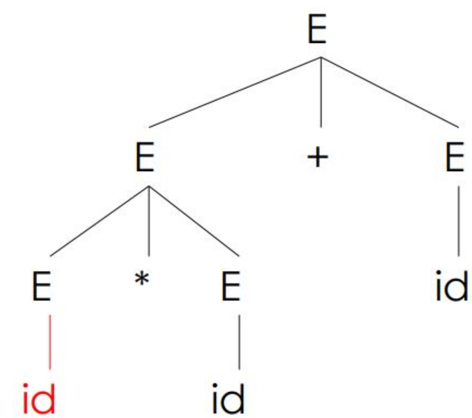
$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E*E+E \\
\rightarrow\ & id*E+E \\
\rightarrow\ & id*id+E \\
\rightarrow\ & id*id+id
\end{aligned}
$$



Rightmost Derivation

- At each step replace each rightmost non-terminal

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



Note that right-most and left-most derivations have the same parse tree

- The difference is the order in which branches are added

Leftmost / Rightmost Derivation

Set of production rules in a CFG over an alphabet {a}

- X → X+X | X*X |X| a

Leftmost derivation for the string **"a+a*a“**

X → X+X → a+X → a + X*X → a+a*X → a+a*a

- Rightmost derivation for the string **"a+a*a“**

- X → X*X → X*a → X+X*a → X+a*a → a+a*a

Q) Given the grammar and a string, construct a leftmost and rightmost derivation for it

S-> (L) |a

L->L,S |S

NOTE: 2 Nonterminals are : S, L

4 Terminals are : (,), a, ','

Terms and Concepts required:
3) Handle of a String

Handle of a string

- A handle of a string is a **substring** that matches the **right side (RHS)** of a **grammar production rule**
- **Its reduction**, to the LHS nonterminal of the grammar production rule **represents one step along** the **reverse of a rightmost derivation**
- A handle of a right – sentential form γ is a production A→β and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ. That is , if S =>αAw =>αβw,then A→β in the position following α is a handle of αβw.

Sample Handle – **bc**

- Assume a rightmost derivation (S is the start symbol)

  S ->* aXw ->a**bc**w

Then a production in grammar, say **X -> bc** in the **position after a** is a handle of abcw

Recall: A handle is a string that can be reduced and also allows further reductions back to the start symbol (using a particular production at a specific spot)

Example: Considerthe following grammar and show the handle of each right sentential form of string (a,(a,a))

$S \rightarrow (L) \mid a$

$L \rightarrow L,S \mid S$

To find the rightmost derivation of given string (a,(a,a))

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

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

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

Example: Considerthe following grammar and show the <u>handle</u> of each right sentential form of string $(a,(a,a))$

$S \rightarrow (L) \mid a$

$L \rightarrow L,S \mid S$

To find the rightmost derivation of given string $(a,(a,a))$

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

Sentential Form Handles

Sentential form	Handle
$(a,(a,a))$	$S \rightarrow a$ (at position preceding first comma)
$(S,(a,a))$	$L \rightarrow S$ (at position preceding first comma)
$(L,(a,a))$	$S \rightarrow a$ (preseding secondcomma)
$(L,(S,a))$	$L \rightarrow S$ (")
$(L,(L,a))$	$S \rightarrow a$ (following 2nd comma)
$(L,(L,S))$	$L \rightarrow L,S$ (position following second left bracket)
$(L,(L))$	$S \rightarrow (L)$ (position following first comma)
(L,S)	$L \rightarrow L,S$ (position following first leftbracket)
(L)	$S \rightarrow (L)$ (position before end marker)

Terms and Concepts required :

4) Left recursion

Left recursive Grammars

How to eliminate left recursion

Left recursive grammar

- If a CFG has a production in the form

  E → Ea ; where **E** is a non-terminal and **'a'** is a string, it is called a **left recursive production**
- Grammar having a left recursive production is called a **Left recursive grammar**

Eliminating Left Recursion

Consider the Sample left-recursive grammar

S -> S a | b

S generates all strings starting with a b and followed by a number of a

Can rewrite using right-recursion

S -> b S'

S' -> a S' | ε

..contd..Left Recursion

- In general, let

  $S \rightarrow S\,\alpha_1 \mid \dots \mid S\,\alpha_n \mid \beta_1 \mid \dots \mid \beta_m$
- All strings derived from S start with one of $\beta_1, \dots, \beta_m$ and continue with several instances of $\alpha_1, \dots, \alpha_n$

Rewrite as

$S \rightarrow \beta_1 S' \mid \dots \mid \beta_m S'$

$S' \rightarrow \alpha_1 S' \mid \dots \mid \alpha_n S' \mid \varepsilon$

General Left Recursion: Sample

The grammar

$S \to A\alpha \mid \delta$

$A \to S\beta$

is also left-recursive because

$S \to^{+} S\beta\alpha$

Left Recursion Elimination

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

$$\begin{array}{lcl} E & \rightarrow & E + T \mid E - T \mid T \\ T & \rightarrow & T * F \mid T \,/\, F \mid F \\ F & \rightarrow & (\, E \,) \mid \mathbf{id} \end{array}$$

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

E-> E+T |T

E->TE’

E’->+TE’|ϵ

E->E-T

T->T*F

T->T/F

T->F

F->(E)

F->**id**

Q) Find out if the following 3 Grammar have left recursion and make appropriate transformation to eliminate it

1) $A \to A\alpha \mid \beta$

2) $E \to T+E \mid T-E \mid T$

   $T \to id \mid num$

3) $E \to T+E \mid T-E \mid T$

   $T \to T*E \mid T/E \mid F$

   $F \to id$

- $A \rightarrow ABd$ / Aa / a
- $B \rightarrow Be$ / b

- Solution:
- $A \rightarrow aA'$
- $A' \rightarrow BdA' / aA' / \in$
- $B \rightarrow bB'$
- $B' \rightarrow eB' / \in$

Terms and Concepts required:

5) Left Factoring a Grammar

Left factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing. When the choice between two alternative productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.
- For example, if we have the two productions

stmt -> if expr then stmt else stmt

-> if expr then stmt

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

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

Left factoring

Sample Grammar:

- E -> T + E | T
- T -> int | int * T | (E)

- Hard to predict because for T two productions start with **int** ,
- For E it is not clear how to predict

Factor out common prefixes of productions
E ->T X
X -> + E | ε
T-> int Y | (E)
Y-> * T | ε

Q) Perform Left factoring on the following Grammar

E -> T + E | T

T -> int | int * T | (E)

Terms and Concepts required:
6) Some Parsing Terms

Parse Trees, Ambiguous Grammar

Definition : Parse Tree

- A **parse tree / parsing tree/derivation tree / syntax tree** is an ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar.

Construction of a Parse Tree

- Made precise by taking a **derivational view** *(i.e. beginning with the start symbol, replace a nonterminal by the body (RHS) of one of its productions)*
  - This derivational view is- Top-Down construction of a parse tree
- Bottom-up generation of parse tree is also possible, where parsing starts from input string and aims to reach the grammar start symbol using **REDUCTIONS at Handles**
  - Uses class of derivations known as “rightmost" derivations, in which the rightmost nonterminal is rewritten at each step

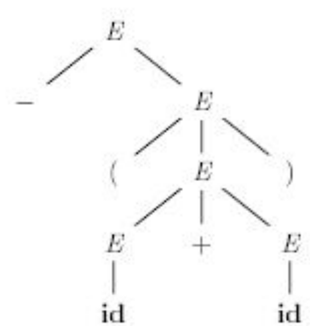
Parsing and Parse Trees

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid E - T \mid T \\
T & \rightarrow & T * F \mid T \,/\, F \mid F \\
F & \rightarrow & (\, E \,) \mid \mathbf{id}
\end{array}
$$

