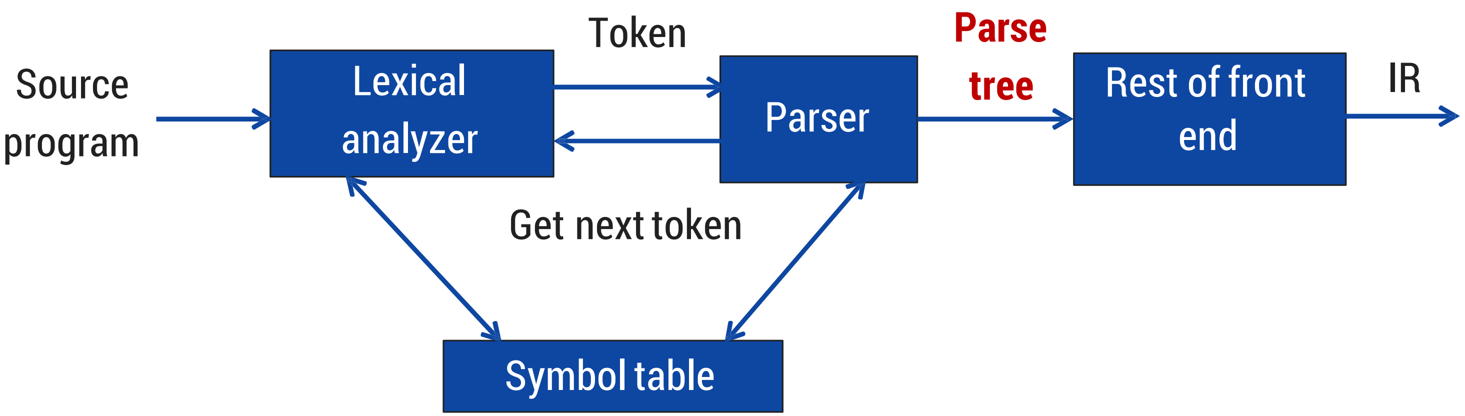
Compiler Design
Unit-3 Syntax Analysis

The Role of the Parser, Types of grammar, CFG, Leftmost derivation, Rightmost derivation, Parse Tree, Restriction on CFG, Ambiguous grammar, TopDown Parsing, Issues of CFG, Recursive Descent Parser, Construction of Predictive Parsing Table , LL (1) Grammar, String Parsing using M-Table, Bottom-Up Parsing: Handle, Shift-reduce parser, LR parsers: LR (0), SLR (1), LALR (1), CLR(1), String parsing procedure using LR parser, R-R and S-R Conflicts.

Role of parser



Syntax Analysis

- Syntax of a language refers to the structure of valid programs/ statements of that language.
  - Specified using certain rules (known as production rules)
  - Collections of such production rules is known as grammar
- Parsing or syntax analysis is a process of determining if a string of tokens can be generated by the grammar
- Parser/syntax analyzer gets string of tokens from lexical analyzer and verifies if that string of tokens is a valid sequence i.e. whether its structures is syntactically correct.

Syntax Analysis

- Other Tasks of parser:
  - Report syntactic errors.
  - Recovery from such errors so as to continue the execution process
- Output of Parser:
  - A representation of parse tree generated by using the stream of tokens provided by the Lexical Analyzer.

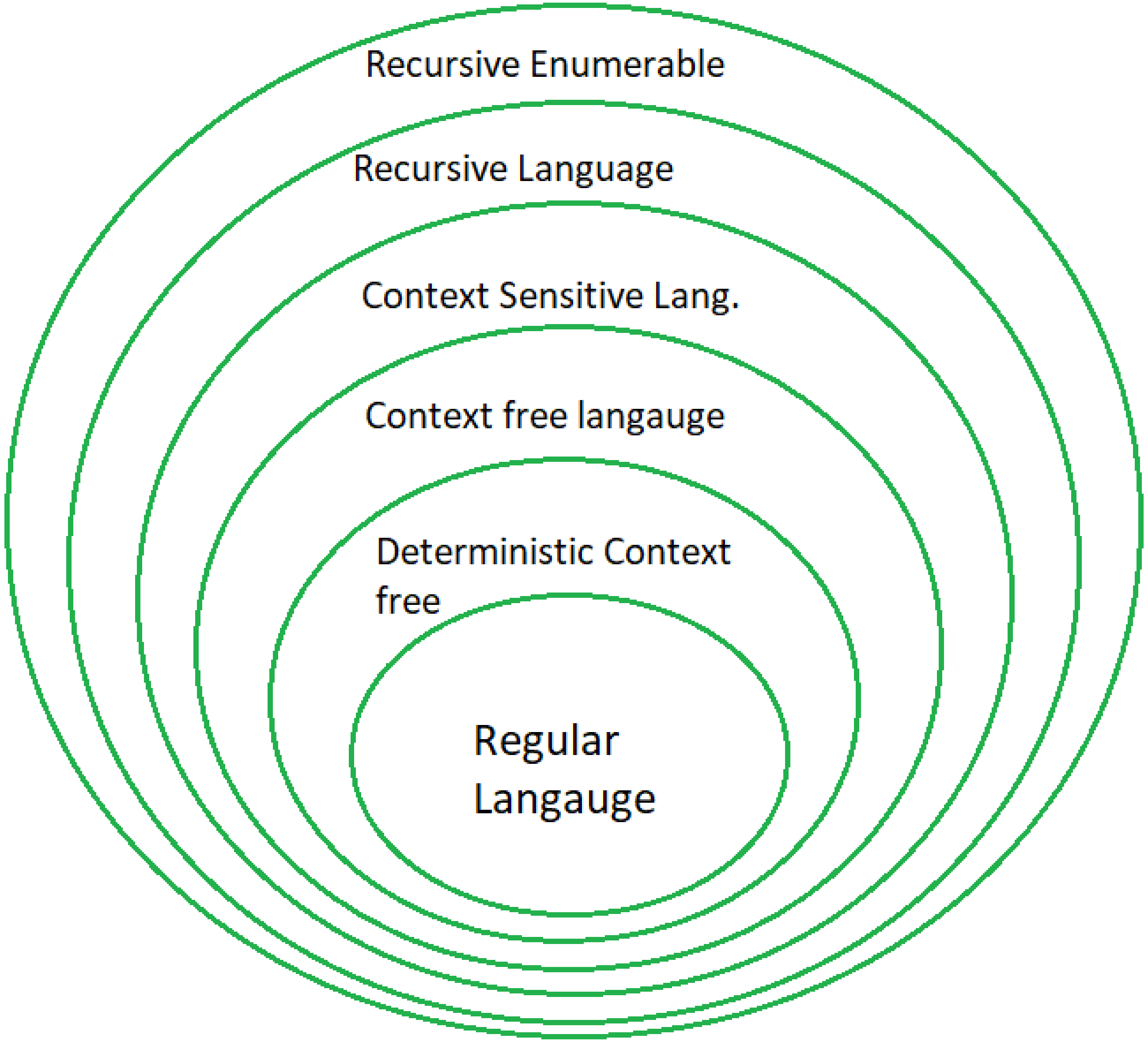
Language

- An alphabet of a language is a set of symbols.
    - Examples : {0,1} for a binary number system(language) = {0,1,100,101,...}
    - {a,b,c} for language={a,b,c, ac,abcc..}
    - {if,(,),else ...} for a if statements={if(a==1)goto10, if--}
- A string over an alphabet:
    - is a sequence of zero or more symbols from the alphabet.
    - Examples : 0,1,10,00,11,111,0101 ... strings for a alphabet {0,1}
    - Null string is a string which does not have any symbol of alphabet.

Language

- Language: Is a subset of all the strings over a given alphabet.

Alphabets Ai	Languages Li for Ai
A0={0,1}	L0={0,1,100,101,...}
A1={a,b,c}	L1={a,b,c, ac, abcc..}
A2={all of C tokens}	L2= {all sentences of C program }


Recursive Enumerable
Recursive Language
Context Sensitive Lang.
Context free langauge
Deterministic Context free
Regular Langauge

Grammar

- A finite set of rules
- that generates only and all sentences of a language.
- that assigns an appropriate structural description to each one.

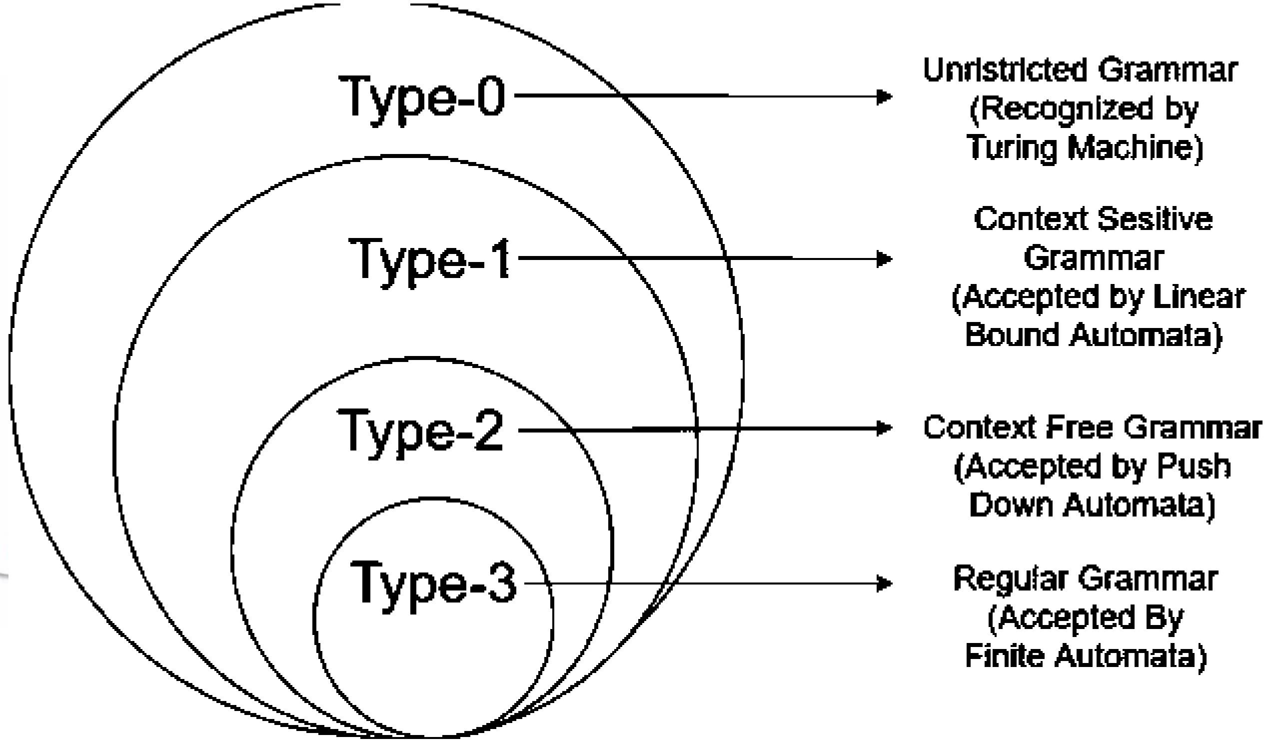
Formal Grammar G

- **formal grammar G** as a production system consisting of the following:
- **A finite alphabet Σ.** The concept of an alphabet used here is a very general one. An alphabet can, for example, consist of all Unicode characters; but it may also consist of all keywords of a programming language, all pictographs of the Sumerian script, the sounds of a bird song (bird songs do have a grammar!2), or the element and attribute names defined for an XML document type.
- **A finite set of non-terminal symbols N.** As the name says, these symbols will not appear in the final document instance but are used only in the production process.
- **A start symbol S** taken out of the set of non-terminal symbols N.
- **A finite set of generative rules R.** Each rule transforms an expression of non-terminal symbols and alphabet symbols (terminal symbols) into another expression of non-terminal symbols and alphabet symbols.

Classification of Grammars

Chomsky Classification of Grammars

Grammar Type	Grammar Accepted (generator)	Language Accepted	Automaton (recognizers)
Type 0	Unrestricted grammar	Recursively enumerable language	Turing Machine
Type 1	Context-sensitive grammar	Context-sensitive language	Linear-bounded automaton
Type 2	Context-free grammar	Context-free language	Pushdown automaton
Type 3	Regular grammar	Regular language	Finite state automaton



Type-0 Recursively Enumerable Grammar

- Type-0 grammars (unrestricted grammars) include all formal grammars.
- They generate exactly all languages that can be recognized by a Turing machine.
- These languages are also known as the recursively enumerable languages.
- **Note that** this is different from the recursive languages which can be decided by an always-halting Turing machine.
- Class 0 grammars are too general to describe the syntax of programming languages and natural languages.

Type 1: Context-sensitive grammars

- Type-1 grammars generate the context-sensitive languages.
- These grammars have rules of the form $\alpha \to \beta$ where $\alpha, \beta \in (T \cup N)^*$ and $len(\alpha) \le len(\beta)$ and α should contain atleast 1 non terminal.
- The languages described by these grammars are exactly all languages that can be recognized by a linear bounded automaton.
- **Example:**
- $AB \to CDB$
- $AB \to CdEB$
- $ABcd \to abCDBcd$
- $B \to b$

Type 2: Context-free grammars

- Type-2 grammars generate the context-free languages.
- These grammars have rules of the form $A \rightarrow \rho$ where $A \in N$ and $\rho \in (T \cup N)^*$.
- The languages described by these grammars are exactly all languages that can be recognized by a non-deterministic pushdown automaton.
- **Example:**
- $A \rightarrow aBc$

Type 3: Regular grammars

- Type-3 grammars generate the regular languages.
- These grammars have rules of the form $A \rightarrow a$ or $A \rightarrow aB$ where $A,B \in N$(non terminal) and $a \in T$(Terminal).
- These languages are exactly all languages that can be decided by a finite state automaton. Additionally,this family of formal languages can be obtained by regular expresions. Regular languages are commonly used to define search patterns and the lexical structure of programming languages.
- **Example:**
- $A \rightarrow \varepsilon$
- $A \rightarrow a$
- $A \rightarrow abc$
- $A \rightarrow B$
- $A \rightarrow abcB$

Chomsky Classification of Languages

Grammar Type	Production Rules	Language Accepted	Automata
Type-3 (Regular Grammar)	$A \to a$ or $A \to aB$ where $A, B \in N$(non terminal) and $a \in T$(Terminal)	Regular (RL)	Finite Automata (FA)
Type-2 (Context Free Grammar)	$A \to \rho$ where $A \in N$ and $\rho \in (T \cup N)^*$	Context Free (CFL)	Push Down Automata (PDA)
Type-1 (Context Sensitive Grammar)	$\alpha \to \beta$ where $\alpha, \beta \in (T \cup N)^*$ and len(α) <= len(β) and α should contain atleast 1 non terminal.	Context Sensitive (CSL)	Linear Bound Automata (LBA)
Type-0 (Recursive Enumerable)	$\alpha \to \beta$ where $\alpha, \beta \in (T \cup N)^*$ and α contains atleast 1 non-terminal	Recursive Enumerable (RE)	Turing Machine (TM)

Context Free Grammar

- A context free grammar (CFG) is a 4-tuple $G = (V, \Sigma, S, P)$ where,
  - V is finite set of non terminals,
  - Σ is disjoint finite set of terminals,
  - S is an element of V and it's a start symbol,
  - P is a finite set formulas of the form $A \rightarrow \alpha$ where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$

Context Free Grammar

- **Nonterminal symbol:**
  - The name of syntax category of a language, e.g., noun, verb, etc.
  - ⮑ The It is written as a **single capital letter**, or as a **name enclosed between < … >,** e.g., A or
    - <Noun>

<Noun Phrase> → <Article><Noun>
<Article> → a | an | the
<Noun> → boy | apple

Context Free Grammar

- Terminal symbol:
  - ↪ A symbol in the alphabet.
  - ↪ It is denoted by lower case letter and punctuation marks used in language.

<Noun Phrase> → <Article><Noun>
<Article> → a | an | the
<Noun> → boy | apple

Context Free Grammar

- Start symbol:
  - ↪ First nonterminal symbol of the grammar is called start symbol.

<Noun Phrase> → <Article><Noun>
<Article> → a | an | the
<Noun> → boy | apple

Context Free Grammar

- **Production:**
  A production, also called a rewriting rule, is a rule of grammar. It has the form of
  A nonterminal symbol → String of terminal and nonterminal symbols

<Noun Phrase> → <Article><Noun>
<Article> → a | an | the
<Noun> → boy | apple

Example of Context Free Grammar

- Write terminals, non terminals, start symbol, and productions for following grammar.
- E → E O E| (E) | -E | id
- O → + | - | * | / | ↑

Example of Context Free Grammar

- Write terminals, non terminals, start symbol, and productions for following grammar.
- E → E O E| (E) | -E | id
- O → + | - | * | / | ↑
- **Terminals:** **id + - * / ↑ ( )**
- **Non terminals: E, O**
- **Start symbol: E**
- **Productions:** **E → E O E| (E) | -E | id**
  O → + | - | * | / | ↑

Example of Grammar

- Grammar for expressions consisting of digits and plus and minus signs.
- Grammar G for a language L={9-5+2, 3-1, ...}
- G=(N,T,P,S)
- N={list,digit}
- T={0,1,2,3,4,5,6,7,8,9,-,+}
- P :
  - list -> list + digit
  - list -> list - digit
  - list -> digit
  - digit -> 0|1|2|3|4|5|6|7|8|9
- S=list

Example of Grammar

- Some definitions for a language L and its grammar G
- Derivation : A sequence of replacements $S \Rightarrow \alpha 1 \Rightarrow \alpha 2 \Rightarrow \ldots \Rightarrow \alpha n$ is a derivation of αn.
- Language of grammar L(G)
  - L(G) is a set of sentences that can be generated from the grammar G.
  - $L(G)=\{x \mid S \Rightarrow^* x\}$ where $x \in$ a sequence of terminal symbols
- Example: Consider a grammar G=(N,T,P,S):
  - N={S} T={a,b}
  - S=S P ={S → aSb | ε }
  - is aabb a sentecne of L(g)? (derivation of string aabb)
  - $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aa\varepsilon bb \Rightarrow aabb$(or $S \Rightarrow^*$ aabb) so, aabbεL(G)
  - there is no derivation for aa, so $aa \notin L(G)$
  - note L(G)={anbn| n≧0} where anbn meas n a's followed by n b's.

Example of Grammar

- Example: S->aSb|ε

Simplifying Context Free Grammars

- Some of the productions of CFGs are not useful and are redundant.
- Types of redundant productions and the procedure of removing them are mentioned below.
- **1. Useless productions**
- **2. λ productions (lambda productions or null productions)**
- **3. Unit productions**

1. Useless productions

- The productions that can never take part in derivation of any string , are called useless productions. Similarly , a variable that can never take part in derivation of any string is called a useless variable.
- **Example:**
- S -> abS | abA | abB
- A -> cd
- B -> aB
- C -> dc
- production ‘C -> dc’ is useless because the variable ‘C’ will never occur in derivation of any string.
- Production ‘B ->aB’ is also useless because there is no way it will ever terminate .
- To remove useless productions , So the modified grammar becomes –
- S -> abS | abA
- A -> cd

2. λ productions (lambda productions or null productions)

- The productions of type 'A -> λ' are called λ productions.
- These productions can only be removed from those grammars that do not generate λ (an empty string). It is possible for a grammar to contain null productions and yet not produce an empty string.
- Consider the grammar –
- S -> ABCd (1)
- A -> BC (2)
- B -> bB | λ (3)
- C -> cC | λ (4)

2. λ productions (lambda productions or null productions)

- start with the first production. Add the first production as it is. Then we create all the possible combinations that can be formed by replacing the nullable variables with λ.
- S -> ABCd | ABd | ACd | BCd | Ad | Bd |Cd | d
- A -> BC | B | C
- B -> bB | b
- C -> cC | c

3. Unit productions

- The productions of type ‘A -> B’ are called unit productions.
- The unit productions are the productions in which one non-terminal gives another non-terminal.
- Example:
- S -> Aa | B
- A -> b | B
- B -> A | a
- Last Result:
- S->Aa|b|a
- A->b|a

Derivation

- Derivation is used to find whether the string belongs to a given grammar or not.
- Types of derivations are:
    1. Leftmost derivation
    2. Rightmost derivation

Leftmost derivation

- A derivation of a string W in a grammar G is a left most derivation if at every step the left most non terminal is replaced.
- Grammar: $S \rightarrow S+S \mid S-S \mid S*S \mid S/S \mid a$ Output string: a*a-a

Leftmost derivation

- A derivation of a string W in a grammar G is a left most derivation if at every step the left most non terminal is replaced.
- Grammar: S→S+S | S-S | S*S | S/S | a   Output string: a*a-a

S

→**S-S**

→**S*S**-S

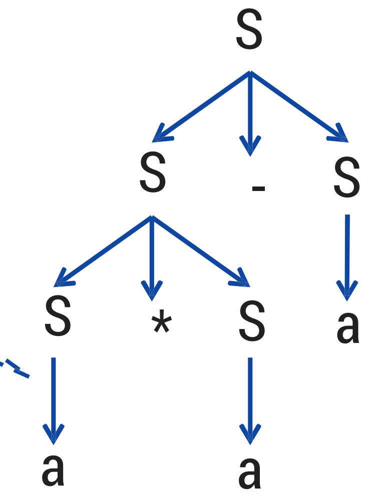
→**a***S-S

→a***a**-S

→a*a-**a**

Leftmost Derivation

Parse tree represents the structure of derivation



Parse tree

Rightmost derivation

- A derivation of a string W in a grammar G is a right most derivation if at every step the right most non terminal is replaced.
- It is all called canonical derivation.
- Grammar: S→S+S | S-S | S*S | S/S | a   Output string: a*a-a

Rightmost derivation

- A derivation of a string W in a grammar G is a right most derivation if at every step the right most non terminal is replaced.
- It is all called canonical derivation.
- Grammar: S→S+S | S-S | S*S | S/S | a   Output string: a*a-a

S

→**S*S**

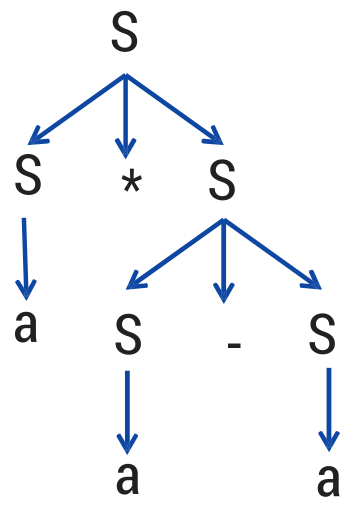
→S***S-S**

→S*S-**a**

→S***a**-a

→**a***a-a

Rightmost Derivation



Parse Tree

Question-1: Derivation

1. Perform leftmost derivation and draw parse tree.

- $S \rightarrow A1B$
  - $A \rightarrow 0A \mid \epsilon$
  - $B \rightarrow 0B \mid 1B \mid \epsilon$
  - Output string: 1001

Question-2: Derivation

- Perform leftmost derivation and draw parse tree. $S \rightarrow 0S1 \mid 01$ Output string: 000111

Question-3: Derivation

- Perform rightmost derivation and draw parse tree.
  - E→E+E | E*E | id | (E) | -E Output string: id + id * id

Parse Tree

- A Parse tree is pictorial depiction of how a start symbol of a grammar derives a string in the language.
- Example:
- A->PQR
- P->a
- Q->b
- R->c|d
- Root is always labelled with the start symbol.
- Each leaf is labelled with a terminal (tokens)
- Each interior node is labelled by a non terminal.