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

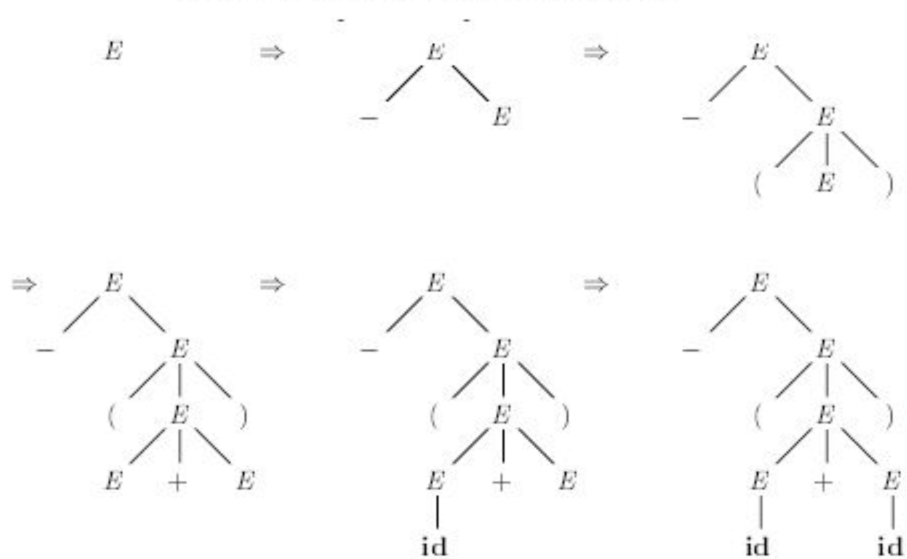
The string $-(\mathbf{id} + \mathbf{id})$ is a sentence of grammar because there is a derivation

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$$



Parse tree for $-(\mathbf{id} + \mathbf{id})$

Sequence of parse trees for derivation



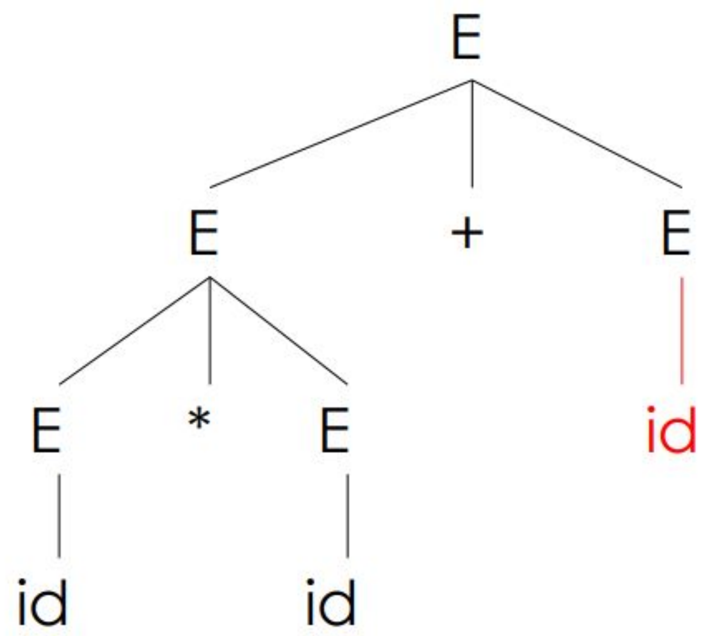
Grammar for a Simple Arithmetic Expression

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

Input String : **id*id+id**

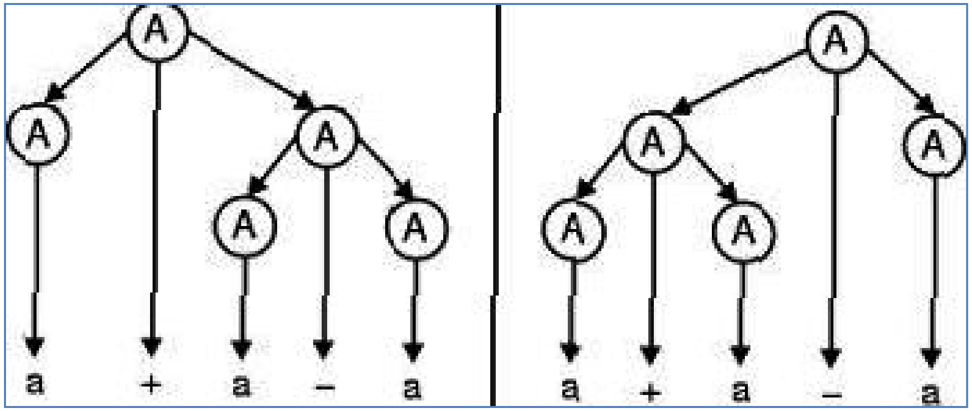
The **parse tree shows** the **association** of operations, the input string does not.

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



Ambiguous Grammar

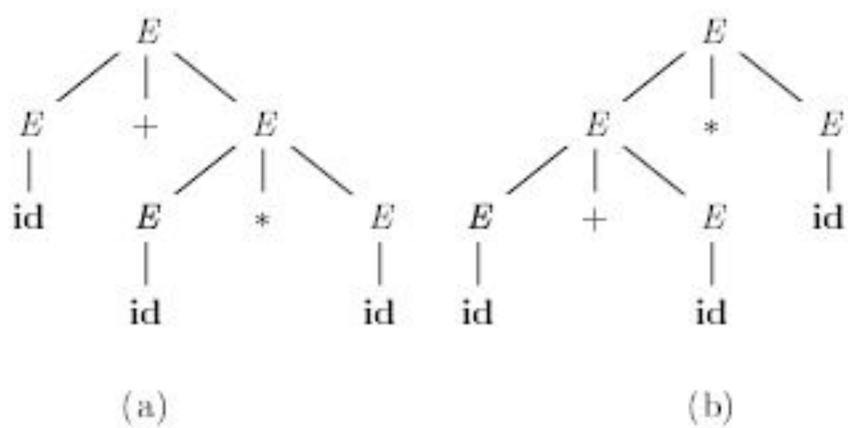
- Grammar can lead to **two or more parse trees**



Ambiguity

The arithmetic expression grammar permits two distinct leftmost derivations for the sentence $\mathbf{id} + \mathbf{id} * \mathbf{id}$:

$$
\begin{array}{lll}
E & \Rightarrow & E + E \\
 & \Rightarrow & \mathbf{id} + E \\
 & \Rightarrow & \mathbf{id} + E * E \\
 & \Rightarrow & \mathbf{id} + \mathbf{id} * E \\
 & \Rightarrow & \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{array}
\qquad
\begin{array}{lll}
E & \Rightarrow & E * E \\
 & \Rightarrow & E + E * E \\
 & \Rightarrow & \mathbf{id} + E * E \\
 & \Rightarrow & \mathbf{id} + \mathbf{id} * E \\
 & \Rightarrow & \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{array}
$$



Two parse trees for **id+id*id**

Q)

1) What is a Parse Tree?
2) What is meant by Ambiguous Grammar?

More on Top Down Parsers

- Simple implementations of top-down parsing cannot accommodate direct and indirect **left-recursion**
  - Exponential time and space complexity while parsing ambiguous CFGs
- Examples : LL parsers, Recursive-descent parser, cannot accommodate left recursive production rules

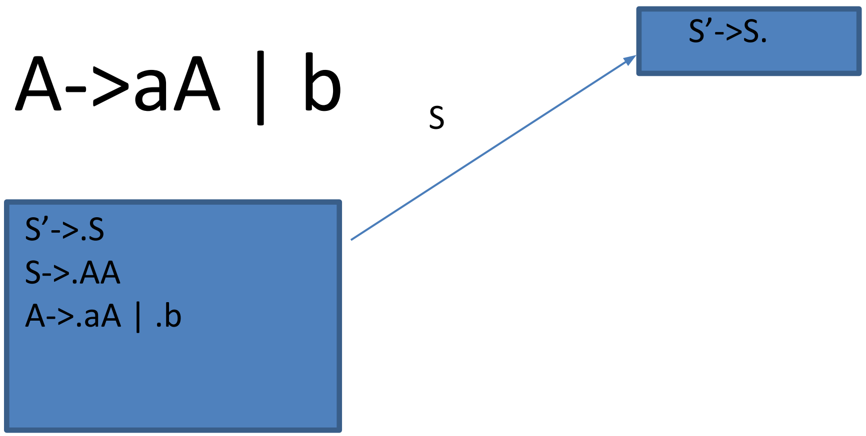
*(Recent **Sophisticated Algorithms** for top-down parsing accommodate ambiguity and left recursion in polynomial time (PT))*

- Bottom UP PARSER

- LR(0): The LR parser is an efficient bottom-up syntax analysis technique that can be used for a large class of context-free grammar.
- This technique is also called LR(0) parsing
- L stands for the left to right scanning
- R stands for rightmost derivation in reverse
- 0 stands for no. of input symbols of lookahead.

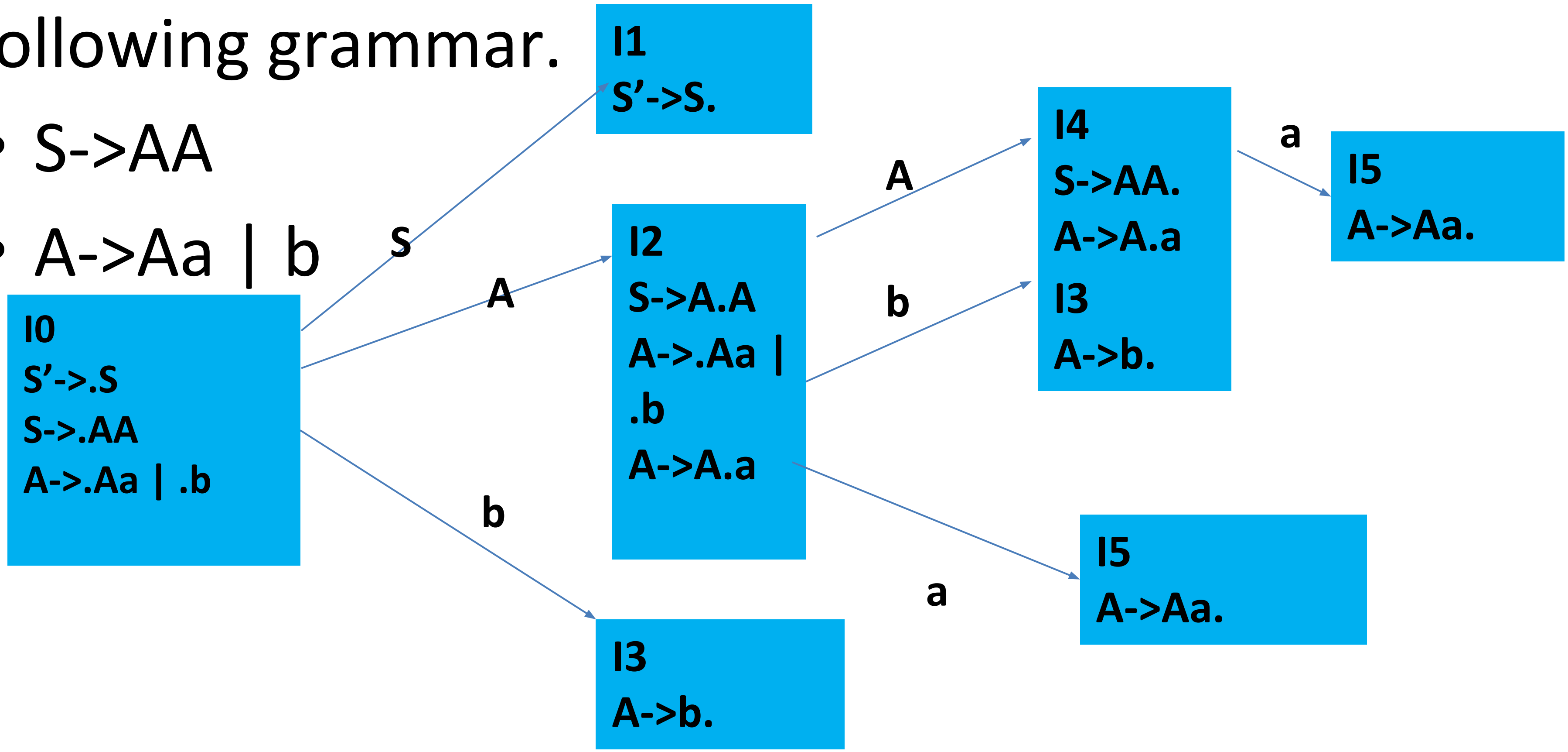
- **Augmented grammar :**
  If G is a grammar with starting symbol S, then G’ (augmented grammar for G) is a grammar with a new starting symbol S‘ and productions S-> .S‘ .
- The purpose of this new starting production is to indicate to the parser when it should stop parsing. The ‘ . ‘ before S indicates the left side of ‘ . ‘ has been read by a compiler and the right side of ‘ . ‘ is yet to be read by a compiler.

- S->AA
- A->aA | b



Example 1: Construct LR(0) Parsing table for the following grammar.

- S->AA
- A->Aa | b



Steps to be performed for LR(0) parsing table

1. Add Augment production in the given grammar
2. Create Canonical collection of LR (0) items
3. Draw a data flow diagram (DFA)
4. Construct a LR(0) parsing table-Defining 2 functions: goto(list of non-terminals) and action(list of terminals) in the parsing table.

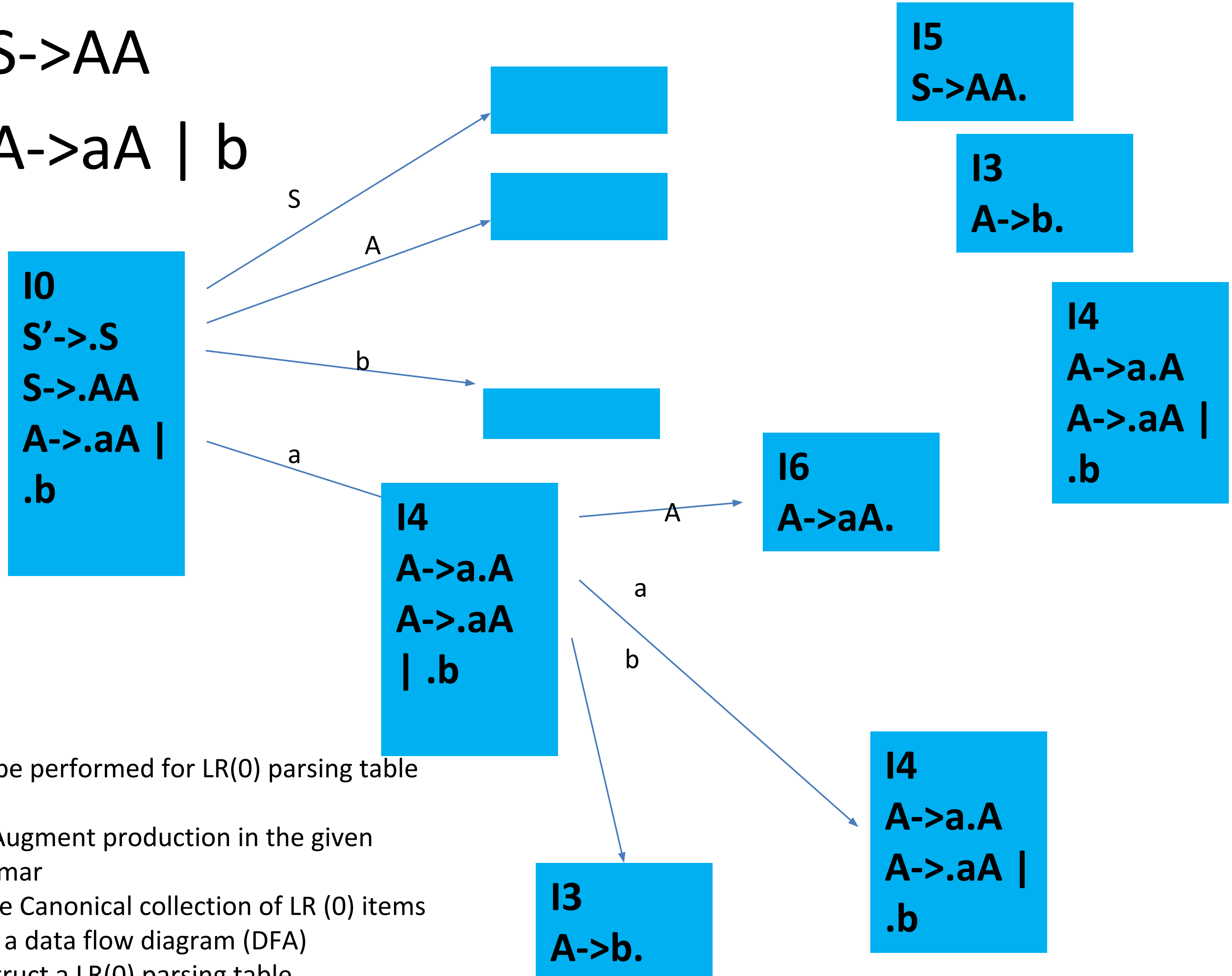
Example 1 conti...
LR(0) Parsing Table

STATES	ACTION			GOTO	
	a	b	$	S	A
I0		S3		1	2
I1			ACCEPT		
I2	S5	S3			4
I3	r3	r3	r3		
I4	S5, r1	r1	r1		
I5	r2	r2	r2		

It is shift reduce conflict, so
it is not a LR(0).