Parse tree

- **Yield of Parse tree:** The Leaves of a parse tree when read from left to right form the yield.
- Language defined by a grammar is set of all strings that are generated by some parse tree formed by that grammar (starting symbol of grammar).
- General Types of Parser:
  - Universal Parser:
    - It can parse any kind of grammar
    - Not very Efficient
    - CYK Algorithm, Earley’s Algorithm
  - Top down Parser:
    - Builds the parse tree from root (top) to leaves (bottom)
  - Bottom up Parser:
    - Builds the parse tree from leaves (bottom) to root(top)

Ambiguous Grammar

- Ambiguity
- Ambiguity, is a word, phrase, or statement which contains more than one meaning.

Chip

Ambiguous Grammar

- Ambiguity
- Ambiguity, is a word, phrase, or statement which contains more than one meaning.

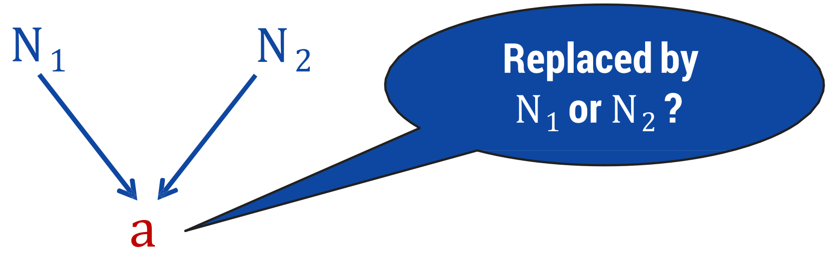
A long thin piece of potato

Chip

A small piece of silicon

Ambiguity

- In formal language grammar, ambiguity would arise if identical string can occur on the RHS of two or more productions.
- Grammar:
  - $N1 \rightarrow \alpha$
  - $N2 \rightarrow \alpha$
- α can be derived from either N1 or N2



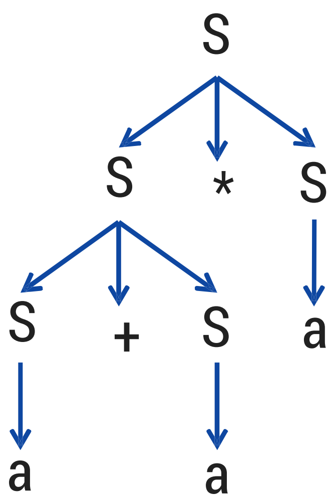
Ambiguous grammar

- Ambiguous grammar is one that produces <u>more than one leftmost</u> or <u>more than one rightmost</u> derivation for the same sentence.
- Grammar: $S \rightarrow S+S \mid S*S \mid (S) \mid a$ Output string: a+a*a

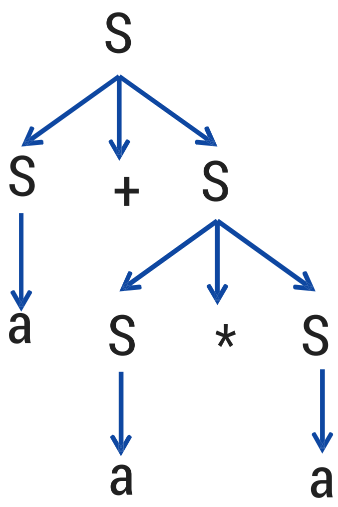
Ambiguous grammar

- Ambiguous grammar is one that produces <u>more than one leftmost</u> or <u>more then one rightmost</u> derivation for the same sentence.
- Grammar: S→S+S | S*S | (S) | a   Output string: a+a*a

S
→S*S
→S+S*S
→a+S*S
→a+a*S
→a+a*a



S
→S+S
→a+S
→a+S*S
→a+a*S
→a+a*a



- Here, Two leftmost derivation for string a+a*a is possible hence, above grammar is ambiguous.

Check Ambiguity in following grammars:

1. S→ aS | Sa | ∈ (output string: aaaa)
2. S→ aSbS | bSaS | ∈ (output string: abab)
3. S→ SS+ | SS* | a (output string: aa+a*)
4. <exp> → <exp> + <term> | <term>
   - <term> → <term> * <letter> | <letter>
   - <letter> → a|b|c|…|z (output string: a+b*c)
5. Prove that the CFG with productions: S → a | Sa | bSS | SSb | SbS is ambiguous (Hint: consider output string yourself)

Associativity of Operators

- When an operand has operators on both its sides (left and right) then we need rules to decide with which operator we will associate this operand.
- Left Associative & Right Associative
- +:Left Associative
- -, *,/:Left Associative
- =, ↑:Right Associative
- Parse trees for left associative operators are more towards left side in terms of length.
- Parse trees for right associative operators are more towards right side in terms of length.

Precedence of Operators

- Whenever an operator has a higher precedence than the other operator, it means that the first operator will get its operands before the operator with lower precedence.
- *,/ **>** +,-

Converting Ambiguous grammar to unambiguous grammar

- * / -> Left, Higher
- + - -> Left, Lower
- E->E+T|E-T|T
- T->T*F|T/F|F
- F->id
- **Left Recursion:** If $A \overset{+}{\Rightarrow} A\alpha$
- **Right Recursion:** If $A \overset{+}{\Rightarrow} \alpha A$

Left recursion

A grammar is said to be left recursive if it has a non terminal A such that there is a derivation $A \rightarrow A\alpha$ for some string α.

- Direct Left Recursion
- A->Aa
- Indirect Left Recursion
- S->Aa
- A->Sb

- Why need to remove Left Recursion?
- Top Down Parsers can not handle left recursion/grammars with having left recursion
- Left recursion elimination:
- $A \rightarrow A\alpha|\beta \Rightarrow$
- $A \rightarrow \beta A'$
- $A' \rightarrow \alpha A'| \in$

Left recursion

- **Advantages:**
- We are able to generate the same language even after remaining Left Recursion

- **Disadvantages:**
- The procedure of Left Recursion only eliminates direct Left Recursion but not indirect Left Recursion.

Left recursion

A grammar is said to be left recursive if it has a non terminal A such that there is a derivation $A \rightarrow A\alpha$ for some string α.

Algorithm to eliminate left recursion

1. Arrange the non terminals in some order $A_1, \dots, An$
2. For $i := 1\ to\ n$ **do begin**

   for $j := 1\ to\ i - 1$ **do begin**

   replace each production of the form $A_i \rightarrow Ai\gamma$

   by the productions $A_i \rightarrow \delta_1\gamma | \delta_2\gamma | \dots\dots | \delta_k\gamma$,

   where $A_j \rightarrow \delta_1 | \delta_2 | \dots . | \delta_k$ are all the current A_j productions;

   end

   eliminate the immediate left recursion among the A_i - productions

end

Example of Left Recursion

- $E \rightarrow E + T|T$
- T$\rightarrow T * F|F$
- $F \rightarrow (E)|id$

- Final Result:
- $E \rightarrow TE'$
- $E \rightarrow +TE'|\epsilon$
- T$\rightarrow FT'$
- T$\rightarrow * FT'| \in$
- $F \rightarrow (E)|id$

- $A \rightarrow A\alpha|\beta \Rightarrow$
- $A \rightarrow \beta A'$
- $A' \rightarrow \alpha A'| \in$

Example of Left Recursion

- S→ $S0S1S|01$

- Final Result:
- S→ $01S'$
- $\mathrm{S} \rightarrow 0S1SS'| \in$

- $A \rightarrow A\alpha|\beta \Rightarrow$
- $A \rightarrow \beta A'$
- $A' \rightarrow \alpha A'| \in$

Example of Left Recursion

- L$\rightarrow L, S|S$

- Final Result:
- $L \rightarrow SL'$
- $\mathrm{L}' \rightarrow, SL'| \in$

- $A \rightarrow A\alpha|\beta \Rightarrow$
- $A \rightarrow \beta A'$
- $A' \rightarrow \alpha A'| \in$

Example of Left Recursion

- $S \rightarrow SX|SSb|XS|a$

- Final Result:
- $S \rightarrow XSS'|aS'$
- $S' \rightarrow XS'|SbS'| \in$

- $A \rightarrow A\alpha|\beta \Rightarrow$
- $A \rightarrow \beta A'$
- $A' \rightarrow \alpha A'| \in$

Example of Left Recursion

- $A \rightarrow AA|Ab$
- Final Result:
- $A' \rightarrow AA'|bA'| \in$

- $A \rightarrow A\alpha|\beta \Rightarrow$
- $A \rightarrow \beta A'$
- $A' \rightarrow \alpha A'| \in$

Example: Left Recursion Elimination

$E \rightarrow E+T \mid T$

Example: Left Recursion Elimination

$E \rightarrow E + T \mid T$

$E \rightarrow TE'$

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

Example: Left Recursion Elimination

$T \rightarrow T*F \mid F$

Example: Left Recursion Elimination

$T \rightarrow T{*}F \mid F$

$T \rightarrow FT'$

$T' \rightarrow {*}FT' \mid \varepsilon$

Example: Left Recursion Elimination

X→X%Y | Z

Example: Left Recursion Elimination

$X \rightarrow X\%Y \mid Z$

$X \rightarrow ZX'$

$X' \rightarrow \%YX' \mid \varepsilon$

Questions:

1. $A \rightarrow Abd \mid Aa \mid a$

   $B \rightarrow Be \mid b$
2. $A \rightarrow AB \mid AC \mid a \mid b$
3. $S \rightarrow A \mid B$

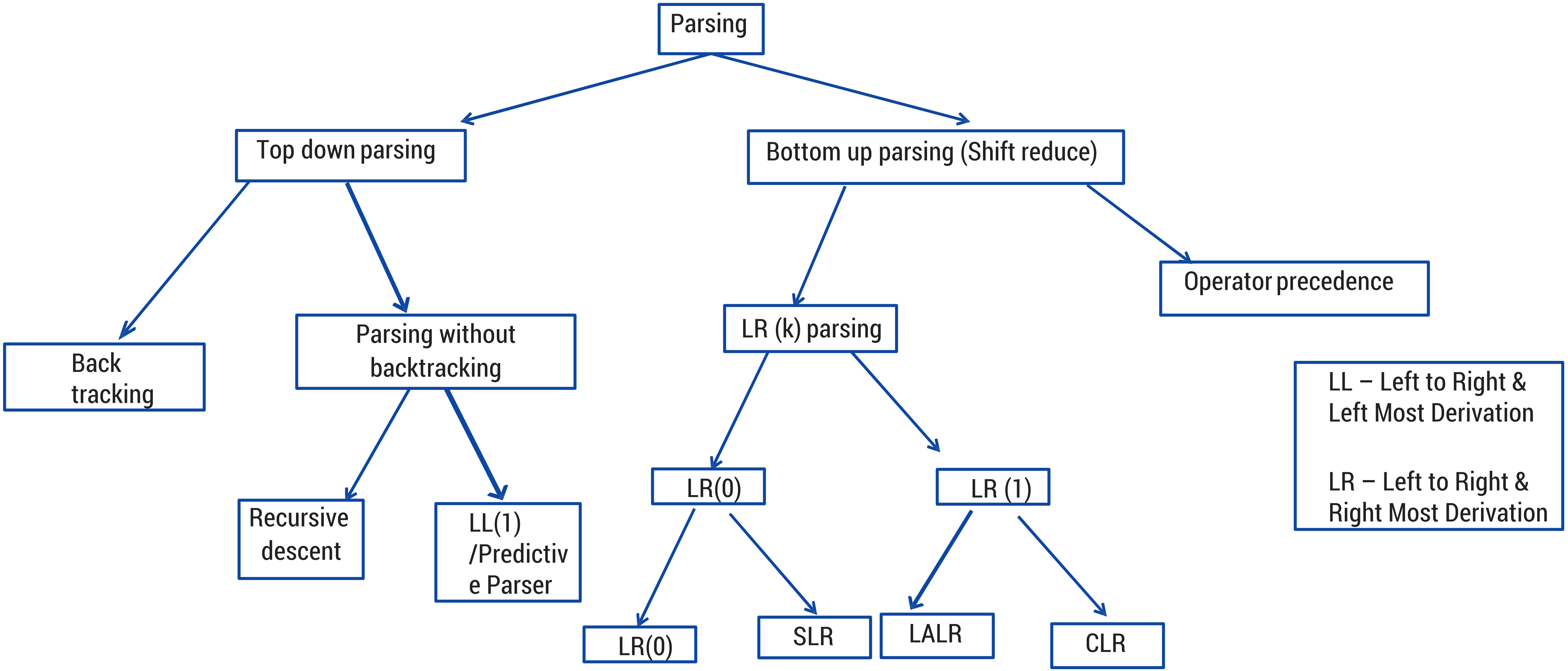
   $A \rightarrow ABC \mid Acd \mid a \mid aa$

   $B \rightarrow Bee \mid b$
4. $Exp \rightarrow Exp+term \mid Exp-term \mid term$

Parsing

- Parsing is a technique that takes input string and produces output either a parse tree if string is valid sentence of grammar, or an error message indicating that string is not a valid.

Classification of parsing methods



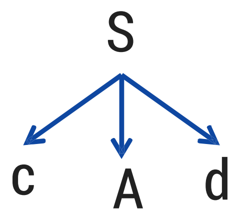
Backtracking

- In backtracking, expansion of nonterminal symbol we choose one alternative and if any mismatch occurs then we try another alternative.
- **Grammar:**
- S→ cAd
- A→ ab | a

- **Input string:** cad

Backtracking

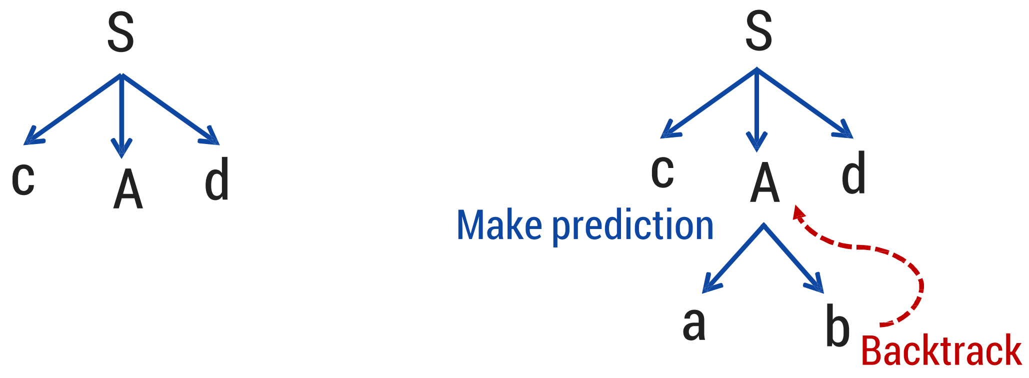
- In backtracking, expansion of nonterminal symbol we choose one alternative and if any mismatch occurs then we try another alternative.
- Grammar:
- S→ cAd
- A→ ab | a



- Input string: cad

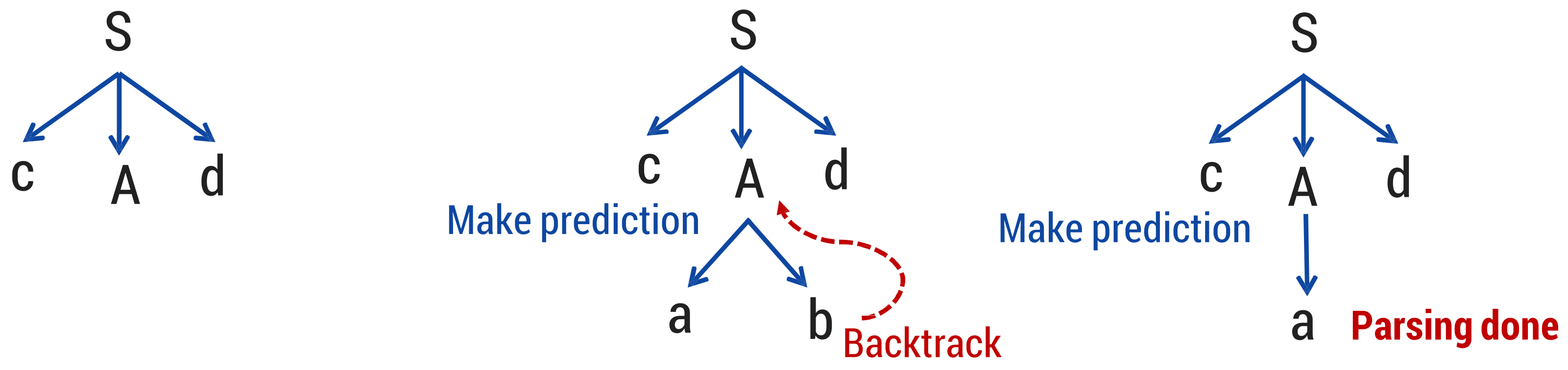
Backtracking

- In backtracking, expansion of nonterminal symbol we choose one alternative and if any mismatch occurs then we try another alternative.
- Grammar:
- S→ cAd
- A→ ab | a
- Input string: cad



Backtracking

- In backtracking, expansion of nonterminal symbol we choose one alternative and if any mismatch occurs then we try another alternative.
- Grammar:
- S→ cAd
- A→ ab | a
- Input string: cad



Question

- E→ 5+T | 3-T
  - T→ V | V*V | V+V V→ a | b
  - **String:** 3-a+b

Left Factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.

- At times, it is not clear which out of 2 (or more) productions to use to expand a non terminal because multiple productions begin with same lookahead.
- A->aa
- A->ab
- A->ac

Input String: ac

- A Grammar with left factoring present is a NON DETERMINISTIC Grammar.
- Top Down Parser will not work with grammar having Left Factoring.
- **Removing Left Factoring:**
- $A \to \alpha\beta_1|\alpha\beta_2$

$$A \to \alpha A'$$
$$A' \to \beta_1|\beta_2$$

Solution:

A->aA'

A'->a|b|c

Left Factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.

Algorithm to left factor a grammar

Input: Grammar G

Output: An equivalent left factored grammar.

Method:

For each non terminal A find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \in$, i.e., there is a non trivial common prefix, replace all the A productions $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \ldots\ldots\ldots\ldots. | \alpha\beta_n | \gamma$ where γ represents all alternatives that do not begin with α by

$$A \rightarrow \alpha A' | \gamma$$
$$A' \rightarrow \beta_1 | \beta_2 | \ldots\ldots\ldots\ldots. |\beta_n$$

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

Example: Left Factoring

- stmt->if expr then stmt else stmt|if expr then stmt

Example: Left Factoring

- stmt->if expr then stmt else stmt|if expr then stmt

- **Elimination:**
- Stmt->if expr then stmt A
- A->else stmt|∈

Questions: Left Factoring

- S->iEtS|iEtSeS|a
- E->b

Questions: Left Factoring

- S->iEtS|iEtSeS|a
- E->b

- **Elimination:**
- S->iEtSS'|a
- S'->eS| ∈
- E->b

Questions: Left Factoring

- X->X+X|X*X|D
- D->1|2|3

Questions: Left Factoring

- X->X+X|X*X|D
- D->1|2|3

- **Elimination:**
- X->XY|D
- Y->+X|*X
- D->1|2|3

Questions: Left Factoring

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

Questions: Left Factoring

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

- **Elimination:**
- E->TE’
- E’->+E| ∈
- T->intT’|(E)
- T’->*T| ∈

Questions: Left Factoring

- S->aSSbS|aSaSb|abb|b

Questions: Left Factoring

- S->aSSbS|aSaSb|abb|b

- **Elimination-1:**
- S->aS’|b
- S’->SSbS|SaSb|bb

Questions: Left Factoring

- S->aSSbS|aSaSb|abb|b

- **Elimination-1:**
- S->aS'|b
- S'->SSbS|SaSb|bb
- **S' Elimination:**
- S'->SS''|bb
- S''->SbS|aSb

Questions: Left Factoring

- S->aSSbS|aSaSb|abb|b

- **Elimination-2:**
- S->aSS’|abb|b
- S’->SbS|aSb

Questions: Left Factoring

- S->aSSbS|aSaSb|abb|b

- **Elimination-2:**
- S->aSS’|abb|b
- S’->SbS|aSb
- **S Elimination:**
- S->aS’’|b
- S’’->SS’|bb

Questions: Left Factoring

- A->aA
- B->aB

Questions: Left Factoring

- A->aA
- B->aB

- **Elimination**
- No Common Non Terminal on LHS then only Left Factoring elimination can perform.

Questions: Left Factoring

- S->aAB|aCD

Questions: Left Factoring

- S->aAB|aCD

- **Elimination:**
- S->aS’
- S’->AB|CD

Questions: Left Factoring

- A->xByA| xByAzA|a

Questions: Left Factoring

- A->xByA| xByAzA|a

- **Elimination:**
- A-> xByAA’|a
- A’->zA|∈

Questions: Left Factoring

- A->aAB | aA |a

Questions: Left Factoring

- A->aAB | aA |a

- **Elimination:**
- A-> aAA’|a
- A’->B|∈

Questions: Left Factoring

- A->aAB | aA |a

- **Elimination:**
- A-> aAA’|a
- A’->B|∈
- **A Elimination:**
- A->aA’’
- A’’->AA’|∈

Questions: Left Factoring

- A->ad | a | ab | abc | x

FIRST()

- If α is any string of grammar symbols then FIRST(α) is the set of terminals that begin the string derived from α.
- If $\alpha \overset{*}{\Rightarrow} \in$ then $\in$ is also in FIRST(α)

Rules to compute first() of non terminal

1. If $A \rightarrow \alpha$ and α is terminal, add α to $FIRST(A)$.
2. If $A \rightarrow \in$, add $\in$ to $FIRST(A)$.
3. If X is nonterminal and $X \rightarrow Y_1\, Y_2 \ldots .Y_k$ is a production, then place a in $FIRST(X)$ if for some i, a is in $FIRST(Yi)$, and ϵ is in all of $FIRST(Y_1), \ldots \ldots \ldots, FIRST(Y_{i-1})$; that is $Y_1 \ldots Y_{i-1} \Rightarrow \epsilon$. If ϵ is in $FIRST(Y_j)$ for all $j = 1,2, \ldots \ldots, k$ then add ϵ to $FIRST(X)$.

   Everything in $FIRST(Y_1)$ is surely in $FIRST(X)$ If Y_1 does not derive ϵ, then we do nothing more to $FIRST(X)$, but if $Y_1 \Rightarrow \epsilon$, then we add $FIRST(Y_2)$ and so on.