Example 2: Construct LR(0) Parsing table

- S->AA
- A->aA | b

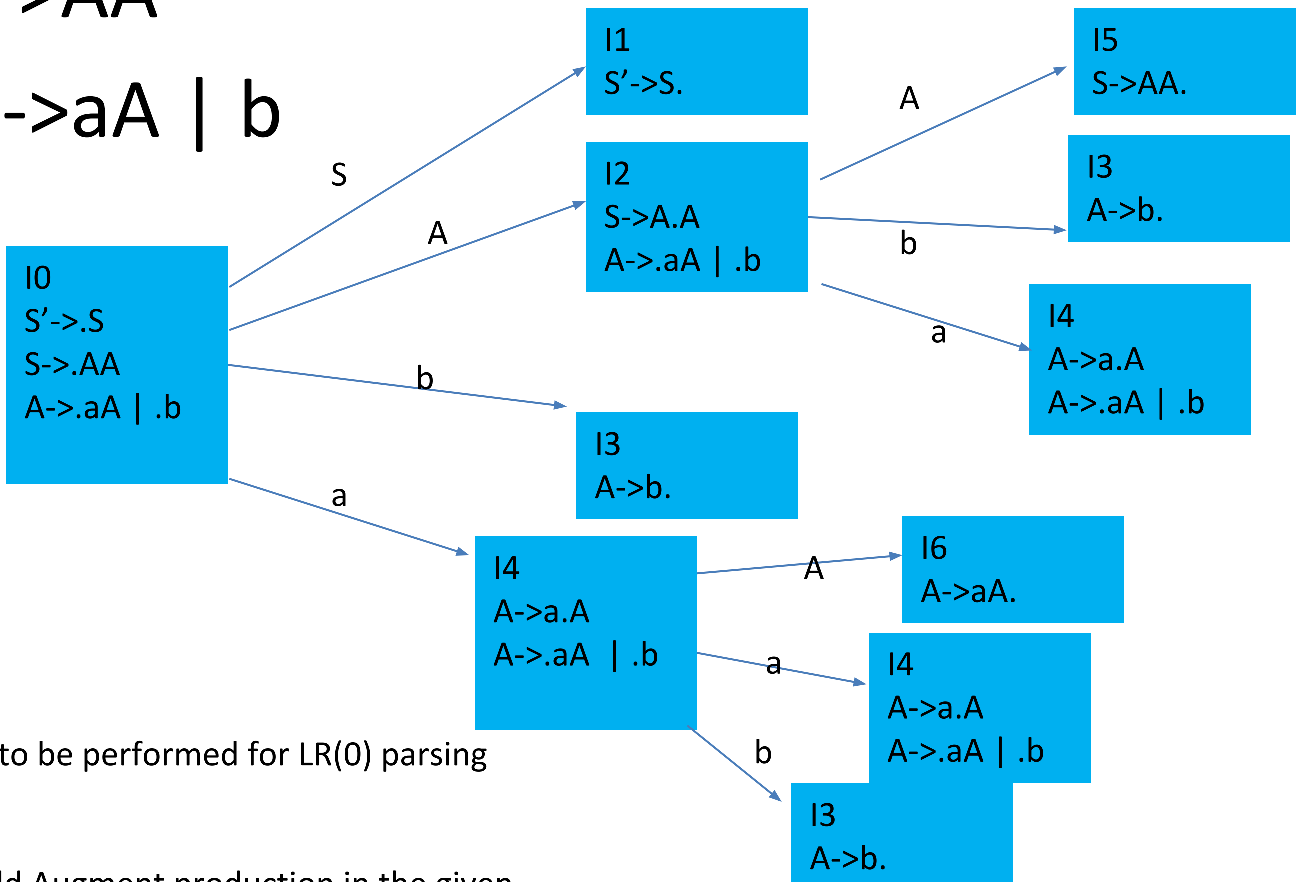


Steps to be performed for LR(0) parsing table

1. Add Augment production in the given grammar
2. Create Canonical collection of LR (0) items
3. Draw a data flow diagram (DFA)
4. Construct a LR(0) parsing table

Example 2: Construct LR(0) Parsing table

- S->AA
- A->aA | b



Steps to be performed for LR(0) parsing table

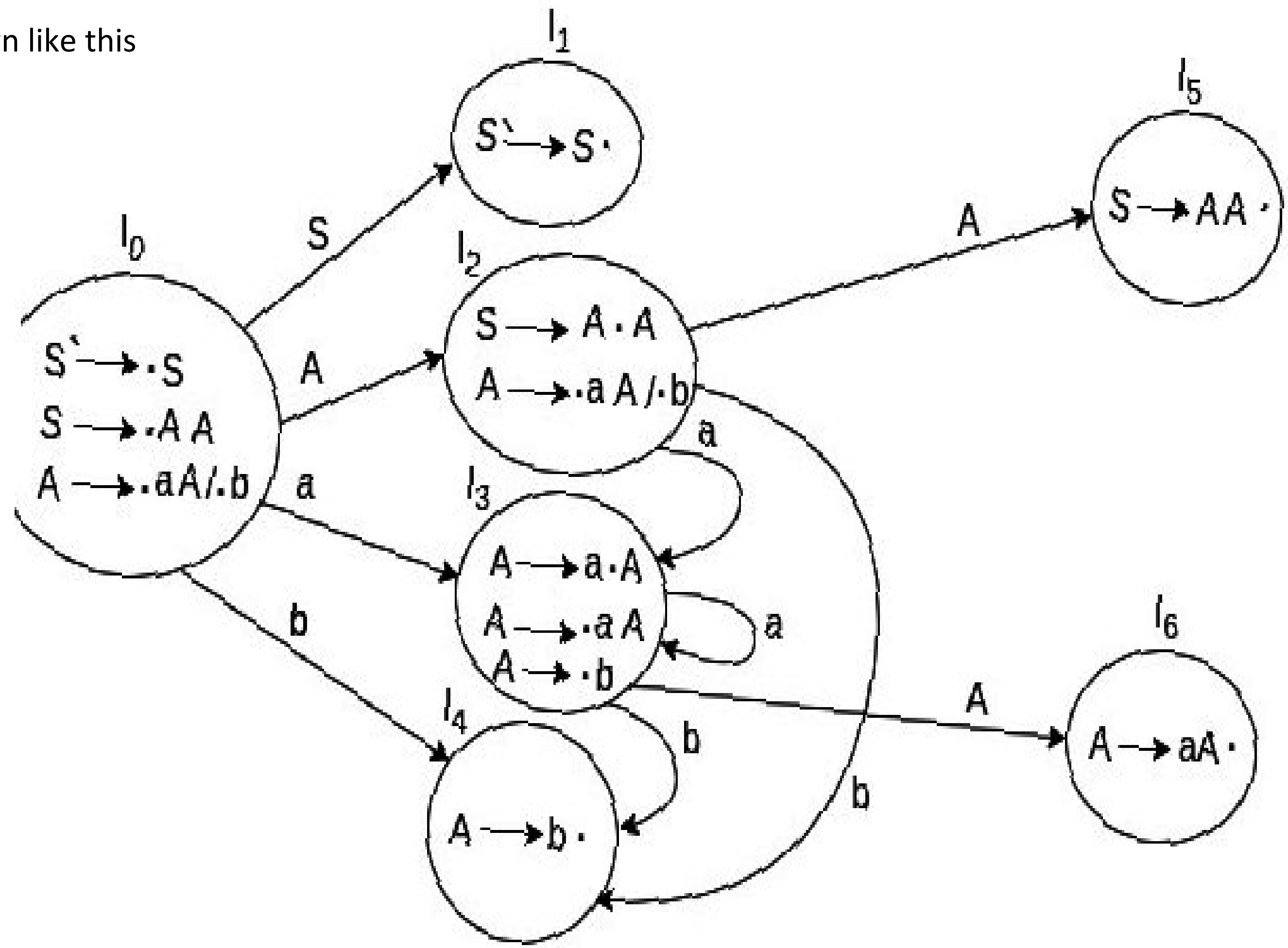
1. Add Augment production in the given grammar
2. Create Canonical collection of LR (0) items
3. Draw a data flow diagram (DFA)
4. Construct a LR(0) parsing table

Example 2: LR(0) Parsing Table

STATES	ACTION			GOTO	
	a	b	$	S	A
I0	S4	S3		1	2
I1			ACCEPT		
I2	S4	S3			5
I3	r3	r3	r3		
I4	S4	S3		6	
I5	r1	r1	r1		
I6	r2	r2	r2		

There is no shift reduce conflict, so it is LR(0) grammar.

DFA can be drawn like this also

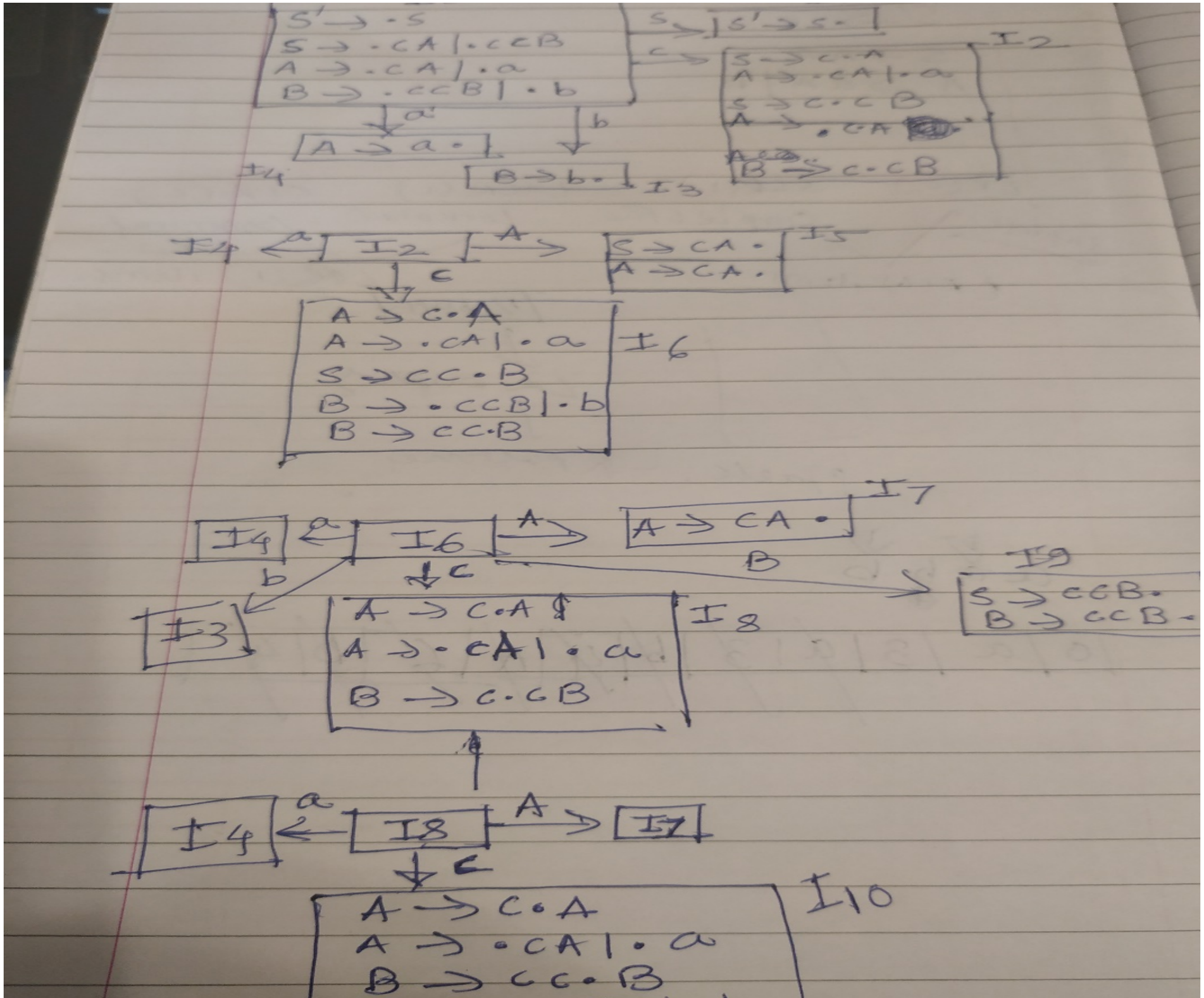


Rules for making entry in the parsing table

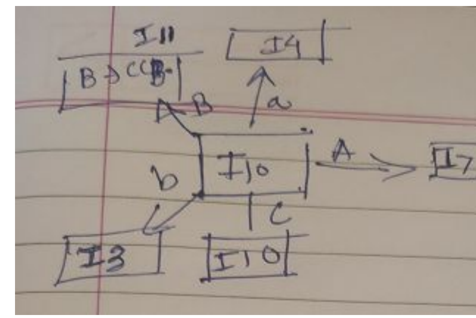
- If a state is going to some other state on a **terminal** then it correspond to a shift move.
- If a state is going to some other state on a **variable(non-terminal)** then it correspond to go to move (goto).
- If a state contains the **final item,** in the **particular row of state** write the reduce node completely

Example 3:
Construct LR(0) parsing table for the following grammar.

- S->cA| ccB
- A->cA | a
- B-> ccB | b



S' → •S
S → •cA | •ccB
A → •cA | •a
B → •ccB | •b
S
S' → S•
c
I2
S → c•A
A → •cA | •a
S → c•cB
A → •cA
B → c•cB
a
A → a•
I4
b
B → b•
I3
I4
a
I2
A
S → cA•
A → cA•
I5
c
A → c•A
A → •cA | •a
S → cc•B
B → •ccB | •b
B → cc•B
I6
I4
a
I6
A
A → cA•
I7
B
b
c
I3
A → c•A
A → •cA | •a
B → c•cB
I8
I9
S → ccB•
B → ccB•
I4
a
I8
A
I7
c
A → c•A
A → •cA | •a
B → cc•B
I10