Rules to compute first() of non terminal

Simplification of Rule 3

If $A \rightarrow Y_1 Y_2 \ldots \ldots . . Y_K$,

- If Y_1 does not derives ∈ $then, FIRST(A) = FIRST(Y_1)$
- If Y_1 derives ∈ $then,$

  $FIRST(A) = FIRST(Y_1) - \epsilon$ U $FIRST(Y_2)$
- If Y_1 & Y_2 derives ∈ $then,$

  $FIRST(A) = FIRST(Y_1) - \epsilon$ U $FIRST(Y_2) - \epsilon\ U\ FIRST(Y_3)$
- If Y_1 , Y_2 & Y_3 derives ∈ $then,$

  $FIRST(A) = FIRST(Y_1) - \epsilon\ U\ FIRST(Y_2) - \epsilon\ U\ FIRST(Y_3) - \epsilon\ U\ FIRST(Y_4)$
- If Y_1 , Y_2, Y_3 .....Y_K all derives ∈ $then,$

  $FIRST(A) = FIRST(Y_1) - \epsilon\ U\ FIRST(Y_2) - \epsilon\ U\ FIRST(Y_3) - \epsilon\ U\ FIRST(Y_4) - \epsilon\ U \ldots \ldots \ldots \ldots\ FIRST(Y_k)$ (note: if all non terminals derives ∈ then add ∈ to FIRST(A))

Example

- S->ABC|ghi|jkl
- A->a|b|c
- B->b
- D->d

Example

- S->ABC|ghi|jkl
- A->a|b|c
- B->b
- D->d

- Solution:
- FIRST(S) = {a,b,c,g,j}
- FIRST(A) = {a,b,c}
- FIRST(B) = {b}
- FIRST(D) = {d}

Example

- S->ABC
- A->a|b|∈
- B->c|d|∈
- C->e|f|∈

- Solution:
- FIRST(S) = {a,b,c,d,e,f, ∈}
- FIRST(A) = {a,b, ∈}
- FIRST(B) = {c,d, ∈}
- FIRST(C) = {e,f, ∈}

Example

- X->AB
- A->a|∈
- B->b

Example

- X->AB
- A->a|∈
- B->b

- **Answer:**
- First(X) = {a, b}
- First(A) = The first set of A is {a, ∈}
- First(B) = The first set of B is {b}

Example

- X->AB
- A->a|∈
- B->b|∈

Example

- X->AB
- A->a|∈
- B->b|∈

- **Answer:**
- First(X) = {a, b, ∈}
- First(A) = The first set of A is {a, ∈}
- First(B) = The first set of B is {b, ∈}

Example

- E->TE’
- E’->+TE’| ∈
- T->FT’
- T’->*FT’| ∈
- F->id|(E)

- FIRST(F)=?
- FIRST(T’)=?
- FIRST(T)=?
- FIRST(E’)=?
- FIRST(E)=?

Example

- E->TE’
- E’->+TE’| ∈
- T->FT’
- T’->*FT’| ∈
- F->id|(E)

- FIRST(F)={id,(}
- FIRST(T’)={*, ∈}
- FIRST(T)={id,(}
- FIRST(E’)={+, ∈}
- FIRST(E)={id,(}

Example

- S->aABb
- A→c|∈
- B->d|∈

Example

- S->aABb
- A→c|∈
- B->d|∈

- **Solution:**
- FIRST(S) = {a}
- FIRST(A) = {c, ∈}
- FIRST(B) = {d, ∈}

Example

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

Example

- S->aBDh
- B->cC
- C->bC|∈
- D->EF
- E->g|∈
- F->f|∈
- **Solution:**
- FIRST(S)={a}
- FIRST(B) = {c}
- FIRST(C)={b, ∈}
- FIRST(D) = {g, f, ∈}
- FIRST(E) = {g, ∈}
- FIRST(F) = {f, ∈}

Example

- S->Bb|Cd
- B->aB|∈
- C->cC|∈

Example

- S->Bb|Cd
- B->aB|∈
- C->cC|∈

- Solution:
- FIRST(S) = {a, b, c, d}
- FIRST(B) = {a, ∈}
- FIRST(C) = {c, ∈}

Example

- A->da|BC
- S->ACB|CbB|Ba
- B->g|∈
- C->h|∈

Example

- A->da|BC
- S->ACB|CbB|Ba
- B->g|∈
- C->h|∈

- Solution:
- FIRST(A) = {d, g, h, ∈}
- FIRST(S) = {d, g, h,b,a, ∈}
- FIRST(B) = {g, ∈}
- FIRST(C) = {h, ∈}

Example

- S->AB
- A->Ca|∈
- B->BaAC|c
- C->b|∈

Example

- S->AB
- A->Ca|∈
- B->BaAC|c
- C->b|∈

- Solution:
- FIRST(S) = {b,a,c}
- FIRST(A) = {b,a, ∈}
- FIRST(B) = {c}
- FIRST(C) = {b, ∈}

Example

- S->ABCDE
- A->a|∈
- B->b|∈
- C->c
- D->d|∈
- E->e|∈

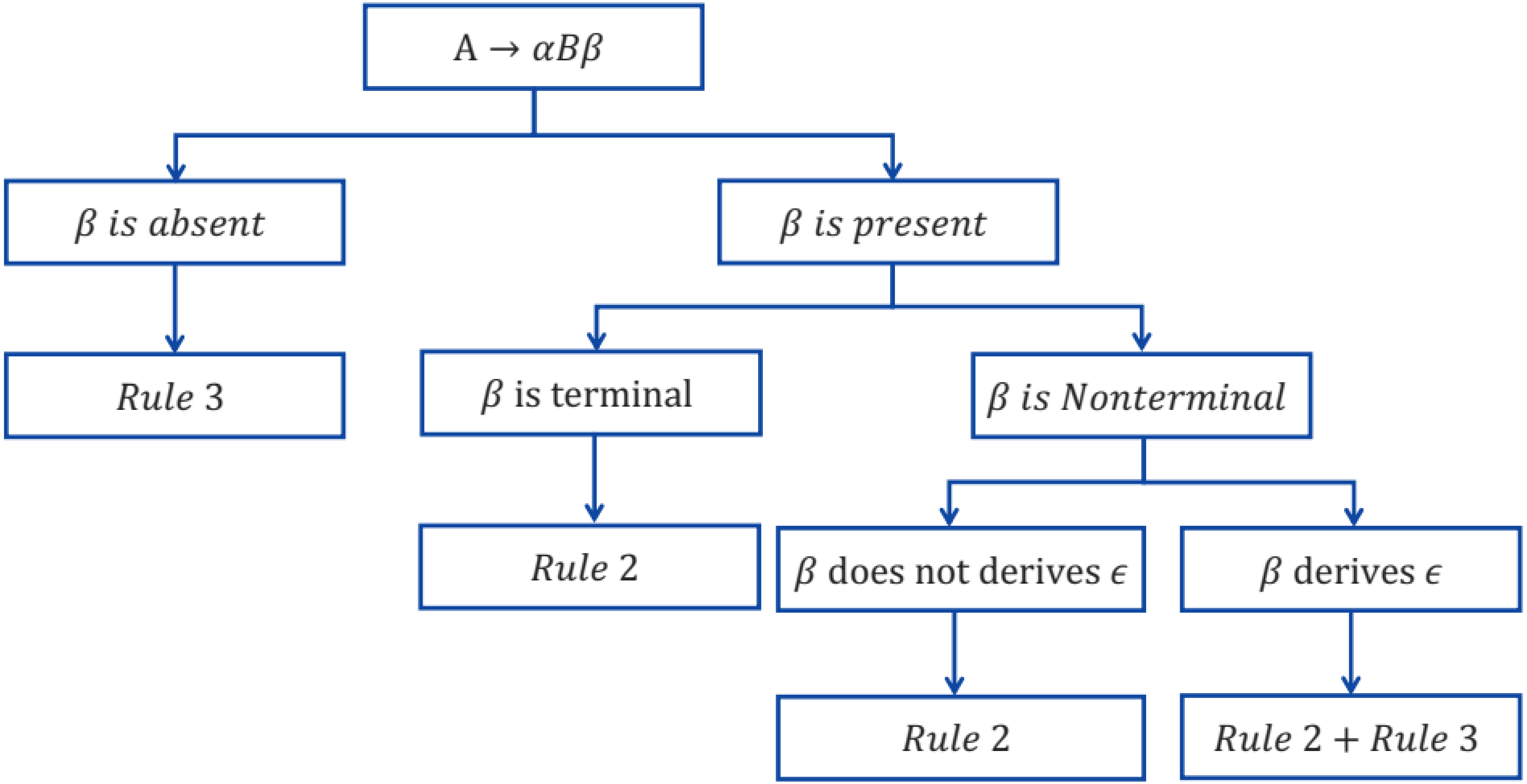
Example

- S->ABCDE
- A->a|∈
- B->b|∈
- C->c
- D->d|∈
- E->e|∈
- Solution:
- FIRST(S) = {a,b,c}
- FIRST(A) = {a, ∈}
- FIRST(B) = {b, ∈}
- FIRST(C) = {c}
- FIRST(D) = {d, ∈}
- FIRST(E) = {e, ∈}

Rules to compute FOLLOW of non terminal

1. Place $\$ \ in \ follow(S)$. (S is start symbol)
2. If $\mathrm{A} \rightarrow \alpha B \beta$, then everything in $FIRST(\beta)$ except for ϵ is placed in $FOLLOW(B)$
3. If there is a production $\mathrm{A} \rightarrow \alpha B$ or a production $\mathrm{A} \rightarrow \alpha B \beta$ where $FIRST(\beta)$ contains ϵ then everything in $FOLLOW(A) = FOLLOW(B)$

How to apply rules to find FOLLOW of non terminal?



Example

- S->AaAb|BbBa
- A->∈
- B->∈

Example

- S->AaAb|BbBa
- A->∈
- B->∈
- Solution:
- Follow(S) = {$}
- Follow(A)={a,b}
- Follow(B) = {b,a}

Example

- S->ABC
- A->DEF
- B->∈
- C->∈
- D->∈
- E->∈
- F->∈

Example

- S->aABb
- A->c|∈
- B->d|∈

Example

- S->aABb
- A->c|∈
- B->d|∈

- Solution:
- Follow(S)={$}
- Follow(A) = {d,b}
- Follow(B) = {b}

Example

- S->aBDh
- B->cC
- C->bc|∈
- D->EF
- E->g|∈
- F->f|∈

Example

- S->aBDh
- B->cC
- C->bC|∈
- D->EF
- E->g|∈
- F->f|∈
- **Solution:**
- Follow(S) = {$}
- Follow(B) = {g,f,h}
- Follow(C) = {g,f,h}
- Follow(D) = {h}
- Follow(E) = {f,h}
- Follow(F) = {h}

Example

- S->Bb|Cd
- B->aB|∈
- C->cC|∈

Example

- S->Bb|Cd
- B->aB|∈
- C->cC|∈
- **Solution:**
- Follow(S) = {$}
- Follow(B) = {b}
- Follow(C) = {d}

Example

- S->ACB|CbB|Ba
- A->da|BC
- B->g|∈
- C->h|∈

Example

- S->ACB|CbB|Ba
- A->da|BC
- B->g|∈
- C->h|∈
- Solution:
- Follow(S) = {$}
- Follow(A) = {$, h,g}
- Follow(B) = {a,h,g,$}
- Follow(C) = {b,g,$,h}

Example

- S->xyz|aBC
- B->c|cd
- C->eg|df

Example

- S->xyz|aBC
- B->c|cd
- C->eg|df

- Solution:
- Follow(S) = {$}
- Follow(B) = {e,d}
- Follow(C) = {$}

Example

- S->ABCDE
- A->a|∈
- B->b|∈
- C->c
- D->d|∈
- E->e|∈
- Solution:

Example

- S->ABCDE
- A->a|∈
- B->b|∈
- C->c
- D->d|∈
- E->e|∈
- Solution:
- Follow(S) = {$}
- Follow(A) = {b,c}
- Follow(B) = {c}
- Follow(C) = {d,e,$}
- Follow(D) = {e,$}
- Follow(E)= {$}

Example

- Calculate the first and follow functions for the given grammar-
  - $E \rightarrow E + T \mid T$
  - $T \rightarrow T * F \mid F$
  - $F \rightarrow (E) \mid id$

Example

- Eliminate Left Recursion:
- E->TE’
- E’->+TE’| ∈
- T->FT’
- T’->*FT’| ∈
- F->id|(E)

Example

- E->TE’
- E’->+TE’| ∈
- T->FT’
- T’->*FT’| ∈
- F->id|(E)

- Follow(E) = {$,)}
- Follow(E’) = Follow(E) = {$,)}
- Follow(T) = {+,$,)}
- Follow(T’) = {+,$,)}
- Follow(F) = {*,+,$,)}

Example

- Consider the following Grammar:
- S->tABCD
- A->qt|t
- B->r|∈
- C->q|∈
- D->p
- What is the Follow(A)?
- A. {r,q,p,t}
- B. {r,q,p}
- C. {r,q,p, ∈}
- D. {r,q,p,$}

Example

- Which of the following is present in FIRST(X) ∩ FIRST(B) of the below given?
- X → A
- A → Bb | Cd
- B → aB | Cd | ∈
- C → Cc | ∈

- A. {a, c, d, ϵ}
- B. {a, c, d, $}
- C. {a, c, d}
- D. {a, c, ϵ}

Example

- Which of the following is present in FIRST(X) ∩ FIRST(B) of the below given?
- X → A
- A → Bb | Cd
- B → aB | Cd | ∈
- C → Cc | ∈

- A. {a, c, d, ϵ}
- B. {a, c, d, $}
- C. {a, c, d}
- D. {a, c, ϵ}
- **Answer: C**

Rules to construct LL(1) or predictive parsing table

1. For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.
2. For each terminal a in $first(\alpha)$, Add $A \rightarrow \alpha$ to $M[A, a]$.
3. If ϵ is in $first(\alpha)$, Add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal b in $FOLLOW(B)$. If ϵ is in $first(\alpha)$, and \$ is in $FOLLOW(A)$, add $A \rightarrow \alpha$ to $M[A, \$]$.
4. Make each undefined entry of M be error.

Grammar for LL(1) Table

- S->(L)|a
- L->SL’
- L’->∈|,L’

- Remove Left recursion if it is there
- Then Find out FIRST & FOLLOW

Grammar for LL(1) Table

- S->(L)|a
- L->SL’
- L’->∈|,L’

- FIRST (S) = {(a}
- FIRST(L) = {(a}
- FIRST (L’) = {∈ ,}

Grammar for LL(1) Table

- S->(L)|a
- L->SL’
- L’->∈|,L’

- FIRST (S) = {(a}
- FIRST(L) = {(a}
- FIRST (L’) = {∈ ,}
- FOLLOW (S) = {$,)}
- FOLLOW (L) = {) }
- FOLLOW (L’) = {) }

Grammar for LL(1) Table

- S->(L)|a
- L->SL’
- L’->∈|,L’

- FIRST (S) = {(a}
- FIRST(L) = {(a}
- FIRST (L’) = {∈ ,}
- FOLLOW (S) = {$,)}
- FOLLOW (L) = {) }
- FOLLOW (L’) = {) }

	(	**)**	**a**	**,**	**$**
S					
L					
L’					

Grammar for LL(1) Table

- S->(L)|a
- L->SL’
- L’->∈|,L’

1. S->(L)
2. S->a
3. L->SL’
4. L’-> ∈
5. L’->,L’

- FIRST (S) = {(a}
- FIRST(L) = {(a}
- FIRST (L’) = {∈ ,}
- FOLLOW (S) = {$,)}
- FOLLOW (L) = {) }
- FOLLOW (L’) = {) }

	(	**)**	**a**	**,**	**$**
S					
L					
L’					

Grammar for LL(1) Table

- S->(L)|a
- L->SL’
- L’->∈|,L’

- FIRST (S) = {(a}
- FIRST(L) = {(a}
- FIRST (L’) = {∈ ,}
- FOLLOW (S) = {$,)}
- FOLLOW (L) = {) }
- FOLLOW (L’) = {) }

1. S->(L)
2. S->a
3. L->SL’
4. L’-> ∈
5. L’->,L’

For First Production rule

	(	**)**	**a**	**,**	**$**
S	1. S->(L)				
L					
L’					

Grammar for LL(1) Table

- S->(L)|a
- L->SL’
- L’->∈|,L’

- FIRST (S) = {(a}
- FIRST(L) = {(a}
- FIRST (L’) = {∈ ,}
- FOLLOW (S) = {$,)}
- FOLLOW (L) = {) }
- FOLLOW (L’) = {) }

1. S->(L)
2. S->a
3. L->SL’
4. L’-> ∈
5. L’->,L’

For Second Production rule

	(	**)**	**a**	**,**	**$**
S	1. S->(L)		2. S->a		
L					
L’					

Grammar for LL(1) Table

- S->(L)|a
- L->SL'
- L'->∈|,SL'

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

1. S->(L)
2. S->a
3. L->SL'
4. L'-> ∈
5. L'->,SL'

For Third Production rule

	(	**)**	**a**	**,**	**$**
S	1. S->(L)		2. S->a		
L	3. L->SL'		3. L->SL'		
L'					

Grammar for LL(1) Table

- S->(L)|a
- L->SL’
- L’->∈|,SL’

- FIRST (S) = {(a}
- FIRST(L) = {(a}
- FIRST (L’) = {∈ ,}
- FOLLOW (S) = {$,)}
- FOLLOW (L) = {) }
- FOLLOW (L’) = {) }

1. S->(L)
2. S->a
3. L->SL’
4. L’-> ∈
5. L’->,SL’

For Fourth Production rule

	(	**)**	**a**	**,**	**$**
S	1. S->(L)		2. S->a		
L	3. L->SL’		3. L->SL’		
L’		4. L’->∈			

Grammar for LL(1) Table

- S->(L)|a
- L->SL’
- L’->∈|,SL’

- FIRST (S) = {(a}
- FIRST(L) = {(a}
- FIRST (L’) = {∈ ,}
- FOLLOW (S) = {$,)}
- FOLLOW (L) = {) }
- FOLLOW (L’) = {) }

1. S->(L)
2. S->a
3. L->SL’
4. L’-> ∈
5. L’->,SL’

For Fifth Production rule

	(	**)**	**a**	**,**	**$**
S	1. S->(L)		2. S->a		
L	3. L->SL’		3. L->SL’		
L’		4. L’->∈		5. L’->,SL’	

Grammar for LL(1) Table

- S->(L)|a
- L->SL’
- L’->∈|,SL’

1. S->(L)
2. S->a
3. L->SL’
4. L’-> ∈
5. L’->,SL’

- FIRST (S) = {(a}
- FIRST(L) = {(a}
- FIRST (L’) = {∈ ,}
- FOLLOW (S) = {$,)}
- FOLLOW (L) = {) }
- FOLLOW (L’) = {) }

Given Grammar is LL(1) Grammar

	(	**)**	**a**	**,**	**$**
S	1. S->(L)		2. S->a		
L	3. L->SL’		3. L->SL’		
L’		4. L’->∈		5. L’->,SL’	

Example

- S -> AaAb | BbBa
- A -> ∈
- B -> ∈

	a	b	$
S			
A			
B			

Example

- S -> AaAb | BbBa
- A -> ∈
- B -> ∈

	a	**b**	**$**
S	S->AaAb	A->BbBa	
A	A->∈	A->∈	
B	B->∈	B->∈	

Example

- S->aBa
- B->bB | ∈

	a	b	$
S			
B			

Example

- S->aBa
- B->bB | ∈

	a	b	$
S	S->aBa		
B	B->∈	B->bB	

Example

- S->aB | ∈
- B->bC | ∈
- C->cS | ∈

Example

- S->aB | ∈
- B->bC | ∈
- C->cS | ∈

	a	**b**	**c**	**$**
S	S->aB			S->∈
B		B->bC		B->∈
C			C->cS	C->∈

Example for LL(1) Table

- S->aSbS|bSaS| ∈

Example for LL(1) Table

- S->aSbS|bSaS| ∈

- First (S) = {a, b, ∈}
- FOLLOW(S) = {b, a, $}

Example for LL(1) Table

- S->aSbS|bSaS| ∈

- First (S) = {a, b, ∈}
- FOLLOW(S) = {b, a, $}

1. S->aSbS
2. S->bSaS
3. S->∈

	a	**b**	**$**
S			

Example for LL(1) Table

- S->aSbS|bSaS| ∈

- First (S) = {a, b, ∈}
- FOLLOW(S) = {b, a, $}

1. S->aSbS
2. S->bSaS
3. S->∈

	a	**b**	**$**
S	1. S->aSbS 3. S->∈	2. S->bSaS 3. S->∈	3. S->∈

Example for LL(1) Table

- S->aSbS|bSaS| ∈

- First (S) = {a, b, ∈}
- FOLLOW(S) = {b, a, $}

1. S->aSbS
2. S->bSaS
3. S->∈

Given Grammar is not LL(1) Grammar

	a	**b**	**$**
S	1. S->aSbS 3. S->∈	2. S->bSaS 3. S->∈	3. S->∈

Short Trick for LL(1) Grammar or not (Only for GATE Exam)

- S->$\alpha_1|\alpha_2|\alpha_3$
- FIRST(α_1) ∩ FIRST(α_2) ∩ FIRST(α_3) = Ø then LL(1) otherwise not LL(1)
- S->$\alpha_1|\alpha_2|\in$
- FIRST(α_1) ∩ FIRST(α_2) ∩ FOLLOW(S) = Ø then LL(1) otherwise not LL(1)

- **Example-1:**
- S->aSa|bS|c

- **Example-2:**
- S -> iCtSS’ |a
- S’ -> eS | ∈
- C -> b

Example

- S -> AB |eDa
- A -> ab |c
- B -> dC
- C -> eC | ∈
- D -> fD | ∈

Example

- S -> AB |eDa
- A -> ab |c
- B -> dC
- C -> eC | ∈
- D -> fD | ∈
- First(S) = {a,c,e}
- First(A) = {a,c}
- First(B) = {d}
- First(C)={e, ∈}
- First(D) = {d, ∈}

Example

- S -> AB |eDa
- A -> ab |c
- B -> dC
- C -> eC | ∈
- D -> fD | ∈
- First(S) = {a,c,e}
- First(A) = {a,c}
- First(B) = {d}
- First(C)={e, ∈}
- First(D) = {d, ∈}

- Follow(S) = {$}
- Follow(A) = {d}
- Follow(B) = {$}
- Follow(C)={$}
- Follow(D) = {a}

Example

- S -> AB |eDa
- A -> ab |c
- B -> dC
- C -> eC | ∈
- D -> fD | ∈
- First(S) = {a,c,e}
- First(A) = {a,c}
- First(B) = {d}
- First(C)={e, ∈}
- First(D) = {d, ∈}