I11
B → ccB•
I4
a
B
A
I10
I7
b
c
I3
I10

Construct LR(0) parsing table for the following grammar

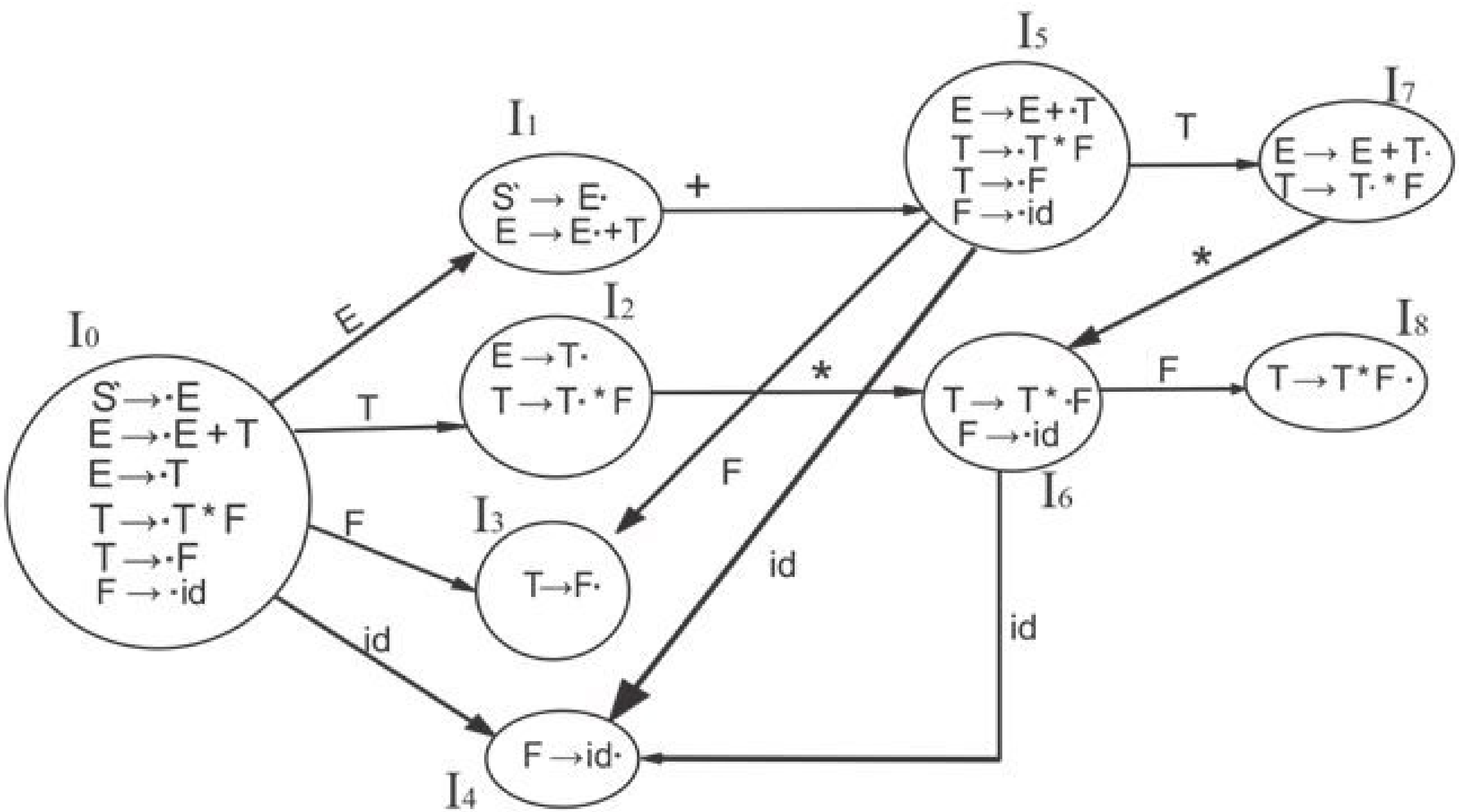
- $S \rightarrow E$

  $E \rightarrow E + T \mid T$

  $T \rightarrow T * F \mid F$

  $F \rightarrow id$

DFA can also be drawn like this



SLR(1) Parsing

SLR (1) refers **to simple LR Parsing**.

It is similar to LR(0) parsing, with 1 difference.

The only difference is in the parsing table.

To construct SLR (1) parsing table, we use the same canonical collection of LR (0) item.

In LR(0) parsing table, there's a chance of 'shift reduced' conflict because we are entering 'reduce' corresponding to all terminal states.

We can solve this problem by <u>entering 'reduce' corresponding to **FOLLOW of LHS of production**</u> in the terminating state.

This is called SLR(1) collection of items

<u>**In the SLR (1) parsing, the reduce move are placed only in the follow of left hand side**</u>.

- Construct SLR(1) parsing table for the following grammar

S->AA

A->aA | b

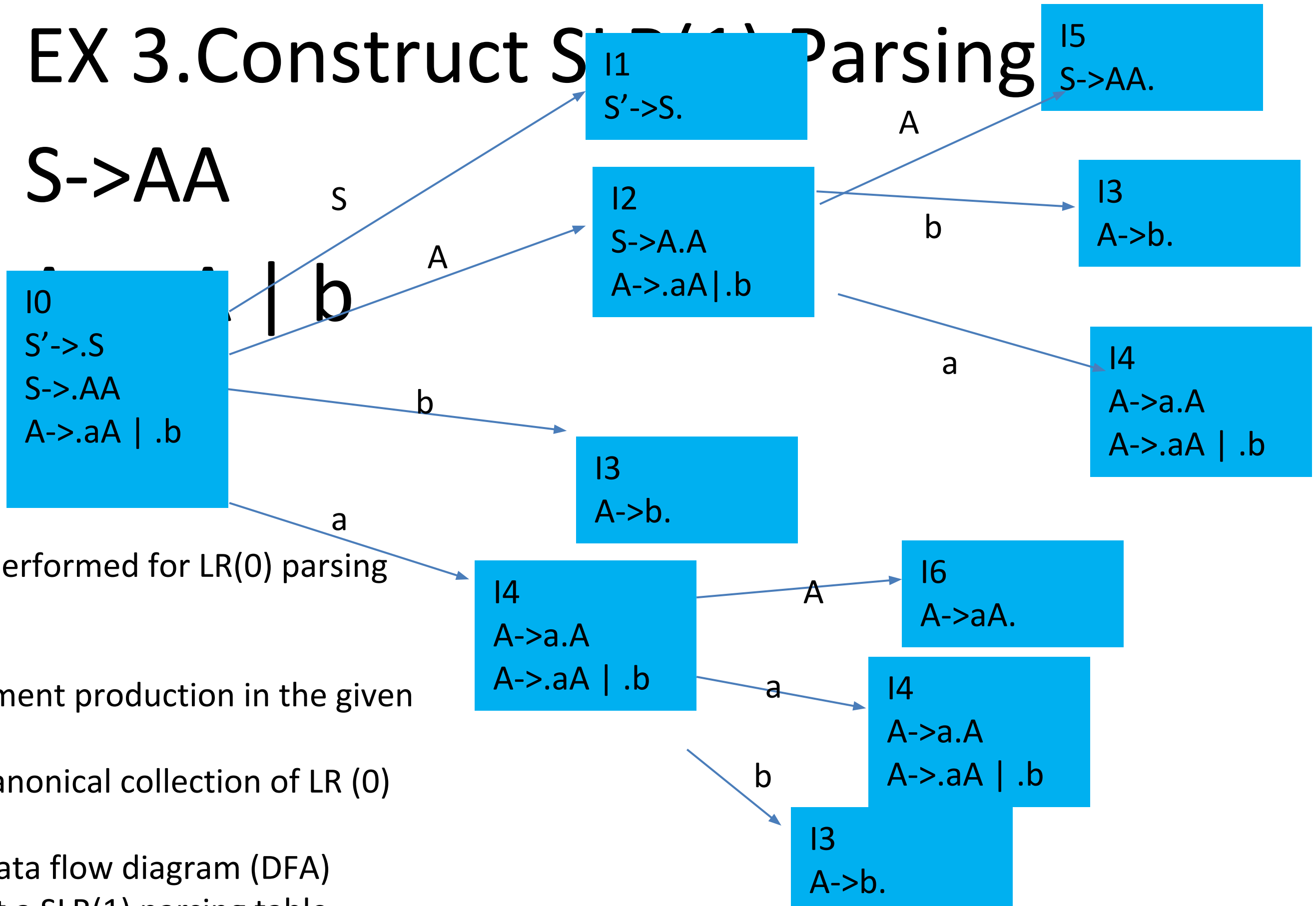
Follow(S)={$}

Follow(A)={FIRST(A)}={a, b}

Steps to be performed for SLR(1) parsing table

- For the given input string write a context free grammar
- Add Augment production in the given grammar
- Create Canonical collection of LR (0) items
- Draw a data flow diagram (DFD)
- Construct a SLR (1) parsing table

- EX 3.Construct SLR(1) Parsing
- S->AA
- A->aA | b



Steps to be performed for LR(0) parsing table

1. Add Augment production in the given grammar
2. Create Canonical collection of LR (0) items
3. Draw a data flow diagram (DFA)
4. Construct a SLR(1) parsing table

Example 3: SLR(1) Parsing Table FOLLOW(S):{$} FOLLOW(A):FIRST(A)={a,b}

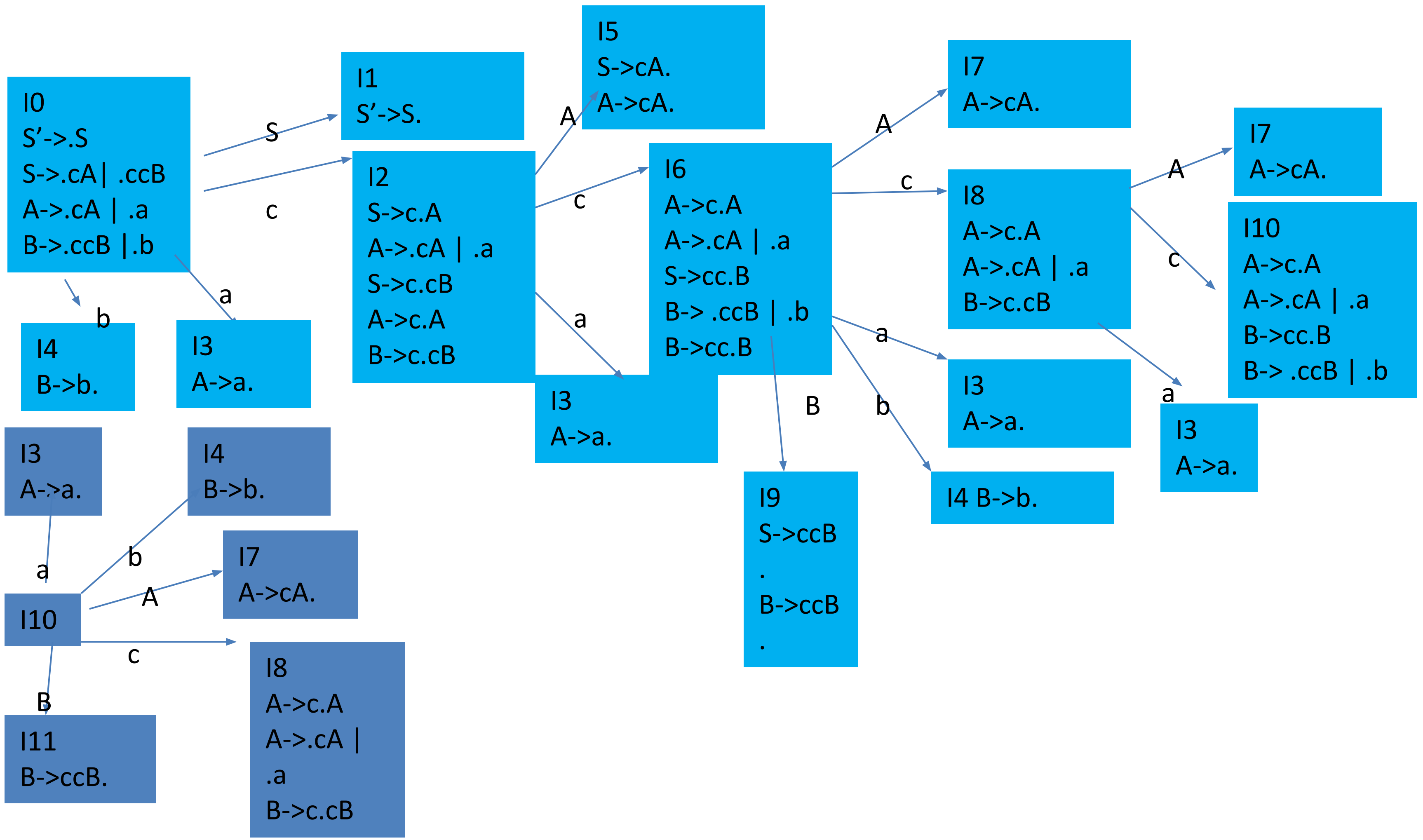
STATES	ACTION			GOTO	
	a	b	$	S	A
I0	S4	S3		1	2
I1			ACCEPT		
I2	S4	S3			5
I3	r3	r3			
I4	S4	S3			6
I5			r1		
I6	r2	r2			

There is no shift reduce conflict, so it is SLR(1) grammar.

Example 4: Construct LR(0) parsing table for the following grammar.

- S->cA(1)| ccB(2)
- A->cA(3) | a(4)
- B-> ccB(5) | b(6)
- S->’S

- Example: 4:LR(0) items



Example4: LR(0) Parsing Table

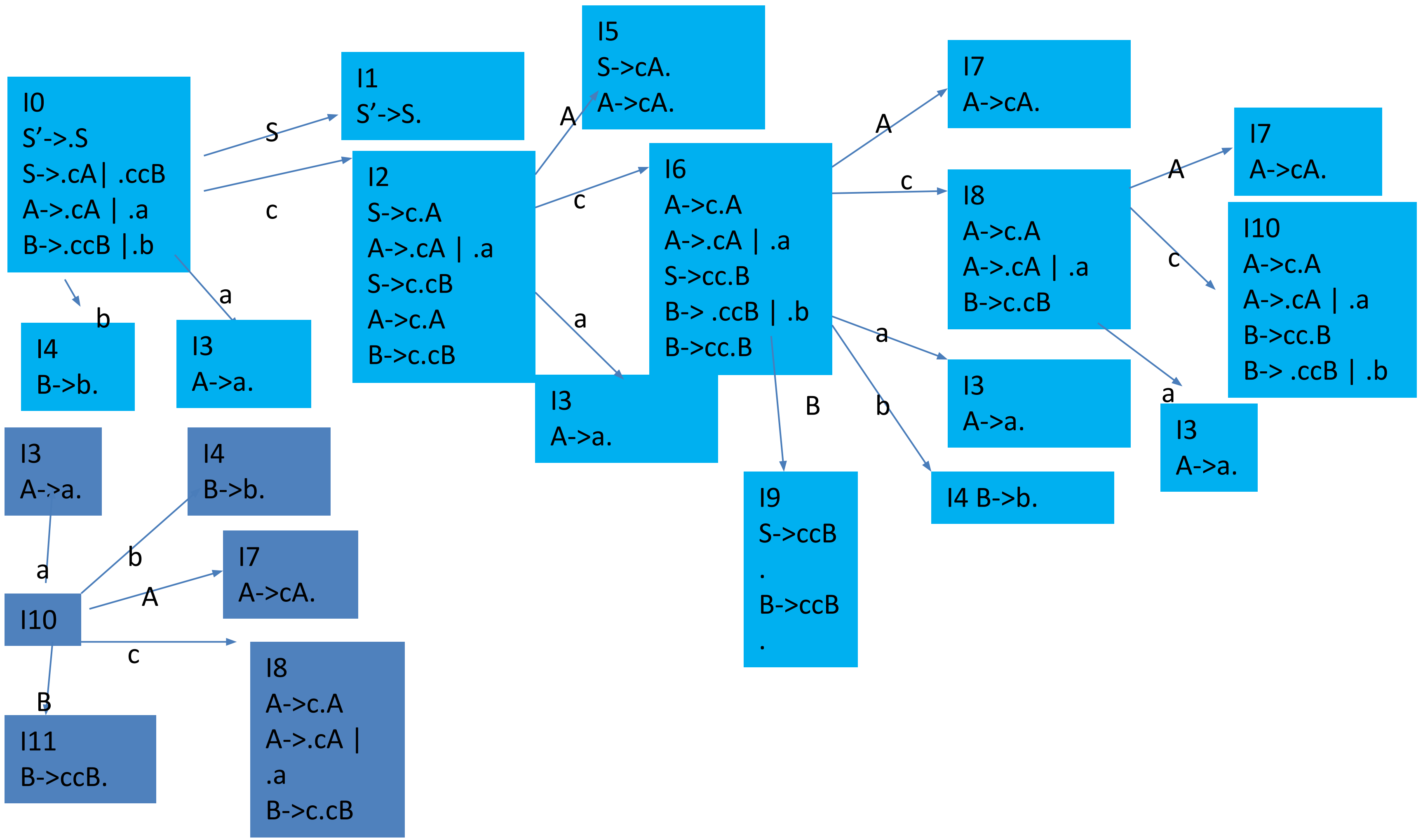
STATES	ACTION				GOTO		
	c	a	b	$	S	A	B
I0	S2	S3	S4		1		
I1				ACCEPT			
I2	S6	S3				5	
I3	R4	r4	r4	r4			
I4	R6	r6	r6	r6			
I5	R1,r3	R1,r3	R1,r3	R1,r3			
I6	S8	S3	S4			7	9
I7	r3	r3	r3	r3			
I8	S10	S3				7	
I9	R2,r5	R2,r5	R2,r5	R2,r5			
I10	S8	S3	S4			7	11
I11	R5	r5	r5	r5			

It is not a LR(0) grammar as it contains reduce, reduce entries.

Example 5: Construct SLR(1) parsing table for the following grammar.

- S->cA(1)| ccB(2)
- A->cA(3) | a(4)
- B-> ccB(5) | b(6)
- S->’S
- FOLLOW(S)={$}
- FOLLOW(A)={$}
- FOLLOW(B)={$}

- Example: 5:LR(0) items



Example5: SLR(1) Parsing Table

STATES	ACTION				GOTO		
	c	a	b	$	S	A	B
I0	S2	S3	S4		1		
I1				ACCEPT			
I2	S6	S3				5	
I3				r4			
I4				r6			
I5				R1,r3			
I6	S8	S3	S4			7	9
I7				r3			
I8	S10	S3				7	
I9				R2,r5			
I10	S8	S3	S4			7	11
I11				r5			

It is not a SLR(1) grammar as it contains reduce, reduce entries.