- Follow(S) = {$}
- Follow(A) = {d}
- Follow(B) = {$}
- Follow(C)={$}
- Follow(D) = {a}

	a	b	c	d	e	f	$
S							
A							
B							
C							
D							

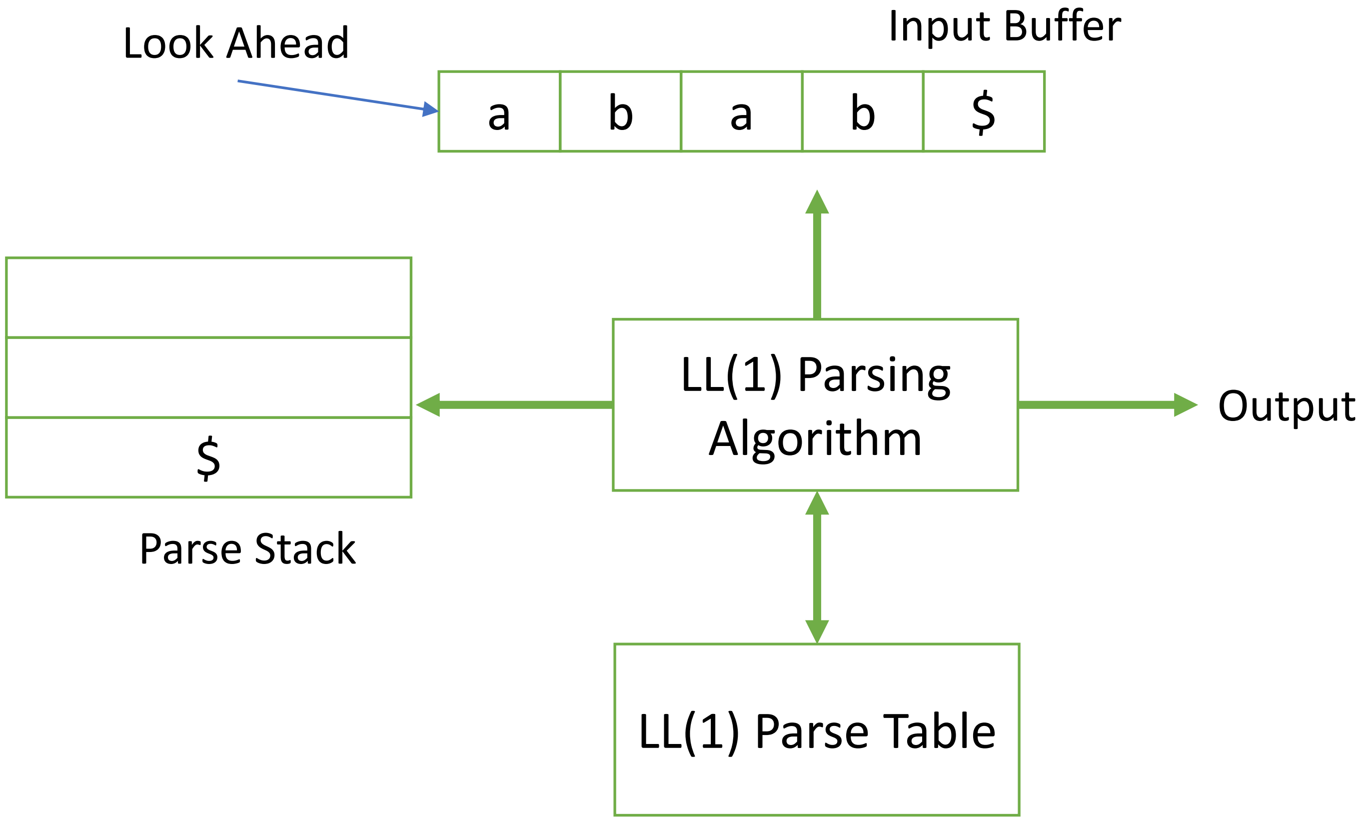
Example

- S -> AB |eDa
- A -> ab |c
- B -> dC
- C -> eC | ∈
- D -> fD | ∈
- First(S) = {a,c,e}
- First(A) = {a,c}
- First(B) = {d}
- First(C)={e, ∈}
- First(D) = {d, ∈}

- Follow(S) = {$}
- Follow(A) = {d}
- Follow(B) = {$}
- Follow(C)={$}
- Follow(D) = {a}

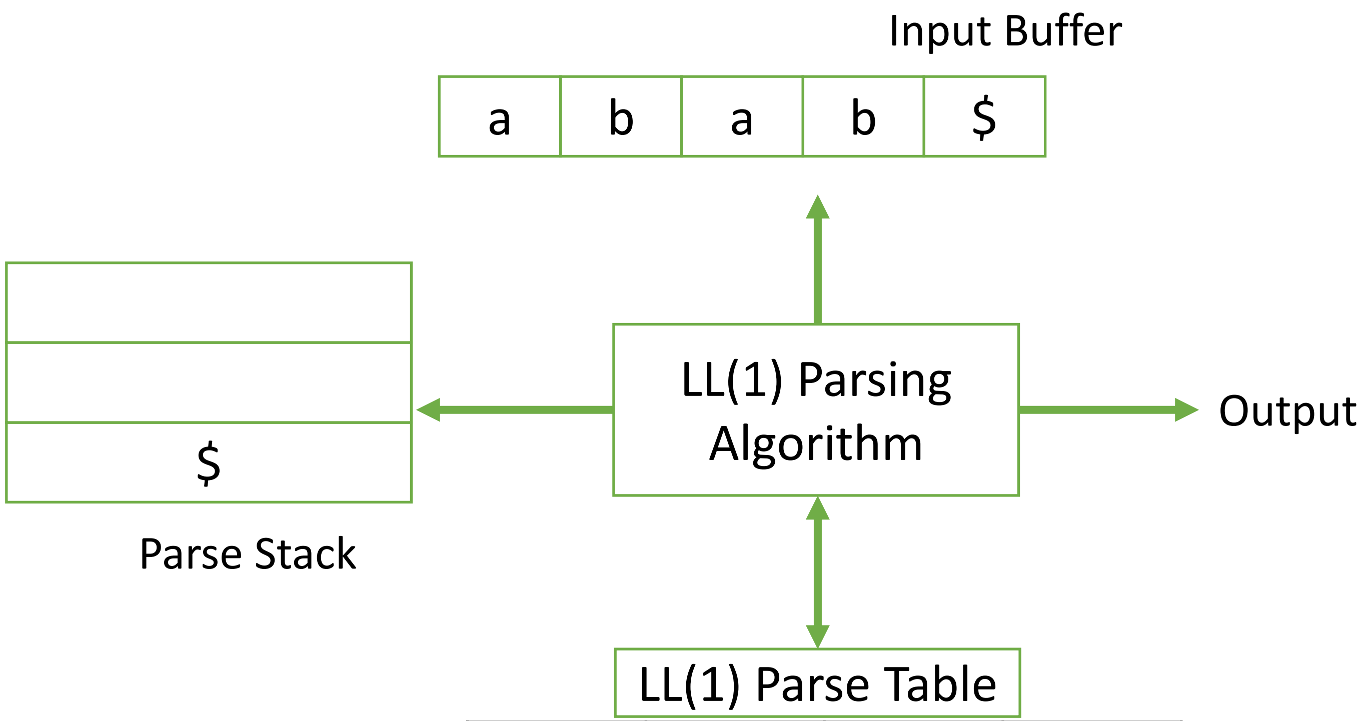
	a	**b**	**c**	**d**	**e**	**f**	**$**
S	S->AB		S->AB		S->eDa		
A	A->ab		A->c				
B				B->dC			
C					C->eC		C->∈
D	D->∈					D->fD	

LL(1) Parser
Left to Right Parser



LL(1) Parser
Left to Right Parser

Input String: abab$



	a	**b**	**$**
S	S->AA	S->AA	
A	A->aA	A->b	

Input Grammar:
S->AA
A->aA|b

LL(1) Parser
Left to Right Parser

Input Grammar:
S->AA
A->aA|b

	a	b	$
S	S->AA	S->AA	
A	A->aA	A->b	

Input String: abab$

Input Buffer

a	b	a	b	$

Stack	Input	Action
$	abab$	Push S into Stack

LL(1) Parser
Left to Right Parser

Input Grammar:
S->AA
A->aA|b

	a	b	$
S	S->AA	S->AA	
A	A->aA	A->b	

Input String: abab$

Input Buffer

a	b	a	b	$

Stack	Input	Action
$	abab$	Push S into Stack
$S	abab$	

LL(1) Parser
Left to Right Parser

Input Grammar:
S->AA
A->aA|b

	a	b	$
S	S->AA	S->AA	
A	A->aA	A->b	

Input String: abab$

Input Buffer

a	b	a	b	$

Stack	Input	Action
$	abab$	Push S into Stack
$S	abab$	S->AA

LL(1) Parser
Left to Right Parser

Input Grammar:
S->AA
A->aA|b

	a	b	$
S	S->AA	S->AA	
A	A->aA	A->b	

Input String: abab$

Input Buffer

a	b	a	b	$

Stack	Input	Action
$	abab$	Push S into Stack
$S	abab$	S->AA
$AA	abab$	

LL(1) Parser
Left to Right Parser

Input Grammar:
S->AA
A->aA|b

	a	b	$
S	S->AA	S->AA	
A	A->aA	A->b	

Input String: abab$

Input Buffer

a	b	a	b	$

Stack	Input	Action
$	abab$	Push S into Stack
$S	abab$	S->AA
$AA	abab$	A->aA

LL(1) Parser
Left to Right Parser

Input Grammar:
S->AA
A->aA|b

	a	b	$
S	S->AA	S->AA	
A	A->aA	A->b	

Input String: abab$

Input Buffer

a	b	a	b	$

Stack	Input	Action
$	abab$	Push S into Stack
$S	abab$	S->AA
$AA	abab$	A->aA
$AAa	abab$	

LL(1) Parser
Left to Right Parser

Input Grammar:
S->AA
A->aA|b

	a	b	$
S	S->AA	S->AA	
A	A->aA	A->b	

Input String: abab$

Input Buffer

a	b	a	b	$

Stack	Input	Action
$	abab$	Push S into Stack
$S	abab$	S->AA
$AA	abab$	A->aA
$AAa	abab$	Pop a

LL(1) Parser
Left to Right Parser

Input Grammar:
S->AA
A->aA|b

	a	b	$
S	S->AA	S->AA	
A	A->aA	A->b	

Input String: abab$

Input Buffer

a	b	a	b	$

Stack	Input	Action
$	abab$	Push S into Stack
$S	abab$	S->AA
$AA	abab$	A->aA
$AAa	abab$	Pop a
$AA	bab$	

LL(1) Parser
Left to Right Parser

Input Grammar:
S->AA
A->aA|b

	a	b	$
S	S->AA	S->AA	
A	A->aA	A->b	

Input String: abab$

Input Buffer

a	b	a	b	$

Stack	Input	Action
$	abab$	Push S into Stack
$S	abab$	S->AA
$AA	abab$	A->aA
$AAa	abab$	Pop a
$AA	bab$	A->b

LL(1) Parser
Left to Right Parser

Input Grammar:
S->AA
A->aA|b

	a	b	$
S	S->AA	S->AA	
A	A->aA	A->b	

Input String: abab$

Input Buffer

a	b	a	b	$

Stack	Input	Action
$	abab$	Push S into Stack
$S	abab$	S->AA
$AA	abab$	A->aA
$AAa	abab$	Pop a
$AA	bab$	A->b
$Ab	bab$	

LL(1) Parser
Left to Right Parser

Input Grammar:
S->AA
A->aA|b

	a	b	$
S	S->AA	S->AA	
A	A->aA	A->b	

Input String: abab$

Input Buffer

a	b	a	b	$

Stack	Input	Action
$	abab$	Push S into Stack
$S	abab$	S->AA
$AA	abab$	A->aA
$AAa	abab$	Pop a
$AA	bab$	A->b
$Ab	bab$	Pop b
$A	ab$	

LL(1) Parser
Left to Right Parser

Input Grammar:
S->AA
A->aA|b

	a	b	$
S	S->AA	S->AA	
A	A->aA	A->b	

Input String: abab$

Input Buffer

a	b	a	b	$

Stack	Input	Action
$	abab$	Push S into Stack
$S	abab$	S->AA
$AA	abab$	A->aA
$AAa	abab$	Pop a
$AA	bab$	A->b
$Ab	bab$	Pop b
$A	ab$	A->aA

LL(1) Parser
Left to Right Parser

Input Grammar:
S->AA
A->aA|b

	a	b	$
S	S->AA	S->AA	
A	A->aA	A->b	

Input String: abab$

Input Buffer

a	b	a	b	$

Stack	Input	Action
$	abab$	Push S into Stack
$S	abab$	S->AA
$AA	abab$	A->aA
$AAa	abab$	Pop a
$AA	bab$	A->b
$Ab	bab$	Pop b
$A	ab$	A->aA
$Aa	ab$	

LL(1) Parser
Left to Right Parser

Input Grammar:
S->AA
A->aA|b

	a	b	$
S	S->AA	S->AA	
A	A->aA	A->b	

Input String: abab$

Input Buffer

a	b	a	b	$

Stack	Input	Action
$	abab$	Push S into Stack
$S	abab$	S->AA
$AA	abab$	A->aA
$AAa	abab$	Pop a
$AA	bab$	A->b
$Ab	bab$	Pop b
$A	ab$	A->aA
$Aa	ab$	Pop a

LL(1) Parser
Left to Right Parser

Input Grammar:
S->AA
A->aA|b

	a	b	$
S	S->AA	S->AA	
A	A->aA	A->b	

Input String: abab$

Input Buffer

a	b	a	b	$

Stack	Input	Action
$	abab$	Push S into Stack
$S	abab$	S->AA
$AA	abab$	A->aA
$AAa	abab$	Pop a
$AA	bab$	A->b
$Ab	bab$	Pop b
$A	ab$	A->aA
$Aa	ab$	Pop a
$A	b$	

LL(1) Parser
Left to Right Parser

Input Grammar:
S->AA
A->aA|b

	a	b	$
S	S->AA	S->AA	
A	A->aA	A->b	

Input String: abab$

Input Buffer

a	b	a	b	$

Stack	Input	Action
$	abab$	Push S into Stack
$S	abab$	S->AA
$AA	abab$	A->aA
$AAa	abab$	Pop a
$AA	bab$	A->b
$Ab	bab$	Pop b
$A	ab$	A->aA
$Aa	ab$	Pop a
$A	b$	A->b

LL(1) Parser
Left to Right Parser

Input Grammar:
S->AA
A->aA|b

	a	b	$
S	S->AA	S->AA	
A	A->aA	A->b	

Input String: abab$

Input Buffer

a	b	a	b	$

Stack	Input	Action
$	abab$	Push S into Stack
$S	abab$	S->AA
$AA	abab$	A->aA
$AAa	abab$	Pop a
$AA	bab$	A->b
$Ab	bab$	Pop b
$A	ab$	A->aA
$Aa	ab$	Pop a
$A	b$	A->b
$b	b$	

LL(1) Parser
Left to Right Parser

Input Grammar:
S->AA
A->aA|b

	a	b	$
S	S->AA	S->AA	
A	A->aA	A->b	

Input String: abab$

Input Buffer

a	b	a	b	$

Stack	Input	Action
$	abab$	Push S into Stack
$S	abab$	S->AA
$AA	abab$	A->aA
$AAa	abab$	Pop a
$AA	bab$	A->b
$Ab	bab$	Pop b
$A	ab$	A->aA
$Aa	ab$	Pop a
$A	b$	A->b
$b	b$	Pop b

LL(1) Parser
Left to Right Parser

Input Grammar:
S->AA
A->aA|b

	a	b	$
S	S->AA	S->AA	
A	A->aA	A->b	

Input String: abab$

Input Buffer

a	b	a	b	$

Stack	Input	Action
$	abab$	Push S into Stack
$S	abab$	S->AA
$AA	abab$	A->aA
$AAa	abab$	Pop a
$AA	bab$	A->b
$Ab	bab$	Pop b
$A	ab$	A->aA
$Aa	ab$	Pop a
$A	b$	A->b
$b	b$	Pop b
$	$	

LL(1) Parser
Left to Right Parser

Input Grammar:
S->AA
A->aA|b

	a	b	$
S	S->AA	S->AA	
A	A->aA	A->b	

Input String: abab$

Input Buffer

a	b	a	b	$

Stack	Input	Action
$	abab$	Push S into Stack
$S	abab$	S->AA
$AA	abab$	A->aA
$AAa	abab$	Pop a
$AA	bab$	A->b
$Ab	bab$	Pop b
$A	ab$	A->aA
$Aa	ab$	Pop a
$A	b$	A->b
$b	b$	Pop b
$	$	Accept

Example for LL(1)

- S -> aABb
- A -> c | ∈
- B -> d | ∈

Example for LL(1)

- S -> aABb
- A -> c | ∈
- B -> d | ∈

- FIRST(S) = {a}
- FIRST(A) = {c, ∈}
- FIRST(B) = {d, ∈}

Example for LL(1)

- S -> aABb
- A -> c | ∈
- B -> d | ∈

- FIRST(S) = {a}
- FIRST(A) = {c, ∈}
- FIRST(B) = {d, ∈}

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

Example for LL(1) Parser Table

- S -> aABb
- A -> c | ∈
- B -> d | ∈
- FIRST(S) = {a}
- FIRST(A) = {c, ∈}
- FIRST(B) = {d, ∈}
- FOLLOW(S) = {$}
- FOLLOW(A) = {d, b}
- FOLLOW(B) = {b}

1. **S -> aABb**
2. **A -> c**
3. **A -> ∈**
4. **B -> d**
5. **B -> ∈**

	a	b	c	d	$
S					
A					
B					

Example for LL(1) Parser Table

- S -> aABb
- A -> c | ∈
- B -> d | ∈
- FIRST(S) = {a}
- FIRST(A) = {c, ∈}
- FIRST(B) = {d, ∈}
- FOLLOW(S) = {$}
- FOLLOW(A) = {d, b}
- FOLLOW(B) = {b}

1. **S -> aABb**
2. **A -> c**
3. **A ->** ∈
4. **B -> d**
5. **B ->** ∈

	a	b	c	d	$
S	1. S -> aABb				
A		3. A->∈	2. A->c	3. A->∈	
B		5. B->∈		4. B->d	

Example for LL(1) Parser

	a	b	c	d	$
S	1. S -> aABb				
A		3. A->∈	2. A->c	3. A->∈	
B		5. B->∈		4. B->d	

- S -> aABb
- A -> c | ∈
- B -> d | ∈
- FIRST(S) = {a}
- FIRST(A) = {c, ∈}
- FIRST(B) = {d, ∈}
- FOLLOW(S) = {$}
- FOLLOW(A) = {d, b}
- FOLLOW(B) = {b}

1. **S -> aABb**
2. **A -> c**
3. **A -> ∈**
4. **B -> d**
5. **B -> ∈**

Example String: acdb$

Stack	Input	Action
$	acdb$	Push S into Stack
$S	acdb$	1. S->aABb
$bBAa	acdb$	Pop a
$bBA	cdb$	2. A->c
$bBc	cdb$	Pop c
$bB	db$	B->d
$bd	db$	Pop d
$b	b$	Pop b
$	$	Accept

Example

- E->E+T | T
- T->T*F | F
- F->(E) | id

Example

- E->E+T | T
- T->T*F | F
- F->(E) | id

- Remove Left Recursion:
- E->TE‘
- E‘->+TE‘| ∈
- T->FT’
- T’->*FT’| ∈
- F->(E)|id

Example

- E->E+T | T
- T->T*F | F
- F->(E) | id

- Remove Left Recursion:
- E->TE‘
- E‘->+TE‘| ∈
- T->FT’
- T’->*FT’| ∈
- F->(E)|id

NT	FIRST	FOLLOW
E		
E’		
T		
T’		
F		

Example

- E->E+T | T
- T->T*F | F
- F->(E) | id

- Remove Left Recursion:
- E->TE‘
- E‘->+TE‘| ∈
- T->FT’
- T’->*FT’| ∈
- F->(E)|id

NT	FIRST	FOLLOW
E	{(,id}	
E’	{+, ∈}	
T	{(,id}	
T’	{*, ∈}	
F	{(,id}	

Example

- E->E+T | T
- T->T*F | F
- F->(E) | id

- Remove Left Recursion:
- E->TE‘
- E‘->+TE‘| ∈
- T->FT’
- T’->*FT’| ∈
- F->(E)|id

NT	FIRST	FOLLOW
E	{(,id}	{$,)}
E’	{+, ∈}	{$,)}
T	{(,id}	{+,$,)}
T’	{*, ∈}	{+,$,)}
F	{(,id}	{*, +,$,)}

Example

- E->E+T | T
- T->T*F | F
- F->(E) | id

NT	FIRST	FOLLOW
E	{(,id}	{$,)}
E’	{+, ∈}	{$,)}
T	{(,id}	{+,$,)}
T’	{*, ∈}	{+,$,)}
F	{(,id}	{*, +,$,)}

- Remove Left Recursion:
- E->TE‘
- E‘->+TE‘| ∈
- T->FT’
- T’->*FT’| ∈
- F->(E)|id

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

Example

- E->E+T | T
- T->T*F | F
- F->(E) | id

- Remove Left Recursion:
- E->TE‘
- E‘->+TE‘| ∈
- T->FT’
- T’->*FT’| ∈
- F->(E)|id

	id	+	*	(	)	$
E	E->TE’	**Error**	**Error**	E->TE’	**Error**	**Error**
E’	**Error**	E’->+TE’	**Error**	**Error**	E’->∈	E’->∈
T	T->FT’	**Error**	**Error**	T->FT’	**Error**	**Error**
T’	**Error**	T’->∈	T’->*FT’	**Error**	T’->∈	T’->∈
F	F->id	**Error**	**Error**	F->(E)	**Error**	**Error**

Example String: id+id*id$

Stack	Input	Action
$	id+id*id$	Push E into Stack
$E	id+id*id$	

Select()

- SELECT(A → α) = FIRST(α) **if α is not nullable**
- SELECT(A → α) = FIRST(α) U FOLLOW(A) **if α is nullable**

Recursive descent parser

- A recursive descent parser is a top down parser built from a set of mutually recursive procedures (or a non recursive equivalent) where each such procedure implements one of the non-terminals of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognizes.

Recursive descent parser

Recursive-Descent Parsing

void $A()$ {
    Choose an A-production, $A \to X_1 X_2 \cdots X_k$;
    for ($i = 1$ to k) {
        if (X_i is a nonterminal)
            call procedure $X_i()$;
        else if (X_i equals the current input symbol a)
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}

A typical procedure for a nonterminal in a top-down parser

Recursive descent parser

Recursive-Descent Parsing

void $A()$ {
    Choose an A-production, $A \to X_1 X_2 \cdots X_k$;
    for ($i = 1$ to k) {
        if (X_i is a nonterminal)
            call procedure $X_i()$;
        else if (X_i equals the current input symbol a)
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}

Example 4.29 : Consider the grammar

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

To construct a parse tree top-down for the input string $w = cad$, begin with a tree consisting of a single node labeled S, and the input pointer pointing to c,

Recursive descent parser

Example 4.29 : Consider the grammar

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

To construct a parse tree top-down for the input string $w = cad$, begin with a tree consisting of a single node labeled S, and the input pointer pointing to c,

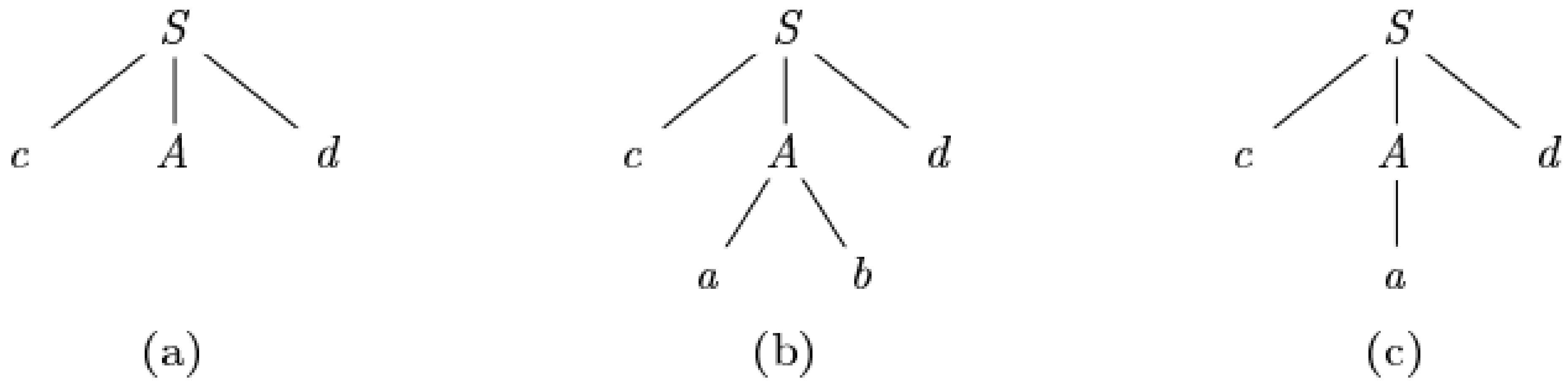


Figure 4.14: Steps in a top-down parse

Recursive descent parser

- E->iE’
- E’->+iE’| ∈

Recursive descent parser

- E->iE’
- E’->+iE’| ∈

```
E(){
    if(look_ahead=='i’){
        look_ahead++;
       EPrime();
    }
    else
       “Error”

}

EPrime(){
    if(look_ahead == '+'){
        if(look_ahead == 'i'){
            look_ahead++;
            EPrime();
        }
    }
    else
        return;
}
```

Recursive descent parser

Input String: i+i$

- E->iE’
- E’->+iE’| ∈

```
E(){
    if(look_ahead=='i’){
        look_ahead++;
       EPrime();
    }
    else
       “Error”

}

EPrime(){
    if(look_ahead == '+'){
        if(look_ahead == 'i'){
            look_ahead++;
            EPrime();
        }
    }
    else
        return;
}
```

Input String: i+i$

Recursive descent parser

- E->iE’
- E’->+iE’| ∈

```
E(){
    if(look_ahead=='i’){
        look_ahead++;
       EPrime();
    }
    else
       “Error”
}

EPrime(){
    if(look_ahead == '+’){
        look_ahead++;
        if(look_ahead == 'i'){
            look_ahead++;
            EPrime();
        }
    }
    else
        return;
}
```

```
void main(){
    E();
    if(look_ahead == '$')
        String is Accepted
    else
        String is not Accepted
}
```

Example-Recursive descent parser

- E->TE’
- E’->+TE’|∈
- T->FT’
- T’->*FT’|∈
- F->(E)|id

Example-Recursive descent parser

- E->TE’
- E’->+TE’|∈
- T->FT’
- T’->*FT’|∈
- F->(E)|id

```
E(){
    T();
    EPrime();
}

EPrime(){
    if(look_ahead == '+'){
        look_ahead++;
        T();
        EPrime();

    }
    else
        return;
```

```
T(){
    F();
    TPrime();
}
TPrime(){
    if(look_ahead == '*'){
        input++;
        F();
        TPrime();

    }
    else
        return;
```

```
F(){
    if(look_ahead == '('){
        look_ahead++;
        E();
        if(look_ahead == ')')
            look_ahead++;
    }
    else if(look_ahead == 'id')
        look_ahead++;
    else
        “error”
}
```

```
void main(){
    E();
    if(input == '$')
        String is Accepted
    else
        String is not Accepted
}
```

Example-Recursive descent parser

- E->TE’
- E’->+TE’|∈
- T->FT’
- T’->*FT’|∈
- F->(E)|id

Input String: id+id$

```
E(){
    T();
    EPrime();
}

EPrime(){
    if(look_ahead == '+'){
            look_ahead++;
            T();
            EPrime();

    }
    else
        return;
```

```
T(){
    F();
    TPrime();
}
TPrime(){
    if(look_ahead == '*'){
            input++;
            F();
            TPrime();

    }
    else
        return;
```

```
void main(){
    E();
    if(input == '$')
        String is Accepted
    else
        String is not Accepted
}
```

```
F(){
    if(look_ahead == '('){
            look_ahead++;
            E();
            if(look_ahead == ')')
                look_ahead++;
    }
    else if(look_ahead == 'id')
        look_ahead++;
    else
        error();
}
```

Example-Recursive descent parser

- E->TE’
- E’->+TE’|∈
- T->FT’
- T’->*FT’|∈
- F->(E)|id

```
E(){
    T();
    EPrime();
}

EPrime(){
    if(look_ahead == '+'){
        look_ahead++;
        T();
        EPrime();

    }
    else
        return;
```

Input String: id+id*id$

```
T(){
    F();
    TPrime();
}
TPrime(){
    if(look_ahead == '*'){
        input++;
        F();
        TPrime();

    }
    else
        return;
```

```
F(){
    if(look_ahead == '('){
        look_ahead++;
        E();
        if(look_ahead == ')')
            look_ahead++;
    }
    else if(look_ahead == 'id')
        look_ahead++;
    else
        “error”
}
```

```
void main(){
    E();
    if(input == '$')
        String is Accepted
    else
        String is not Accepted
}
```

Example-Recursive descent parser

- S->(L)|a
- L->L,S|S

- Verify acceptability of below String:
- (a,(a,a))
- (a,((a,a),(a,a))

Example-Recursive descent parser

- S->(L)|a
- L->L,S|S

- Eliminate Left Recursion

Example-Recursive descent parser

- S->(L)|a
- L->L,S|S

- Eliminate Left Recursion
- S->(L)|a
- L->SL’
- L’->,SL’| ∈

Example-Recursive descent parser

- S->(L)|a
- L->SL’
- L’->,SL’| ∈

- Verify acceptability of below String:
- (a,(a,a))
- (a,((a,a),(a,a))

Example-Recursive descent parser

- S->(L)|a
- L->SL’
- L’->,SL’| ∈

```
S(){
    if(look_ahead == '('){
        look_ahead++;
        L();
        if(look_ahead == ")")
            look_ahead++;
        else
                error();
    }
    else if(look_ahead == 'a')
        look_ahead++;
    else
        error();
}
L(){
    S();
    LPrime();
}
```

```
void main(){
    S();
    if(look_ahead == '$')
        String is Accepted
    else
        String is not Accepted
}

LPrime(){
    if(look_ahead == ','){
            look_ahead++;
            S();
            LPrime();

    }
    else
        return;
}
```

Example-Recursive descent parser

- S->(L)|a
- L->SL’
- L’->,SL’| ∈

```
S(){
    if(look_ahead == '('){
        look_ahead++;
        L();
        if(look_ahead == ")")
            look_ahead++;
        else
                error();
    }
    else if(look_ahead == 'a')
        look_ahead++;
    else
        error();
}
L(){
    S();
    LPrime();
}
```

Input String: (a,(a,a))$

```
void main(){
    S();
    if(look_ahead == '$')
        String is Accepted
    else
        String is not Accepted
}

LPrime(){
    if(look_ahead == ','){
            look_ahead++;
            S();
            LPrime();

    }
    else
        return;
}
```

Example-Recursive descent parser

- S->(L)|a
- L->SL’
- L’->,SL’| ∈

Input String: (a,((a,a),(a,a))$

```
S(){
    if(look_ahead == '('){
        look_ahead++;
        L();
        if(look_ahead == ")")
            look_ahead++;
         else
                error();
    }
    else if(look_ahead == 'a')
        look_ahead++;
   else
       error();
}
L(){
    S();
    LPrime();
}
```

```
void main(){
    S();
    if(look_ahead == '$')
        String is Accepted
    else
        String is not Accepted
}

LPrime(){
    if(look_ahead == ','){
            look_ahead++;
            S();
            LPrime();

    }
    else
        return;
}
```

Example

- S->aAB|bB
- A->aA|b
- B->b

Example

- $S \rightarrow rXd \mid rZd$
- $X \rightarrow oa \mid ea$
- $Z \rightarrow ai$

Example

- S->Aa
- A->BD
- B->b|∈
- D->d|∈

Example

- S-> A
- A -> BC | x
- B -> t | ε
- C -> v | ε

Example

- S-> A
- A -> BC | x
- B -> t
- C -> v

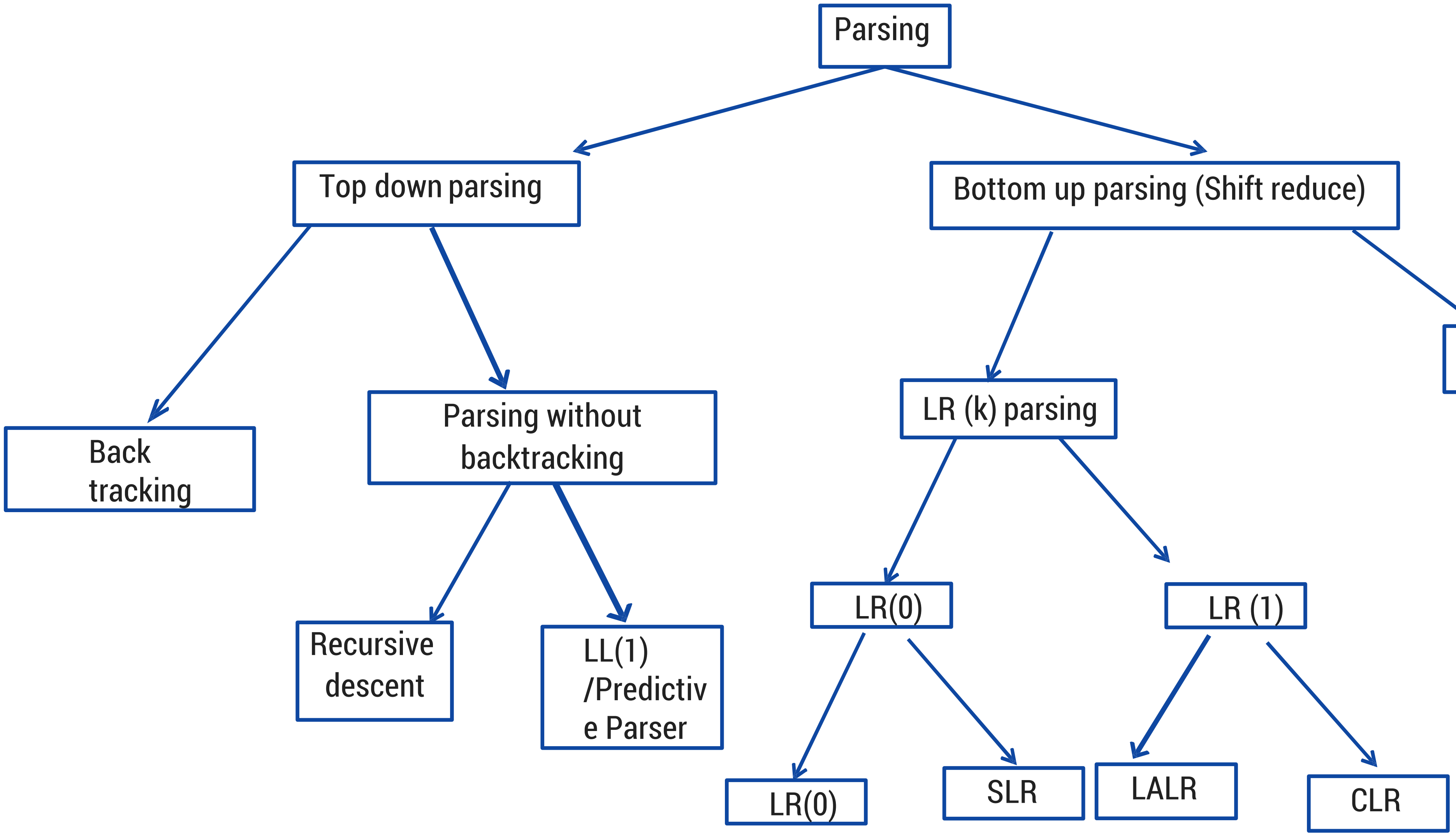
- $E \rightarrow T\ X$
- $X \rightarrow +\ E \mid \varepsilon$
- $T \rightarrow (\ E\ ) \mid \text{int}\ Y$
- $Y \rightarrow *\ T \mid \varepsilon$

Example

expr	→ *term* { *add-op* *term* }
term	→ *factor* { *mult-op* *factor* }
factor	→ `(` *expr* `)` | *number*
add-op	→ `+` | `-`
mult-op	→ `*` | `DIV` | `REM`
number	→ `0` | *nz-digit* { `0` | *nz-digit* }
nz-digit	→ `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

Input String: 4 + 29 DIV 3

Bottom up Parser



LL – Left to Right & Left Most Derivation

LR – Left to Right & Right Most Derivation

Bottom up Parser (Shift Reduce Parser)

- It is the process of reducing the input string to start symbol i.e. the parse tree is constructed in from leaves to the root (bottom to top)
- It is also known as shift reduce Parsing.
- Shift means push into stack
- Reduce means pop from stack
- Also called as LR Parser

Shift Reduce Parser

- Left to Right Scanning
- Right Most derivation
- E->E+E|E*E|id
- Input String: id*id+id
- At each reduction step, a particular substring matching the right side of a production is replaced by the symbol on the left of that production and if the substring is chosen correctly at each step a right most derivation is traced in reverse.

Handle

- S->aABe
- A->Abc|b
- B->d
- Handles: A handle of a string is a substring that matches the right side of a production and whose reduction to the non terminal on the left side of the production represents one step along the reverse of a rightmost derivation.
- Is left most substring always handle? No, choosing the left most substring as the handle always, may not give correct SR parsing.
- A handle of a right sentential form Y is a production A->B and a position of Y where the string B may be found and replaced by A to produce the previous right sentential form in a rightmost derivation of Y.
- Example String: abbcde

- S->aABe
- A->Abc|b
- B->d
- Example String: abbcde
- abbcde: Y=abbcde, A->b, Handle=b
- aAbcde: Y=aAbcde, A->Abc, Handle=Abc
- aAde: Y=aAde, B->d, Handle=d
- aABe: Y=aABe, S->aABe, Handle = aABe
- S

Handle the Pruning

- Removing the children of Left Hand side non terminal from the parse tree is called as Handle Pruning.
- A rightmost derivation in reverse can be obtained by Handle Pruning.
- Steps to follow:
- Start with a string of terminals ‘w’ that is to be parsed.
- Let w = Yn, where Yn is the nth right sentential form of an unknown RMD.
- To reconstruct the RMD in reverse, locate handle Bn in Yn; Replace Bn by LHS by some An->Bn to get (n-1) th RSF Yn-1, Repeat.
- S=>Y0=>Y1=>…=>Yn-1=>Yn

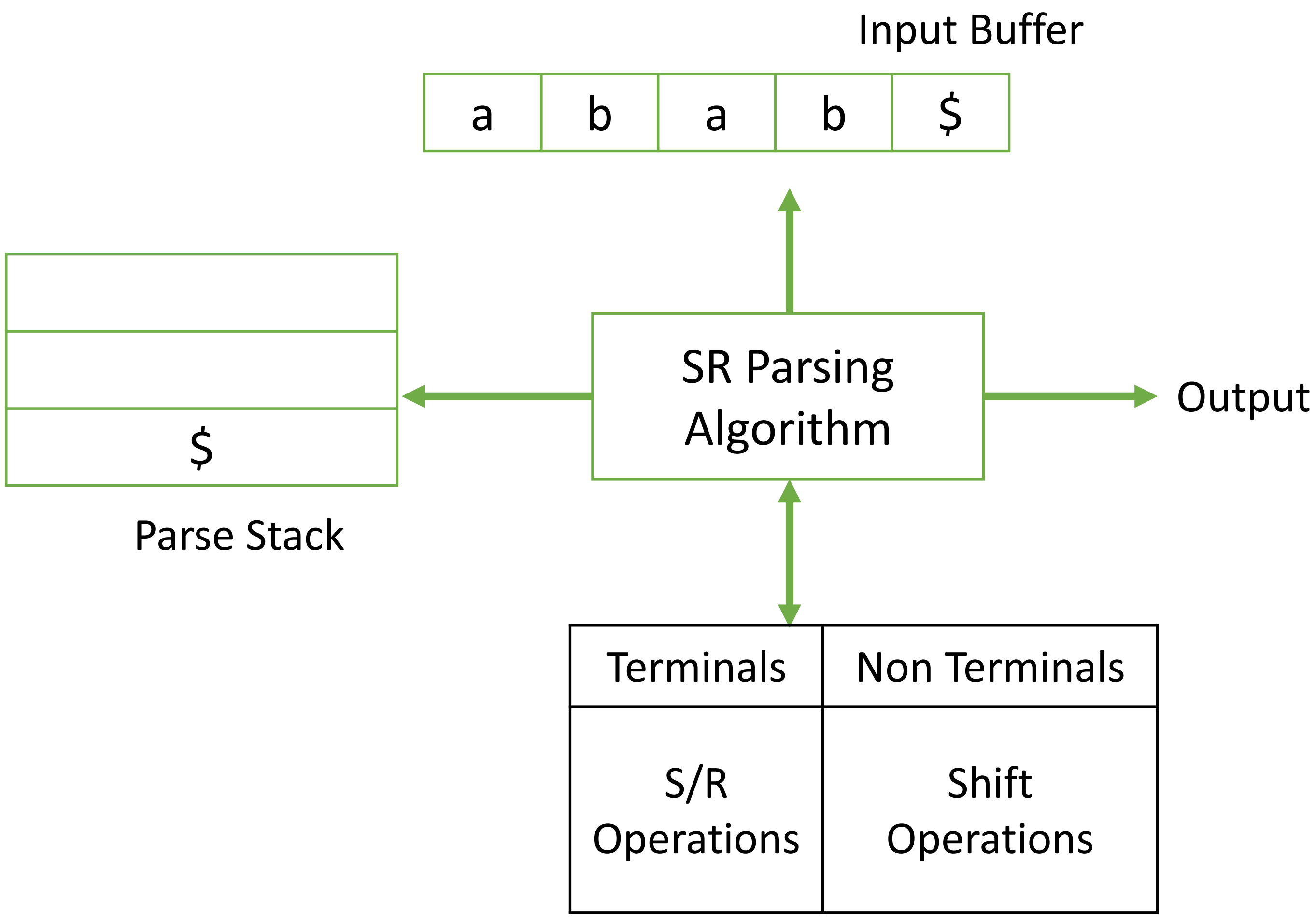
Example of Handle Pruning

- E->E+E|E*E|id

Right Sentential Form(RSF)	Handle	Reducing Production
Id1+id2*id3	Id1	E->id
E+id2*id3	Id2	E->id
E+E*id3	Id3	E->id
E+E*E	E+E	E->E+E
E*E	E*E	E->E*E
E		

- S->aA
- A->bc
- Input: abc$

SR Parser


Input Buffer
a
b
a
b
$
$
Parse Stack
SR Parsing Algorithm
Output
Terminals
Non Terminals
S/R Operations
Shift Operations

Performing SR Parsing using a Stack

- Major data structure used by SR Parsing are:
  - Stack: It is used to hold grammar symbols.
  - Input Buffer: Holds the input string that needs to be parsed.
- Major actions performed are:
  - SHIFT: Pushing the next input symbol on the top of the stack
  - REDUCE: Popping the handle whose right end is at Top Of the Stack and replacing it with left side non terminal.
  - ACCEPT
  - ERROR
- Stack Implementation of SR Parser:
  - Shift input symbols onto the stack until a handle B is on top of stack.
  - Reduce B to left side Non terminal appropriate production
  - Repeat until error or stack has the start symbol left and input is empty.

Example of SR Parsing using a stack

- E->E+E|E*E|id

Stack Content	**Input**	**Action**
$	Id1+id2*id3$	shift
$id1	+id2*id3	Reduce by E->id
$E	+id2*id3	shift
$E+	Id2*id3$	shift
$E+id2	*id3$	Reduce by E->id
$E+E	*id3$	shift

Example of SR Parsing using a stack

- E->E+E|E*E|id

Stack Content	**Input**	**Action**
$	Id1+id2*id3$	shift
$id1	+id2*id3	Reduce by E->id
$E	+id2*id3	shift
$E+	Id2*id3$	shift
$E+id2	*id3$	Reduce by E->id
$E+E	*id3$	Reduced by E->E+E
$E	*id3$	Shift
$E*	Id3$	Shift
$E*id3	$	Reduce by E->id
$E*E	$	Reduce by E->E*E
$E	$	**Accept**

Conflicts of SR parser

- Why use stack for SR Parsing
- Any Handle will always appear on the top of the stack and the parser need not search within the stack at any times.
- Conflict In SR Parsing
- 2 decisions decide a successful SR parsing
  - Locate the substring to reduce
  - Which production to choose when multiple productions with the selected substring on RHS exist.
- SR parser may reach a configuration in which knowing the contents of stack and input buffer, still the parser can not decide.
- Whether to perform a shift or a reduce operations (Shift-Reduce Conflicts)
- Which out of the several reductions to make (Reduce – Reduce Conflicts)

EXAMPLE Conflicts: Scenario-1

- E->E+T|T
- T->T*F|F
- F->(E)|x

Stack Content	Input	Action
$	x*x$	shift
$x	*x$	Reduce F->x
$F	*x$	Reduce T->F
$T	*x$	Shift
$T*	x$	Shift
$T*x	$	Reduce F->x
$T*F	$	Reduce T->T*F
$T	$	Reduce E->T
$E	$	**Accept**

EXAMPLE Conflicts: Scenario-2

- E->E+T|T
- T->T*F|F
- F->(E)|x

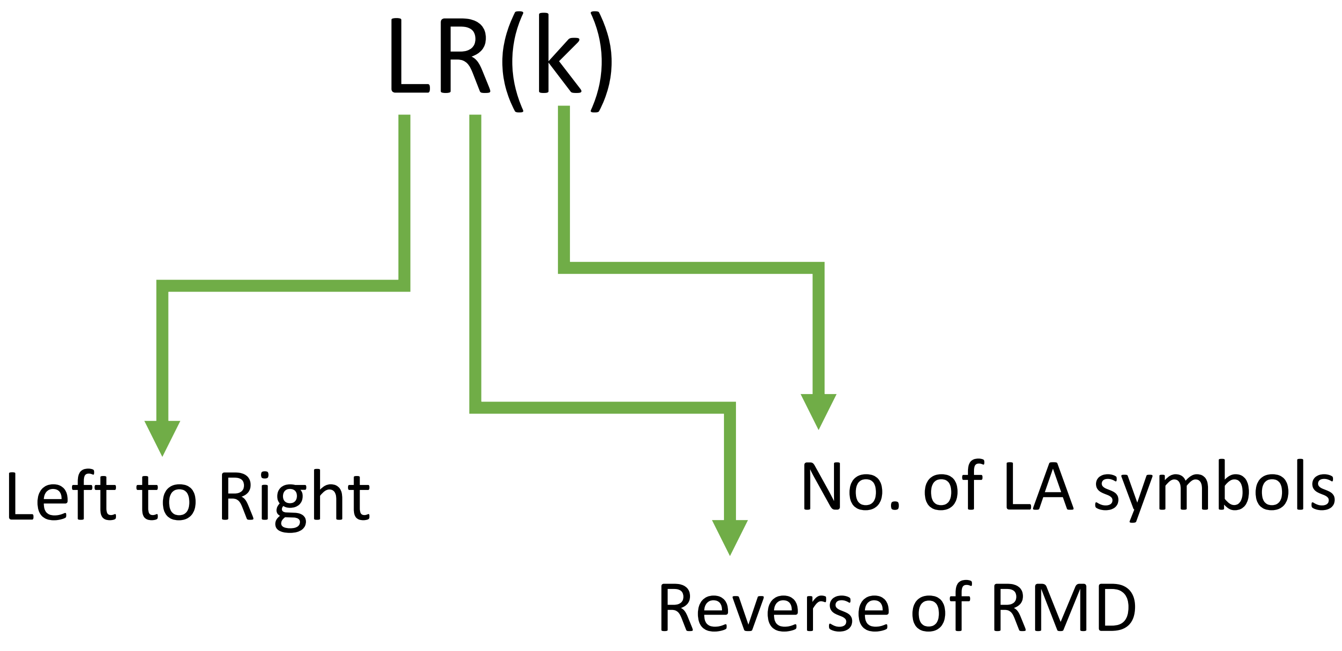
Stack Content	Input	Action
$	x*x$	shift
$x	*x$	Reduce F->x
$F	*x$	Reduce T->F
$T	*x$	Reduce E->T
$E	*x$	Shift
$E*	x$	Shift
$E*x	$	Reduce F->x
$E*F	$	Reduce T->F
$E*T	$	Reduce E->T
$E*E	$	**Error**