CLR (1) Parsing: CLR refers to canonical lookahead

- CLR parsing use the canonical collection of LR (1) items to build the CLR (1) parsing table
- CLR (1) parsing table produces the more number of states as compare to the SLR (1) parsing
- In the CLR (1), we place the reduce node only in the lookahead symbols

LR (1) item is a collection of LR (0) items and a look ahead symbol

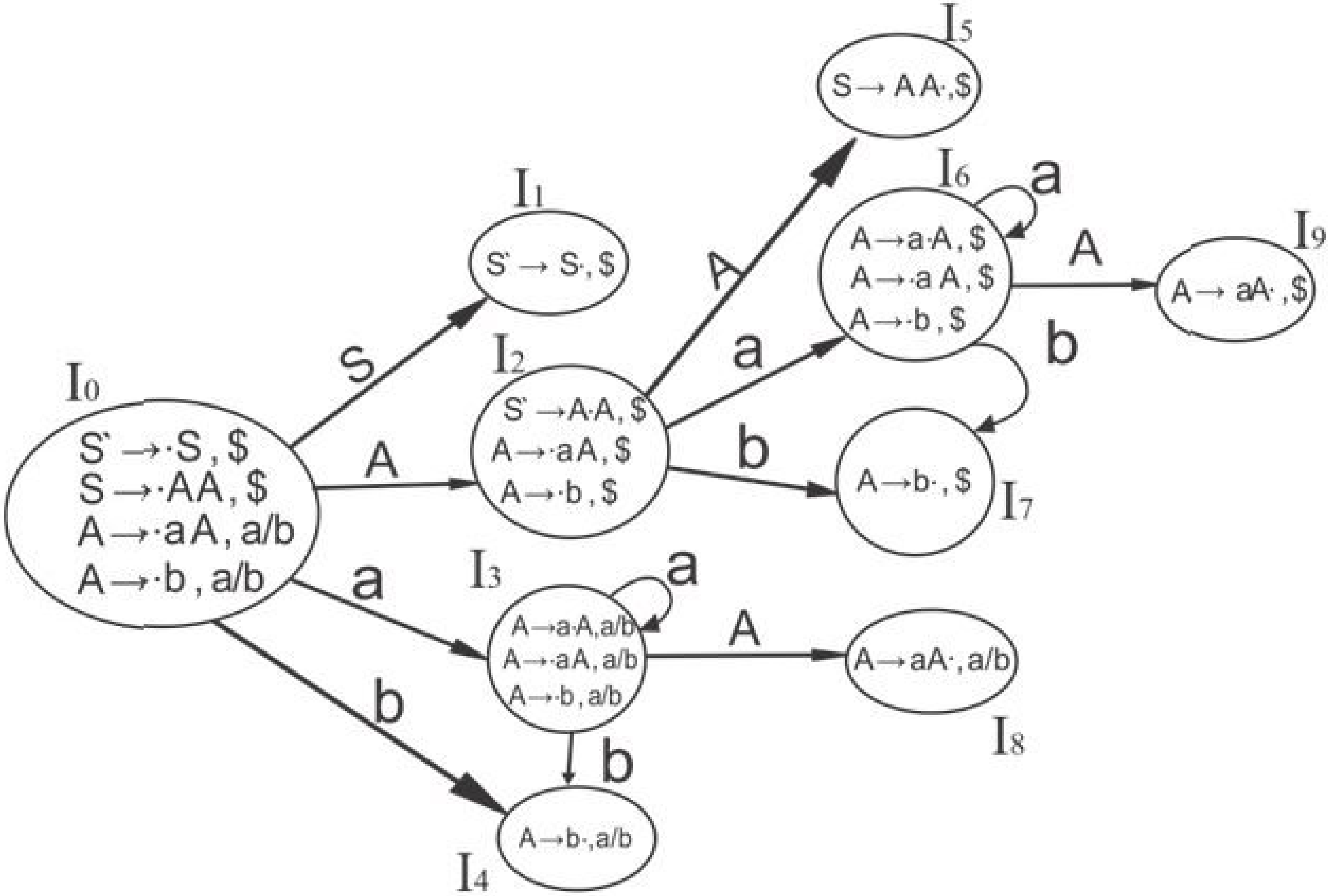
Construct LR(1) Parsing table

- S->AA
- A->aA | b
- S'->.S, $
- S->.AA, $
- A-> .aA, a/b
- A->.b, a/b

Steps to be performed for SLR(1) parsing table

1. Add Augment production in the given grammar
2. Create Canonical collection of LR (1) items
3. Draw a data flow diagram (DFA)
4. Construct a CLR (1) parsing table

DFA can also be drawn like this



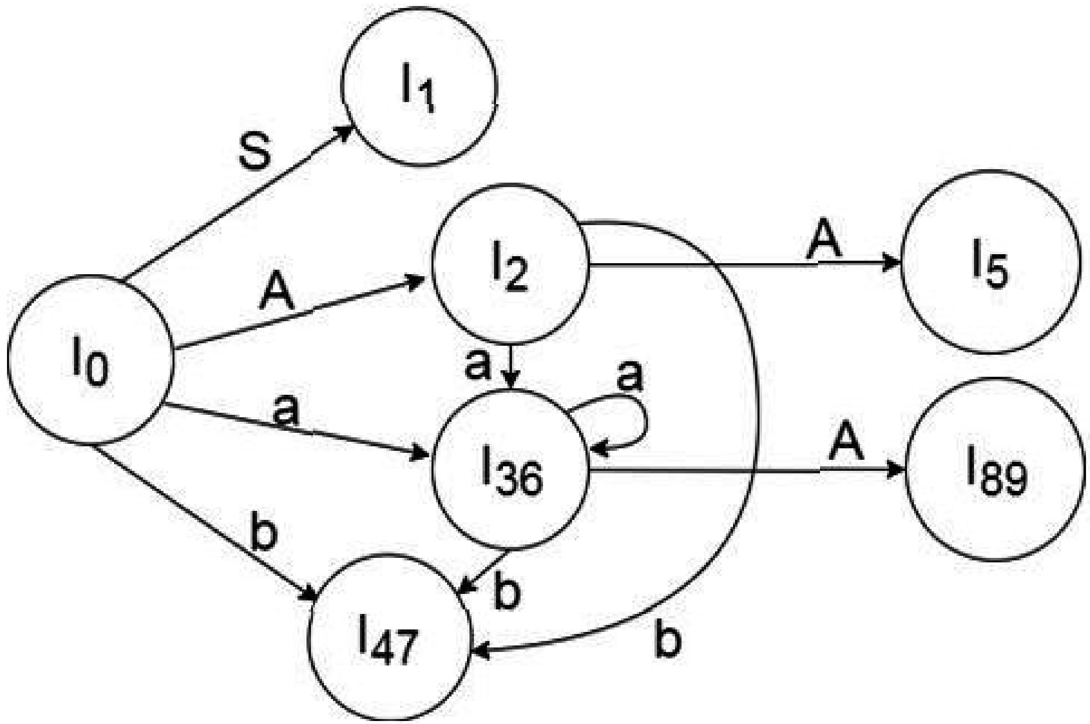
CL(1) Parsing Table

States	a	b	$	S	A
I_0	S_3	S_4			2
I_1			Accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	R_3	R_3			
I_5			R_1		
I_6	S_6	S_7			9
I_7			R_3		
I_8	R_2	R_2			
I_9			R_2		

LALR (1) Parsing

- LALR refers to the lookahead LR
- For constructing LALR (1) parsing table, the canonical collection of LR (1) items are used
- In the LALR (1) parsing, the LR (1) items which have same productions but different look ahead are combined to form a single set of items
- LALR (1) parsing is same as the CLR (1) parsing, only difference in the parsing table

DFA can also be drawn like this



LALR(1) Parsing Table

States	a	b	S	S	A
I_0	S_{36}	S_{47}		1	2
I_1			accept		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	R_3	R_3	R_3		
I_5			R_1		
I_{89}	R_2	R_2	R_2		