- S->aABe
- A->Abc|b
- B->d
- Example String: abbcde

- S->(L)|w
- L->L,S|S
- Input String: (w,(w,w))

LR(k) parser

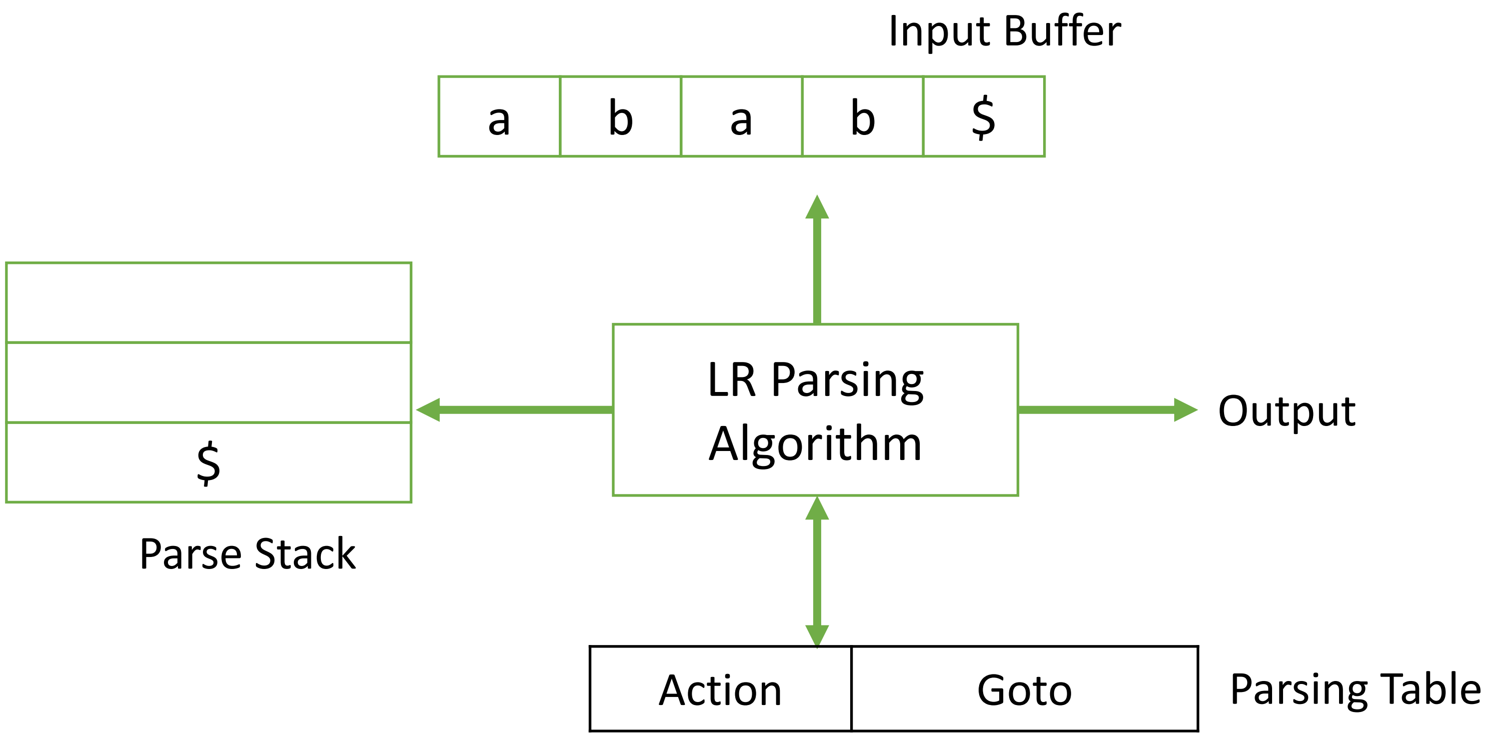
- Constructed for unambiguous grammar
- May or may not depend on LA symbol



Classification of LR Parser

- LR(0)
- SLR(1): Simple LR
- CLR(1): Canonical LR
- LALR(1): Look Ahead LR

Components of LR Parser



Behavior of LR Parser

- Parsing algorithm reads the next unread input character from the Input Buffer
- Parsing algorithm also reads the character on the top of the stack.
- A stack can have grammar symbol (Xi) or state symbol (Si)
- Combination of input character and top of stack char is used to index parsing table
- Parsing action can be: (1) Shift (2) Reduce (3) Error (4) Accept
- Goto function takes a state and a grammar symbol and produces a state.

General Procedure to construct LR Parse Table

1. Construct the augmented grammar
2. Create canonical Collection of LR item or items of compiler
3. Draw the DFA using sets of LR items
4. Prepare the LR parse table based on LR items

- Note:
  - Any grammar for which we construct the LR(k) parser is called LR(k) grammar
  - LR(k) grammar is accepted by DPDA
  - The language generated by LR(k) grammar is DCFL

Augmented Grammar

- If G is a grammar with start symbol S, then G' (Augmented Grammar) contains the production from G along with a new production $S' \rightarrow S$ where S' is new start symbol of G'.
- The grammar which is obtained by adding one more production is called as augmented grammar.
- **Example:**
- S->AB
- A->a
- B->b
- **Augmented Grammar:**
- S'->S
- S->AB
- A->a
- B->b

<u>Why Required?</u>

It indicates that parser should stop parsing and announce acceptable when it is about to reduce $S' \rightarrow S$

LR(0) items

- The Production which has a dot (.) any where on R.H.S. is called as LR(0) items
- Ex. A->abc
- Item indicates how much part of a production we have seen at a given point in parsing process.
- LR(0)items:

A -> .abc
A -> a.bc
A -> ab.c

Non Final (Shift)

A -> abc.

Final (Reduce)

Handle

$\boxed{A \to \epsilon}$

$\downarrow$

$\boxed{A \to \cdot}$

Canonical Collection

- If $I_0, I_1, I_2, \ldots, I_k$be the set containing LR(0) items so then the set $I = \{I_0, I_1, \ldots, I_k\}$ this called canonical collection.
- The Function is used to generate LR(0) items:
- Closure(I) where, $I = Item$
- Goto(I, x) where, x is Grammar Symbol

Closure(I)

Set of Items (I_i) → **Closure** → Set of Items (I_j)

I

$A \rightarrow \alpha \cdot B\beta$

$\mathrm{B} \rightarrow \cdot\, \gamma$

1. Add everything from input to output
2. If $A \rightarrow \alpha \cdot B\beta$ is in I and $\mathrm{B} \rightarrow \gamma$ is in the grammar G then add $\mathrm{B} \rightarrow \cdot\, \gamma$ to the Closure (I). Where B is non terminal.
3. Repeat the step(2) for every newly added item.

Given Grammar
$A \rightarrow aA$
$A \rightarrow b$

Augmented Grammar
$A' \rightarrow A$
$A \rightarrow aA$
$A \rightarrow b$

$I_0: Closure(A' \rightarrow A)$

I_0

$A' \rightarrow \cdot\, A$

$A \rightarrow \cdot\, aA$

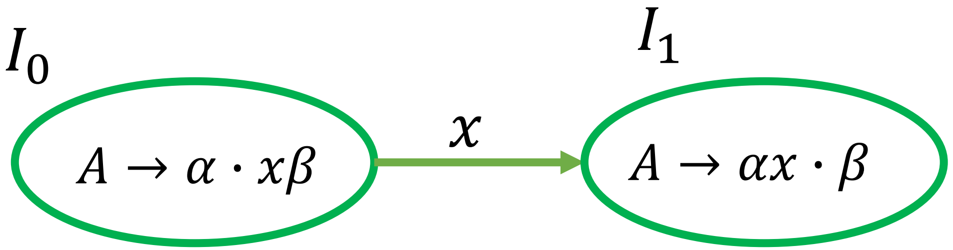
$\mathrm{A} \rightarrow \cdot\, b$

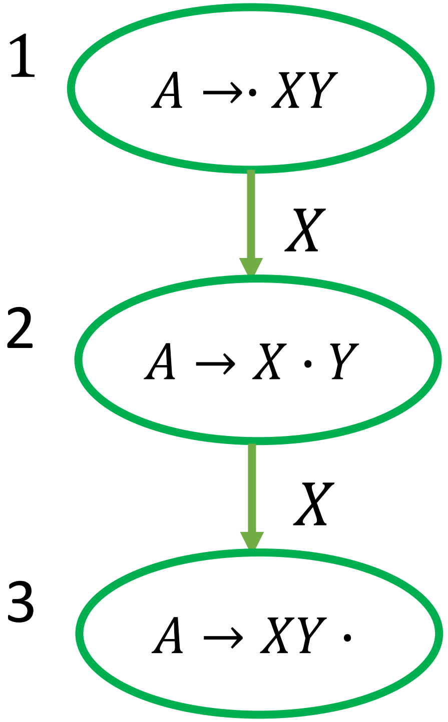
Closure(I)

- Compute CLOSURE whenever there is a dot to the immediate left of a Non-Terminal(NT) and the NT has not yet been expanded.
- Expansion of such NT into items with dot at extreme left is called CLOSURE.
- STEPS:
- Construct the Augmented Grammar
- Construct set I of LR(0) items of augmented Grammar.
- For each item that has dot to the immediate left of a non terminal expand the Set I by including items formed from this NT; including only those items with dot at extreme left.
- Repeat until new items are added.

$Goto(I, x)$

- $Goto(I, x)$ is the closure of $A \to \alpha x \cdot \beta$ such that $A \to \alpha \cdot x\beta$ is in I.
- Example,





Goto Function Example

Example Grammar:

$$A \to aA$$

$$A \to b$$

Goto Function Example

Example Grammar:

$$A \to aA$$
$$A \to b$$

Augmented Grammar:

$$A' \to A$$
$$A \to aA$$
$$A \to b$$

Goto Function Example

Example Grammar:

$$A \to aA$$
$$A \to b$$

Augmented Grammar:

$$A' \to A$$
$$A \to aA$$
$$A \to b$$

I_0: Closure($A' \to A$)

$$A' \to \cdot A$$
$$A \to \cdot aA$$
$$A \to \cdot b$$

Goto Function Example

Example Grammar:

$$A \rightarrow aA$$
$$A \rightarrow b$$

Augmented Grammar:

$$A' \rightarrow A$$
$$A \rightarrow aA$$
$$A \rightarrow b$$

I_0: Closure($A' \rightarrow A$)

$$A' \rightarrow \cdot A$$
$$A \rightarrow \cdot aA$$
$$A \rightarrow \cdot b$$

I_1: Goto(I_0,A)

$$A' \rightarrow A \cdot$$

Goto Function Example

Example Grammar:

$$A \rightarrow aA$$
$$A \rightarrow b$$

Augmented Grammar:

$$A' \rightarrow A$$
$$A \rightarrow aA$$
$$A \rightarrow b$$

I_0: Closure($A' \rightarrow A$)

$$A' \rightarrow \cdot A$$
$$A \rightarrow \cdot aA$$
$$A \rightarrow \cdot b$$

I_1: Goto(I_0,A)

$$A' \rightarrow A \cdot$$

I_2: Goto(I_0,a)

$$A \rightarrow a \cdot A$$
$$A \rightarrow \cdot aA$$
$$A \rightarrow \cdot b$$

Goto Function Example

Example Grammar:

$$A \to aA$$
$$A \to b$$

Augmented Grammar:

$$A' \to A$$
$$A \to aA$$
$$A \to b$$

I_0: Closure($A' \to A$)

$$A' \to \cdot A$$
$$A \to \cdot aA$$
$$A \to \cdot b$$

I_1: Goto(I_0,A)

$$A' \to A \cdot$$

I_2: Goto(I_0,a)

$$A \to a \cdot A$$
$$A \to \cdot aA$$
$$A \to \cdot b$$

I_3: Goto(I_0,b)

$$A \to b \cdot$$

Goto Function Example

Example Grammar:

$$A \to aA$$
$$A \to b$$

Augmented Grammar:

$$A' \to A$$
$$A \to aA$$
$$A \to b$$

I_0: Closure($A' \to A$)

$$A' \to \cdot A$$
$$A \to \cdot aA$$
$$A \to \cdot b$$

I_1: Goto(I_0,A)

$$A' \to A \cdot$$

I_2: Goto(I_0,a)

$$A \to a \cdot A$$
$$A \to \cdot aA$$
$$A \to \cdot b$$

I_3: Goto(I_0,b)

$$A \to b \cdot$$

I_4: Goto(I_2,A)

$$A \to aA \cdot$$

$I_?$: Goto(I_2,a)

?

$I_?$: Goto(I_2,b)

?

Goto Function Example

Example Grammar:

$$A \rightarrow aA$$
$$A \rightarrow b$$

Augmented Grammar:

$$A' \rightarrow A$$
$$A \rightarrow aA$$
$$A \rightarrow b$$

I_0: Closure($A' \rightarrow A$)

$$A' \rightarrow \cdot A$$
$$A \rightarrow \cdot aA$$
$$A \rightarrow \cdot b$$

I_1: Goto(I_0,A)

$$A' \rightarrow A \cdot$$

I_2: Goto(I_0,a)

$$A \rightarrow a \cdot A$$
$$A \rightarrow \cdot aA$$
$$A \rightarrow \cdot b$$

I_3: Goto(I_0,b)

$$A \rightarrow b \cdot$$

I_4: Goto(I_2,A)

$$A \rightarrow aA \cdot$$

Goto(I_2,a) = I_2

Goto(I_2,b)= I_3

Canonical Collection

Example Grammar:

$$A \rightarrow aA$$
$$A \rightarrow b$$

Augmented Grammar:

$$A' \rightarrow A$$
$$A \rightarrow aA$$
$$A \rightarrow b$$

I_0: Closure($A' \rightarrow A$)

$$A' \rightarrow \cdot A$$
$$A \rightarrow \cdot aA$$
$$A \rightarrow \cdot b$$

I_1: Goto(I_0,A)

$$A' \rightarrow A \cdot$$

I_2: Goto(I_0,a)

$$A \rightarrow a \cdot A$$
$$A \rightarrow \cdot aA$$
$$A \rightarrow \cdot b$$

I_3: Goto(I_0,b)

$$A \rightarrow b \cdot$$

I_4: Goto(I_2,A)

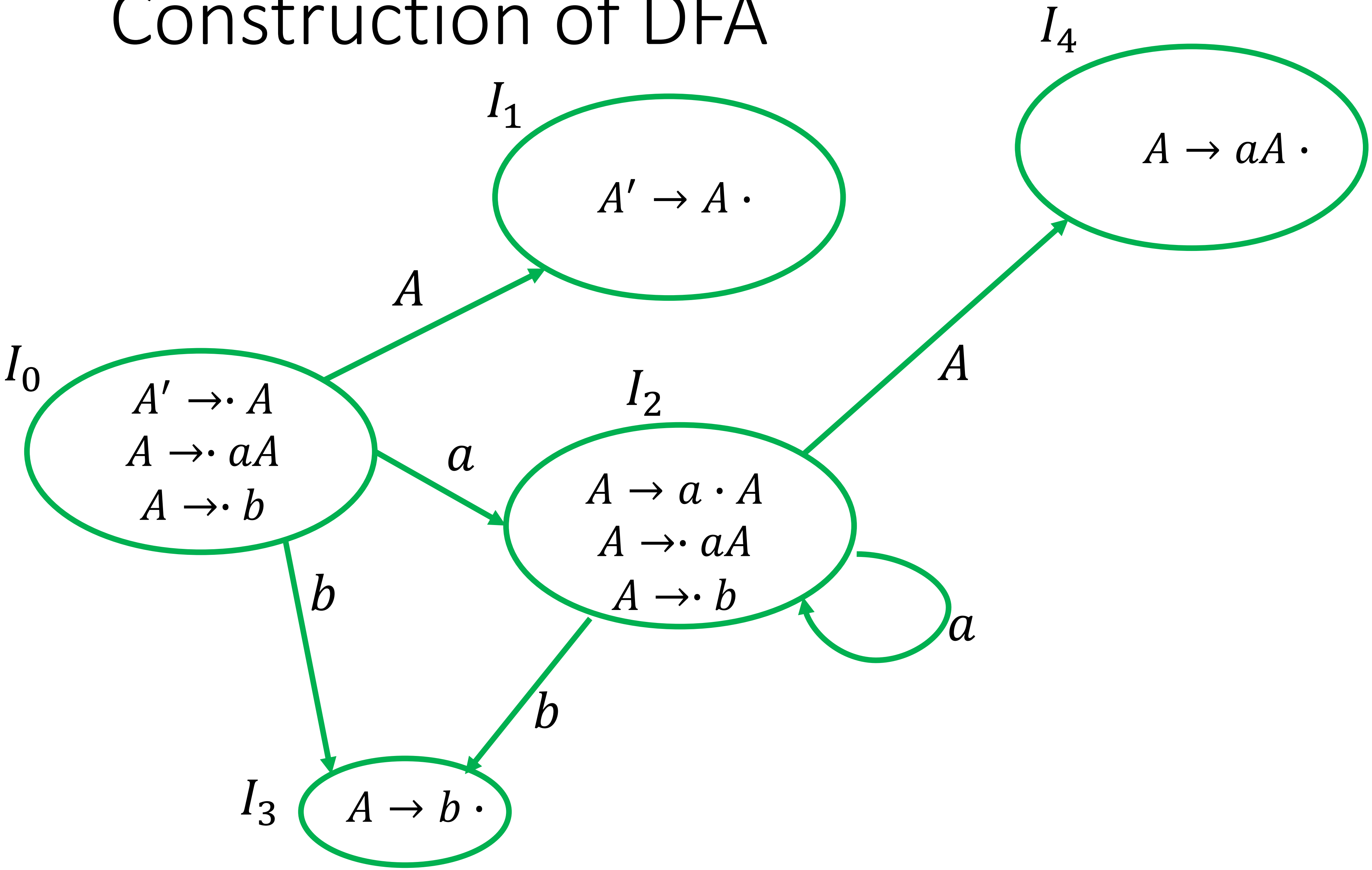
$$A \rightarrow aA \cdot$$

Goto(I_2,a) = I_2

Goto(I_2,b)= I_3

Canonical Collection $C = \{I_0, I_1, I_2, I_3, I_4\}$

Construction of DFA



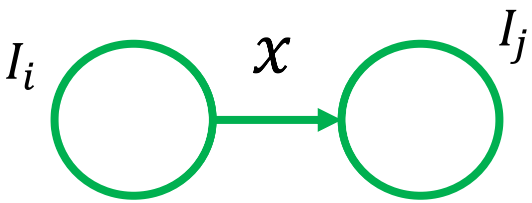
Construction of LR(0) Parse Table

States	Action Terminals			GoTo Non Terminals	
	Terminal-1	Terminal-2	Terminal-3	Non Terminal-1	Non Terminal-2
I_0					
I_1					
I_2					

Action

	x
I_i	S_j

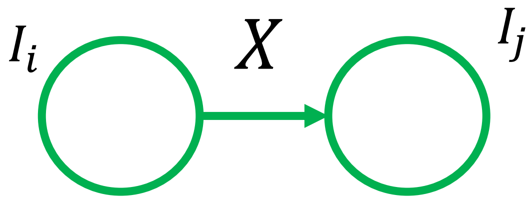
$x \rightarrow$ **Terminals**



GoTo

	X
I_i	j

$X \rightarrow$ **Non Terminals**



Construction of LR(0) Parse Table

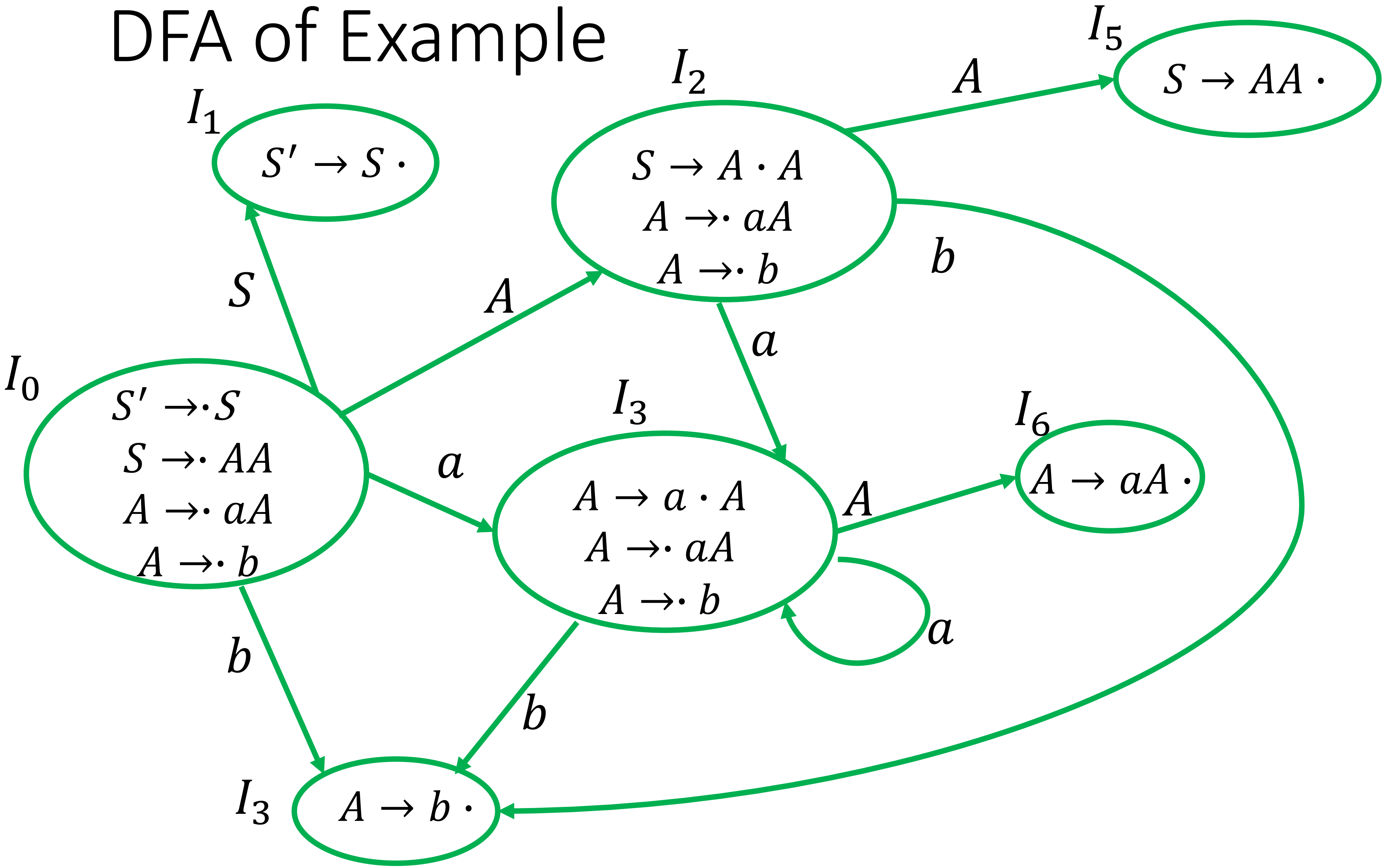
States	Action			GoTo
	a	b	$	A
I_0	s_2	s_3		1
I_1			Acc	
I_2	s_2	s_3		4
I_3	r_2	r_2	r_2	
I_4	r_1	r_1	r_1	

$$A \rightarrow aA\ (\boldsymbol{r_1})$$
$$A \rightarrow b\ (\boldsymbol{r_2})$$

Example

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

DFA of Example



Parse Table of Example

States	Action			GoTo	
	a	b	\$	S	A
I_0	s_3	s_4		1	2
I_1			Accept		
I_2	s_3	s_4			5
I_3	s_3	s_4			6
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

$$S \to AA$$
$$A \to aA|b$$

$$S \to AA\ (\boldsymbol{r_1})$$
$$A \to aA\ (\boldsymbol{r_2})$$
$$A \to b\ (\boldsymbol{r_3})$$

Parsing a string by LR(0) Parser

$$S \rightarrow AA\ (\boldsymbol{r_1})$$
$$A \rightarrow aA\ (\boldsymbol{r_2})$$
$$A \rightarrow b\ (\boldsymbol{r_3})$$

Stack Content	**Input**	**Action**
$\$\boldsymbol{I_0}$	bb$	

States	**Action**			**GoTo**	
	a	**b**	**$**	**S**	**A**
$\boldsymbol{I_0}$	$\boldsymbol{s_3}$	$\boldsymbol{s_4}$		1	2
$\boldsymbol{I_1}$			Acc		
$\boldsymbol{I_2}$	$\boldsymbol{s_3}$	$\boldsymbol{s_4}$			5
$\boldsymbol{I_3}$	$\boldsymbol{s_3}$	$\boldsymbol{s_4}$			6
$\boldsymbol{I_4}$	$\boldsymbol{r_3}$	$\boldsymbol{r_3}$	$\boldsymbol{r_3}$		
$\boldsymbol{I_5}$	$\boldsymbol{r_1}$	$\boldsymbol{r_1}$	$\boldsymbol{r_1}$		
$\boldsymbol{I_6}$	$\boldsymbol{r_2}$	$\boldsymbol{r_2}$	$\boldsymbol{r_2}$		

Parsing a string by LR(0) Parser

$S \rightarrow AA\ (\boldsymbol{r_1})$
$A \rightarrow aA\ (\boldsymbol{r_2})$
$A \rightarrow b\ (\boldsymbol{r_3})$

Stack Content	**Input**	**Action**
$\$\boldsymbol{I_0}$	bb$	$\boldsymbol{s_4}$
$\$\boldsymbol{I_0}\mathrm{b}\boldsymbol{I_4}$	b$	

States	**Action**			**GoTo**	
	a	**b**	**$**	**S**	**A**
$\boldsymbol{I_0}$	$\boldsymbol{s_3}$	$\boldsymbol{s_4}$		1	2
$\boldsymbol{I_1}$			Acc		
$\boldsymbol{I_2}$	$\boldsymbol{s_3}$	$\boldsymbol{s_4}$			5
$\boldsymbol{I_3}$	$\boldsymbol{s_3}$	$\boldsymbol{s_4}$			6
$\boldsymbol{I_4}$	$\boldsymbol{r_3}$	$\boldsymbol{r_3}$	$\boldsymbol{r_3}$		
$\boldsymbol{I_5}$	$\boldsymbol{r_1}$	$\boldsymbol{r_1}$	$\boldsymbol{r_1}$		
$\boldsymbol{I_6}$	$\boldsymbol{r_2}$	$\boldsymbol{r_2}$	$\boldsymbol{r_2}$		

Parsing a string by LR(0) Parser

$$S \to AA\ (\boldsymbol{r_1})$$
$$A \to aA\ (\boldsymbol{r_2})$$
$$A \to b\ (\boldsymbol{r_3})$$

Stack Content	Input	Action
$\$\boldsymbol{I_0}$	bb$	$\boldsymbol{s_4}$
$\$\boldsymbol{I_0}\mathrm{b}\boldsymbol{I_4}$	b$	$\boldsymbol{r_3}$
$\$\boldsymbol{I_0}\mathrm{A}\boldsymbol{I_2}$	b$	

States	Action			GoTo	
	a	b	$	S	A
$\boldsymbol{I_0}$	$\boldsymbol{s_3}$	$\boldsymbol{s_4}$		1	2
$\boldsymbol{I_1}$			Acc		
$\boldsymbol{I_2}$	$\boldsymbol{s_3}$	$\boldsymbol{s_4}$			5
$\boldsymbol{I_3}$	$\boldsymbol{s_3}$	$\boldsymbol{s_4}$			6
$\boldsymbol{I_4}$	$\boldsymbol{r_3}$	$\boldsymbol{r_3}$	$\boldsymbol{r_3}$		
$\boldsymbol{I_5}$	$\boldsymbol{r_1}$	$\boldsymbol{r_1}$	$\boldsymbol{r_1}$		
$\boldsymbol{I_6}$	$\boldsymbol{r_2}$	$\boldsymbol{r_2}$	$\boldsymbol{r_2}$		

Parsing a string by LR(0) Parser

$S \rightarrow AA\ (\boldsymbol{r_1})$
$A \rightarrow aA\ (\boldsymbol{r_2})$
$A \rightarrow b\ (\boldsymbol{r_3})$

Stack Content	**Input**	**Action**
$\$\boldsymbol{I_0}$	bb$	$\boldsymbol{s_4}$
$\$\boldsymbol{I_0}\mathrm{b}\boldsymbol{I_4}$	b$	$\boldsymbol{r_3}$
$\$\boldsymbol{I_0}\mathrm{A}\boldsymbol{I_2}$	b$	$\boldsymbol{s_4}$
$\$\boldsymbol{I_0}\mathrm{A}\boldsymbol{I_2}\mathrm{b}\boldsymbol{I_4}$	$	

States	**Action**			**GoTo**	
	a	**b**	**$**	**S**	**A**
$\boldsymbol{I_0}$	$\boldsymbol{s_3}$	$\boldsymbol{s_4}$		1	2
$\boldsymbol{I_1}$			Acc		
$\boldsymbol{I_2}$	$\boldsymbol{s_3}$	$\boldsymbol{s_4}$			5
$\boldsymbol{I_3}$	$\boldsymbol{s_3}$	$\boldsymbol{s_4}$			6
$\boldsymbol{I_4}$	$\boldsymbol{r_3}$	$\boldsymbol{r_3}$	$\boldsymbol{r_3}$		
$\boldsymbol{I_5}$	$\boldsymbol{r_1}$	$\boldsymbol{r_1}$	$\boldsymbol{r_1}$		
$\boldsymbol{I_6}$	$\boldsymbol{r_2}$	$\boldsymbol{r_2}$	$\boldsymbol{r_2}$		

Parsing a string by LR(0) Parser

$S \rightarrow AA\ (\boldsymbol{r_1})$
$A \rightarrow aA\ (\boldsymbol{r_2})$
$A \rightarrow b\ (\boldsymbol{r_3})$

Stack Content	Input	Action
$\$\boldsymbol{I_0}$	bb$	$\boldsymbol{s_4}$
$\$\boldsymbol{I_0}\mathrm{b}\boldsymbol{I_4}$	b$	$\boldsymbol{r_3}$
$\$\boldsymbol{I_0}\mathrm{A}\boldsymbol{I_2}$	b$	$\boldsymbol{s_4}$
$\$\boldsymbol{I_0}\mathrm{A}\boldsymbol{I_2}\mathrm{b}\boldsymbol{I_4}$	$	$\boldsymbol{r_3}$
$\$\boldsymbol{I_0}\mathrm{A}\boldsymbol{I_2}\boldsymbol{AI_5}$	$	

States	Action			GoTo	
	a	b	$	S	A
$\boldsymbol{I_0}$	$\boldsymbol{s_3}$	$\boldsymbol{s_4}$		1	2
$\boldsymbol{I_1}$			Acc		
$\boldsymbol{I_2}$	$\boldsymbol{s_3}$	$\boldsymbol{s_4}$			5
$\boldsymbol{I_3}$	$\boldsymbol{s_3}$	$\boldsymbol{s_4}$			6
$\boldsymbol{I_4}$	$\boldsymbol{r_3}$	$\boldsymbol{r_3}$	$\boldsymbol{r_3}$		
$\boldsymbol{I_5}$	$\boldsymbol{r_1}$	$\boldsymbol{r_1}$	$\boldsymbol{r_1}$		
$\boldsymbol{I_6}$	$\boldsymbol{r_2}$	$\boldsymbol{r_2}$	$\boldsymbol{r_2}$		

Parsing a string by LR(0) Parser

$S \rightarrow AA\ (\boldsymbol{r_1})$
$A \rightarrow aA\ (\boldsymbol{r_2})$
$A \rightarrow b\ (\boldsymbol{r_3})$

Stack Content	Input	Action
$\$\boldsymbol{I_0}$	bb$	$\boldsymbol{s_4}$
$\$\boldsymbol{I_0}\mathrm{b}\boldsymbol{I_4}$	b$	$\boldsymbol{r_3}$
$\$\boldsymbol{I_0}\mathrm{A}\boldsymbol{I_2}$	b$	$\boldsymbol{s_4}$
$\$\boldsymbol{I_0}\mathrm{A}\boldsymbol{I_2}\mathrm{b}\boldsymbol{I_4}$	$	$\boldsymbol{r_3}$
$\$\boldsymbol{I_0}\mathrm{A}\boldsymbol{I_2}\boldsymbol{AI_5}$	$	$\boldsymbol{r_1}$
$\$\boldsymbol{I_0}\mathrm{S}\boldsymbol{I_1}$	$	

States	Action			GoTo	
	a	b	$	S	A
$\boldsymbol{I_0}$	$\boldsymbol{s_3}$	$\boldsymbol{s_4}$		1	2
$\boldsymbol{I_1}$			Acc		
$\boldsymbol{I_2}$	$\boldsymbol{s_3}$	$\boldsymbol{s_4}$			5
$\boldsymbol{I_3}$	$\boldsymbol{s_3}$	$\boldsymbol{s_4}$			6
$\boldsymbol{I_4}$	$\boldsymbol{r_3}$	$\boldsymbol{r_3}$	$\boldsymbol{r_3}$		
$\boldsymbol{I_5}$	$\boldsymbol{r_1}$	$\boldsymbol{r_1}$	$\boldsymbol{r_1}$		
$\boldsymbol{I_6}$	$\boldsymbol{r_2}$	$\boldsymbol{r_2}$	$\boldsymbol{r_2}$		

Parsing a string by LR(0) Parser

$S \rightarrow AA\ (\boldsymbol{r_1})$
$A \rightarrow aA\ (\boldsymbol{r_2})$
$A \rightarrow b\ (\boldsymbol{r_3})$

Stack Content	**Input**	**Action**
$\$\boldsymbol{I_0}$	bb$	$\boldsymbol{s_4}$
$\$\boldsymbol{I_0}\mathrm{b}\boldsymbol{I_4}$	b$	$\boldsymbol{r_3}$
$\$\boldsymbol{I_0}\mathrm{A}\boldsymbol{I_2}$	b$	$\boldsymbol{s_4}$
$\$\boldsymbol{I_0}\mathrm{A}\boldsymbol{I_2}\mathrm{b}\boldsymbol{I_4}$	$	$\boldsymbol{r_3}$
$\$\boldsymbol{I_0}\mathrm{A}\boldsymbol{I_2}\boldsymbol{AI_5}$	$	$\boldsymbol{r_1}$
$\$\boldsymbol{I_0}\mathrm{S}\boldsymbol{I_1}$	$	Accept

States	**Action**			**GoTo**	
	a	**b**	**$**	**S**	**A**
$\boldsymbol{I_0}$	$\boldsymbol{s_3}$	$\boldsymbol{s_4}$		1	2
$\boldsymbol{I_1}$			Acc		
$\boldsymbol{I_2}$	$\boldsymbol{s_3}$	$\boldsymbol{s_4}$			5
$\boldsymbol{I_3}$	$\boldsymbol{s_3}$	$\boldsymbol{s_4}$			6
$\boldsymbol{I_4}$	$\boldsymbol{r_3}$	$\boldsymbol{r_3}$	$\boldsymbol{r_3}$		
$\boldsymbol{I_5}$	$\boldsymbol{r_1}$	$\boldsymbol{r_1}$	$\boldsymbol{r_1}$		
$\boldsymbol{I_6}$	$\boldsymbol{r_2}$	$\boldsymbol{r_2}$	$\boldsymbol{r_2}$		

Example

- $S \rightarrow Aa|Bb$
- $A \rightarrow d$
- $B \rightarrow d$

LR(0) Grammar

- The Grammar for which LR(0) Parser can be constructed is called as LR(0) Grammar
- The grammar whose LR(0) parse table is free from multiple entries is called as LR(0) grammar.

SLR(1)

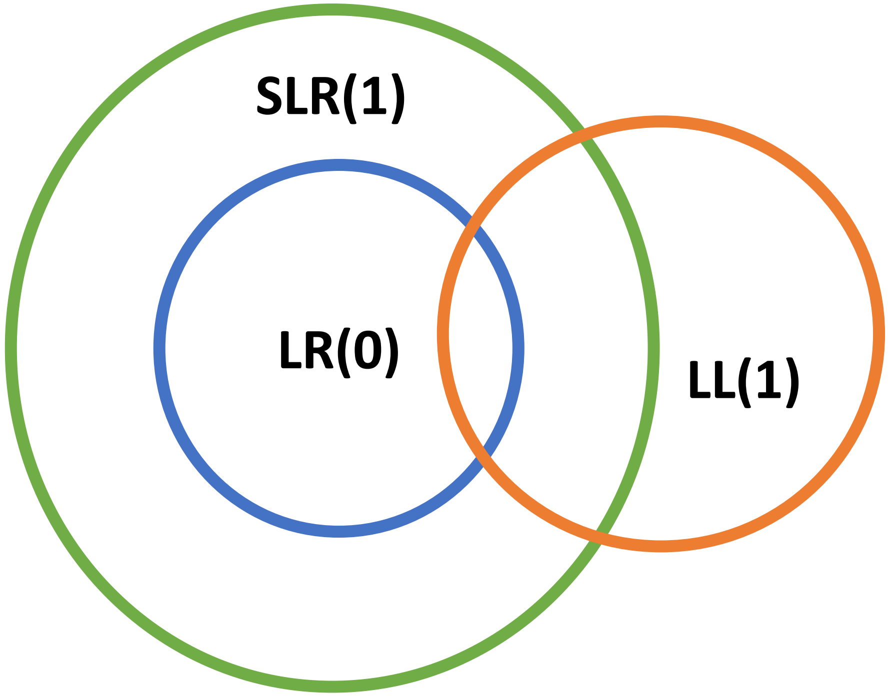
- In SLR(1) Parsing Table, reduce moves are not written in the complete row. Rather, reduce moves only appear in those columns that have terminals which appear in the follow of the left side Non-Terminal of the final item for which reduce move is being written.
- SLR(1) has a single lookahead symbol, unlike LR(0) parser which has NO lookahead symbol.
- Due to less no. of reduce moves, there are more empty cells. Hence, HIGHER ERROR DETECTION POWER.

SLR(1)

- The Procedure for constructing the parse table for SLR(1) is similar to LR(0) but there is restrictions on reducing the entry.
- Wherever there is a final item then place the reduce entries under the follow symbol of LHS Non Terminal
- If SLR(1) parse table is free from multiple entries then the grammar is SLR(1) grammar

Relations between LR(0), SLR(1), LL(1)

- Every LR(0) grammar is SLR(1) but every SLR(1) grammar need not be LR(0).
- Number of entries in SLR(1) parse table $\leq$ Number of entries in LR(0) parse table



SR Conflict

- In a parsing table, if a cell has both shift move as well as reduce move, then shift-reduce conflict arises.
- SR Conflict is caused when grammar allows a production rule to be reduced in a state and in the same state another production rule is shifted for same token.

RR Conflict

- In a parsing table, if a cell has 2 different reduce moves then reduce-reduce conflict occurs.

SLR(1) Example

- $E \to T + E|T$
- $T \to T * F|F$
- $F \to id|(E)$

Examples

$$S \to AaB$$
$$A \to ab|a$$
$$B \to b$$

$$S \to Aa|bAc|dc|bda$$
$$A \to d$$

- $E \to T + E|T$
- $T \to id$

$$S \to Aa|Bb$$
$$A \to d$$
$$B \to d$$

$$S \to Aa|Ba$$
$$A \to d$$
$$B \to d$$

$$S \to AaAb|BaBa$$
$$A \to \epsilon$$
$$B \to \epsilon$$

$$S \to aAB|Ba|Ab$$
$$A \to c$$
$$B \to c$$

$$S \to Aab|Bc$$
$$A \to aA|a$$
$$B \to Ba|b$$

$$S \to AB|BA$$
$$A \to Aab|b$$
$$B \to BaA|a$$

$$S \to Aab|bab|bac|acb$$
$$A \to aBA|b$$
$$B \to b$$

$$A \to (A)|bA|a$$

Example

- $E \rightarrow T + E|T$
- $T \rightarrow id$

CLR(1)

- S->CC
- C->cC|d

$$S \rightarrow aAd|bBd|aBe|bAe$$
$$A \rightarrow c$$
$$B \rightarrow c$$

CLR(1)

$$S \rightarrow AA$$
$$A \rightarrow aA|b$$

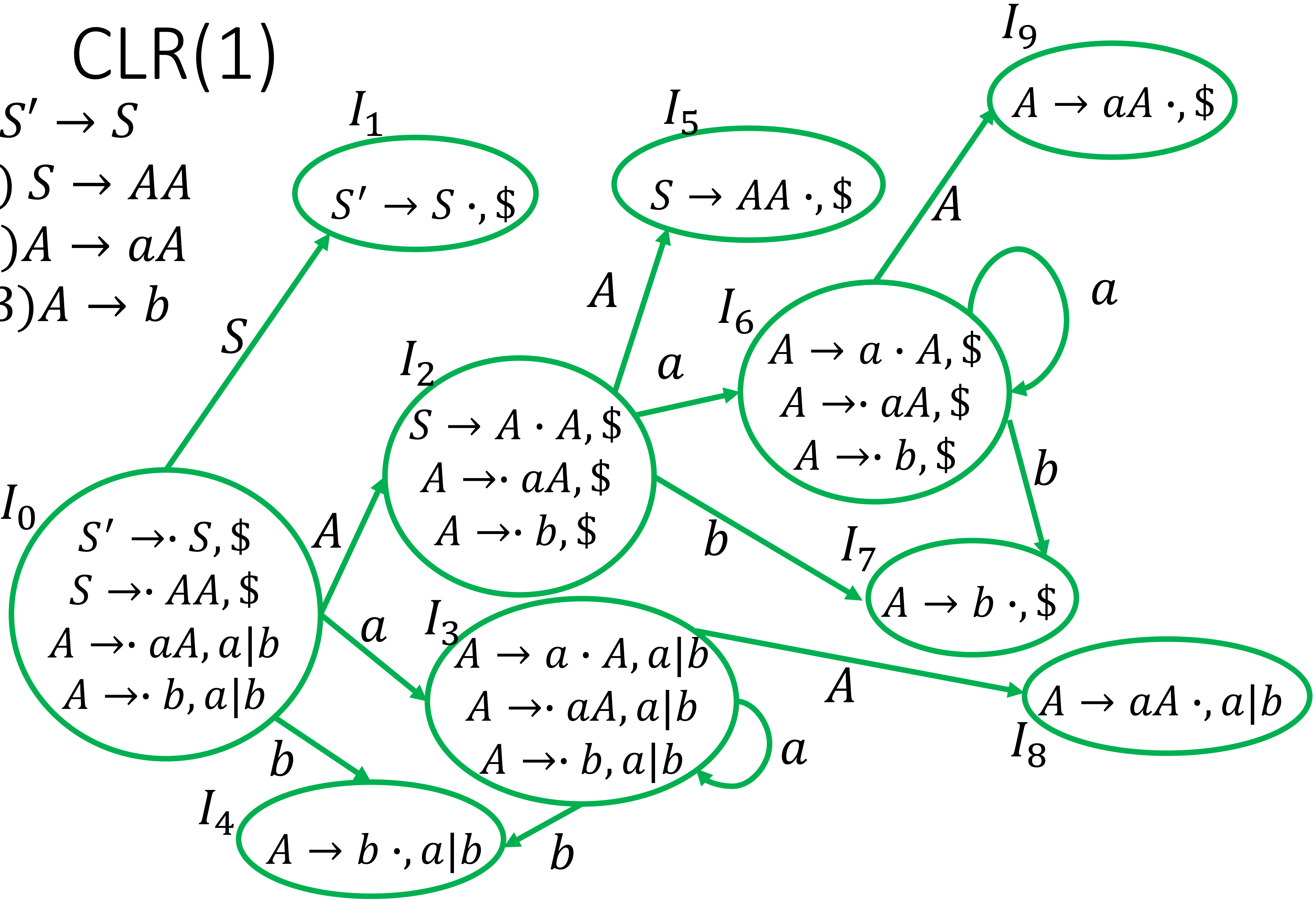
CLR(1)

$S' \rightarrow S$

(1) $S \rightarrow AA$

(2) $A \rightarrow aA$

(3) $A \rightarrow b$



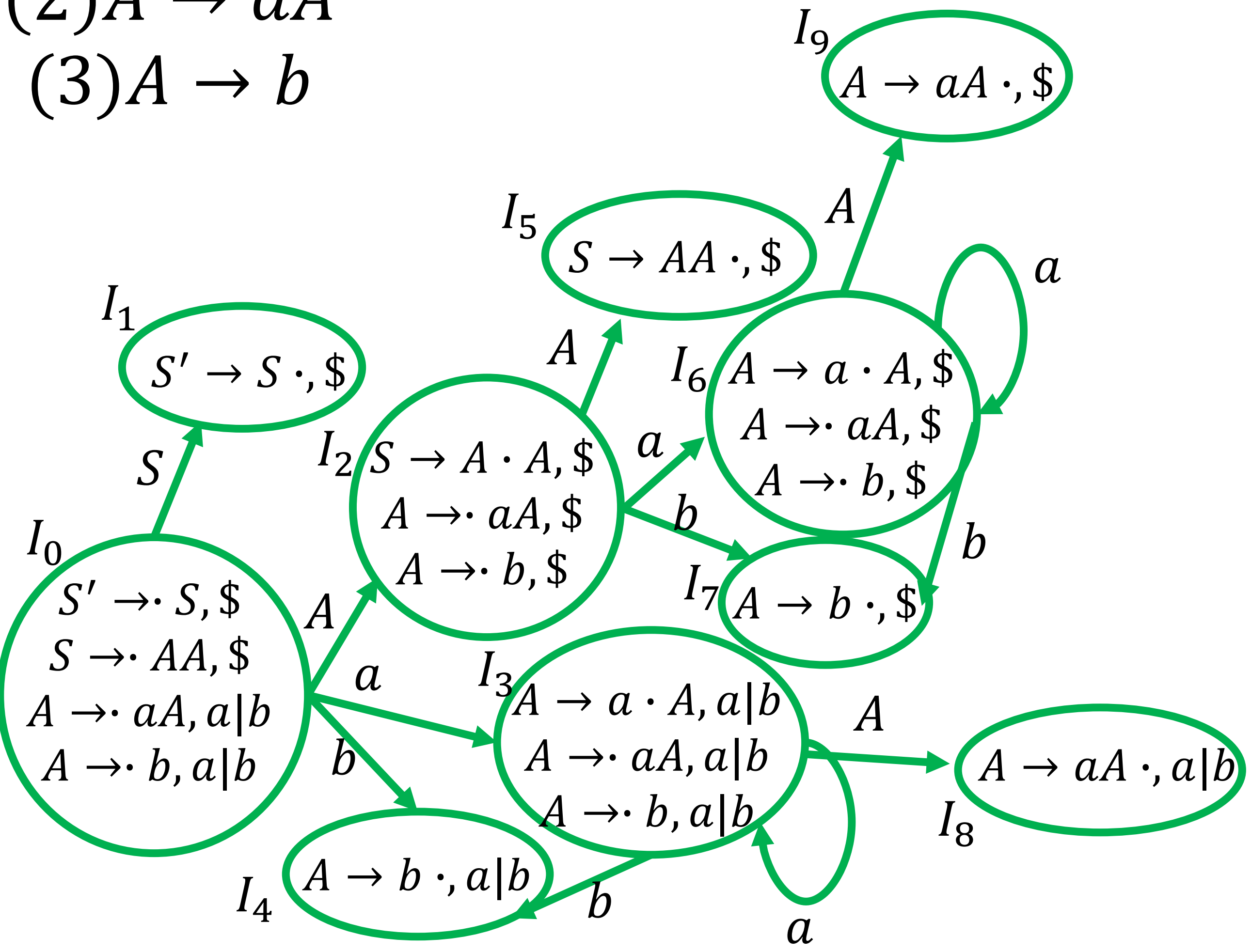
CLR(1)

$S' \to S$

(1) $S \to AA$

(2) $A \to aA$

(3) $A \to b$



States	Action			GoTo	
	a	b	$	S	A
I_0	s_3	s_4		1	2
I_1					
I_2	s_6	s_7			5
I_3	s_3	s_4			8
I_4					
I_5					
I_6	s_6	s_7			9
I_7					
I_8					
I_9					

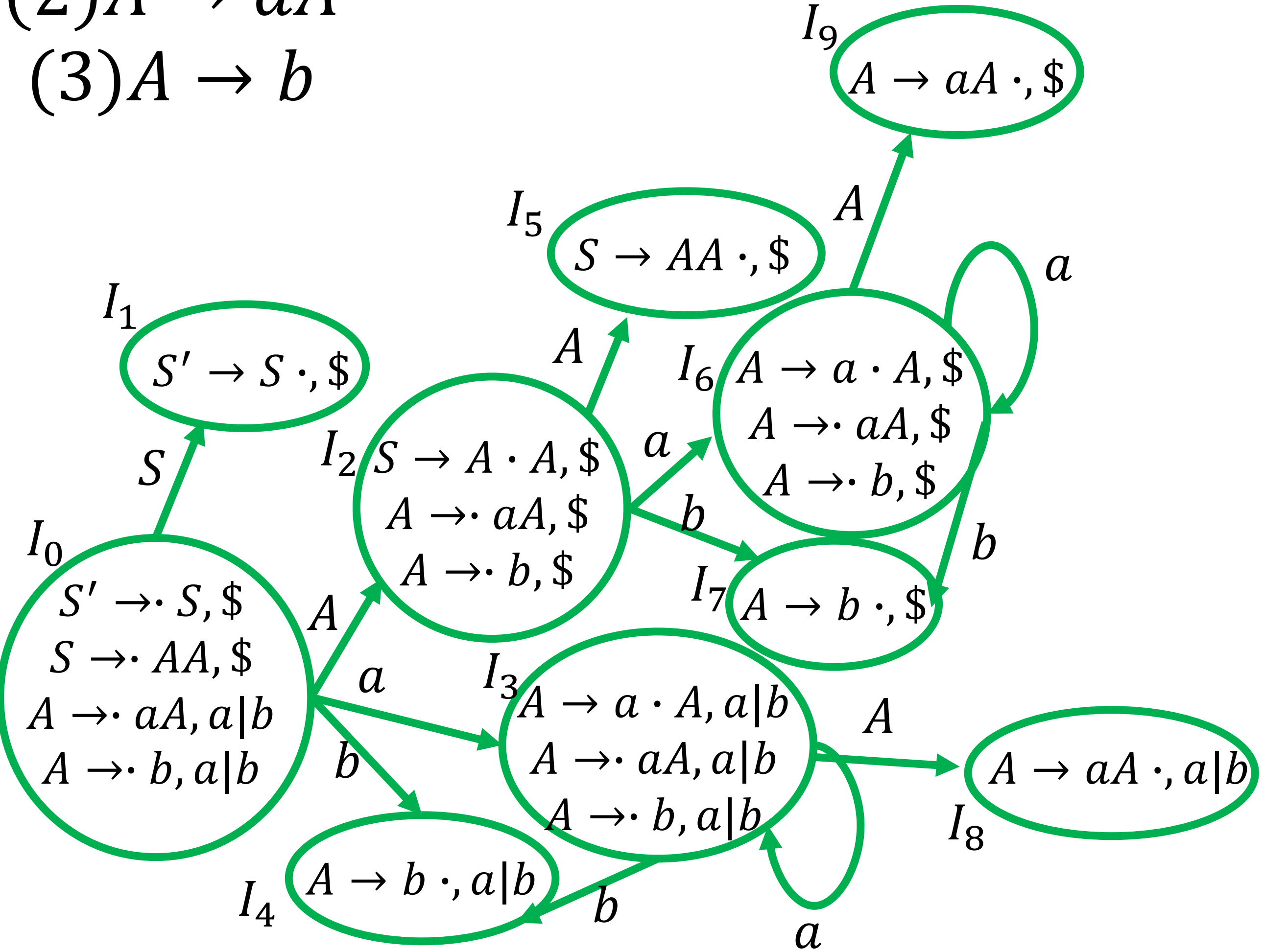
CLR(1)

$S' \to S$

(1) $S \to AA$

(2) $A \to aA$

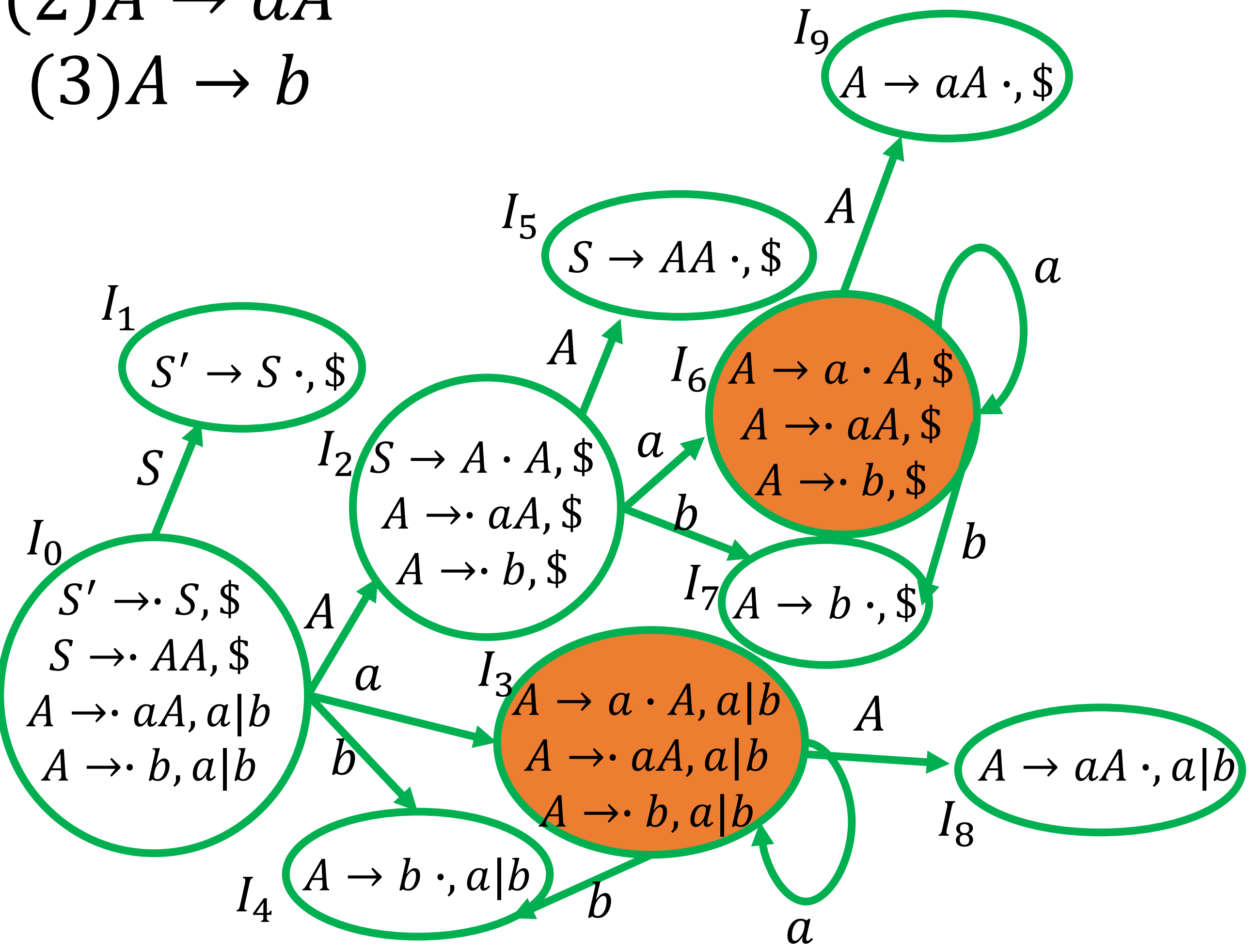
(3) $A \to b$



States	Action			GoTo	
	a	b	\$	S	A
I_0	s_3	s_4		1	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)

$$S' \to S$$
$$(1)\ S \to AA$$
$$(2)\ A \to aA$$
$$(3)\ A \to b$$



States	Action			GoTo	
	a	b	\$	S	A
I_0	s_3	s_4		1	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		

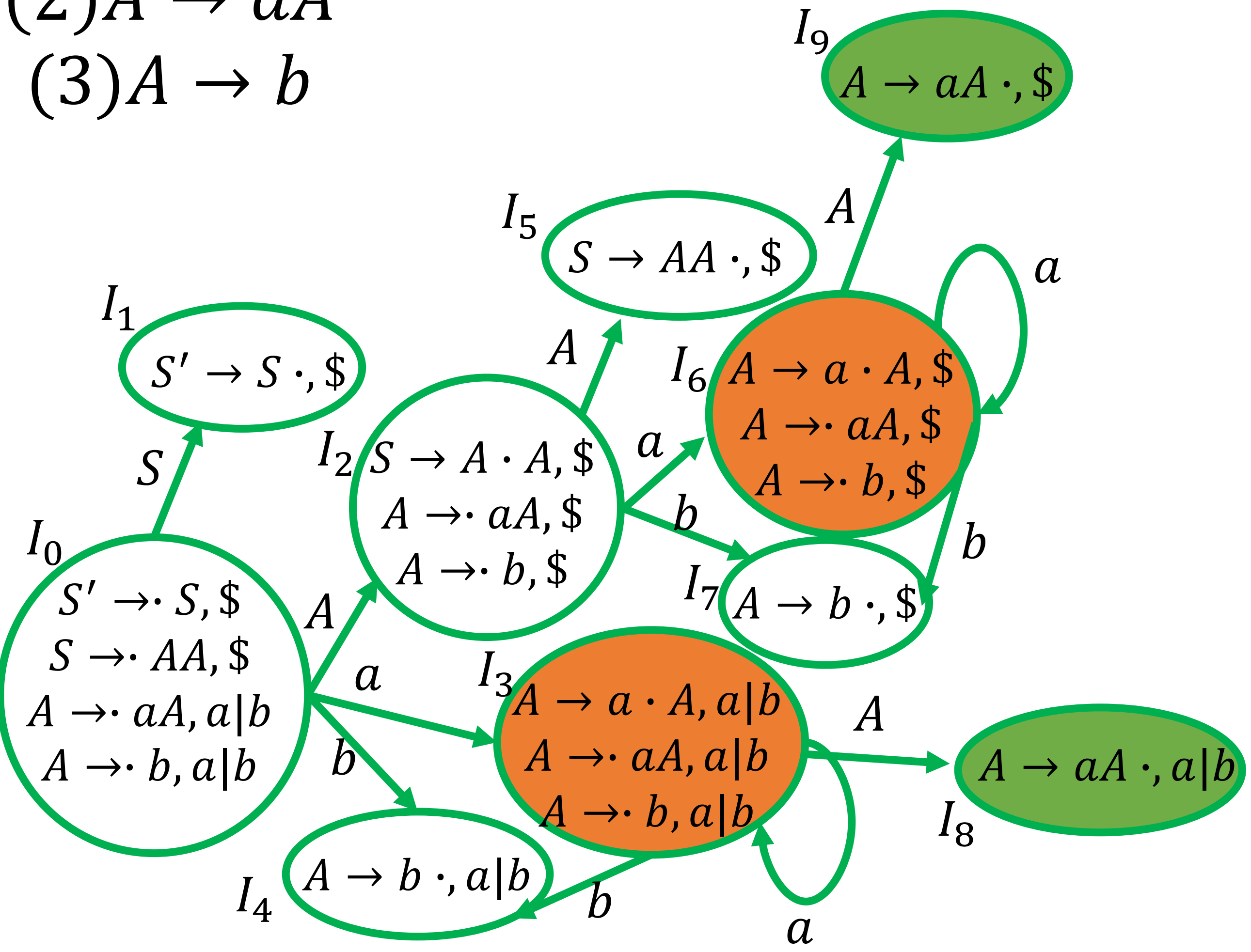
LALR(1)

$S' \to S$

$(1)\ S \to AA$

$(2) A \to aA$

$(3) A \to b$



States	Action			GoTo	
	a	b	\$	S	A
I_0	s_3	s_4		1	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		

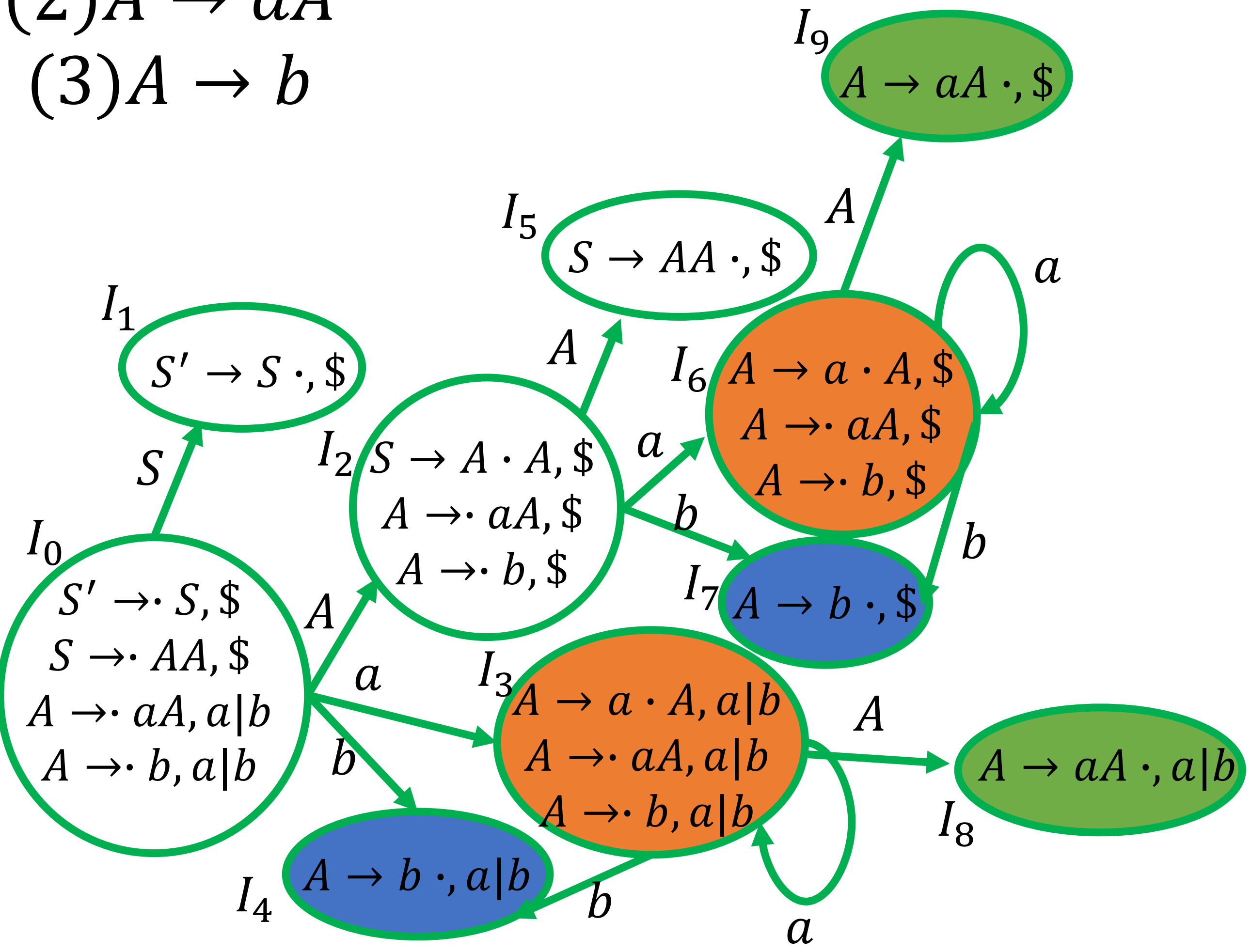
LALR(1)

$S' \rightarrow S$

$(1)\ S \rightarrow AA$

$(2) A \rightarrow aA$

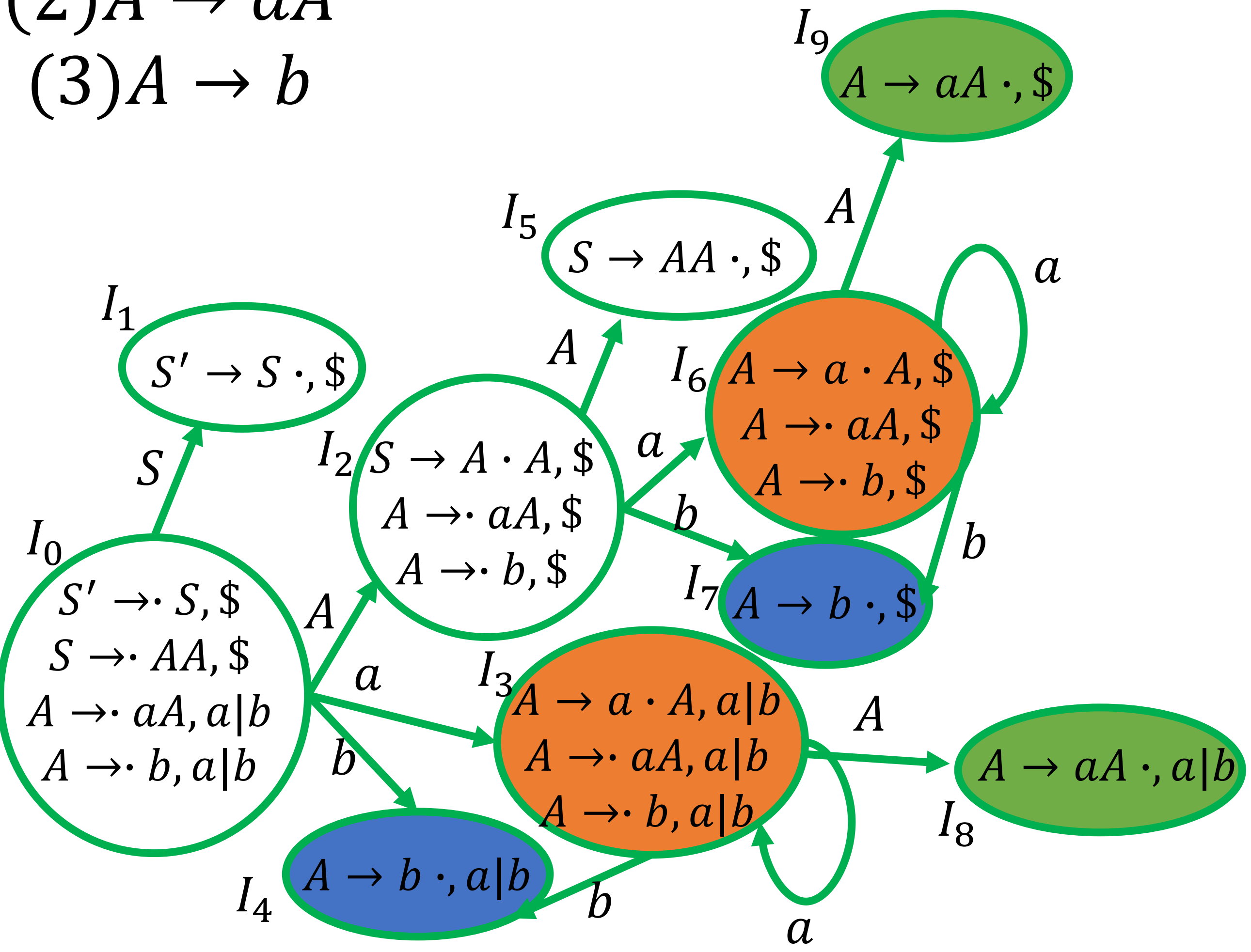
$(3) A \rightarrow b$



States	Action			GoTo	
	a	b	$	S	A
I_0	s_3	s_4		1	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)

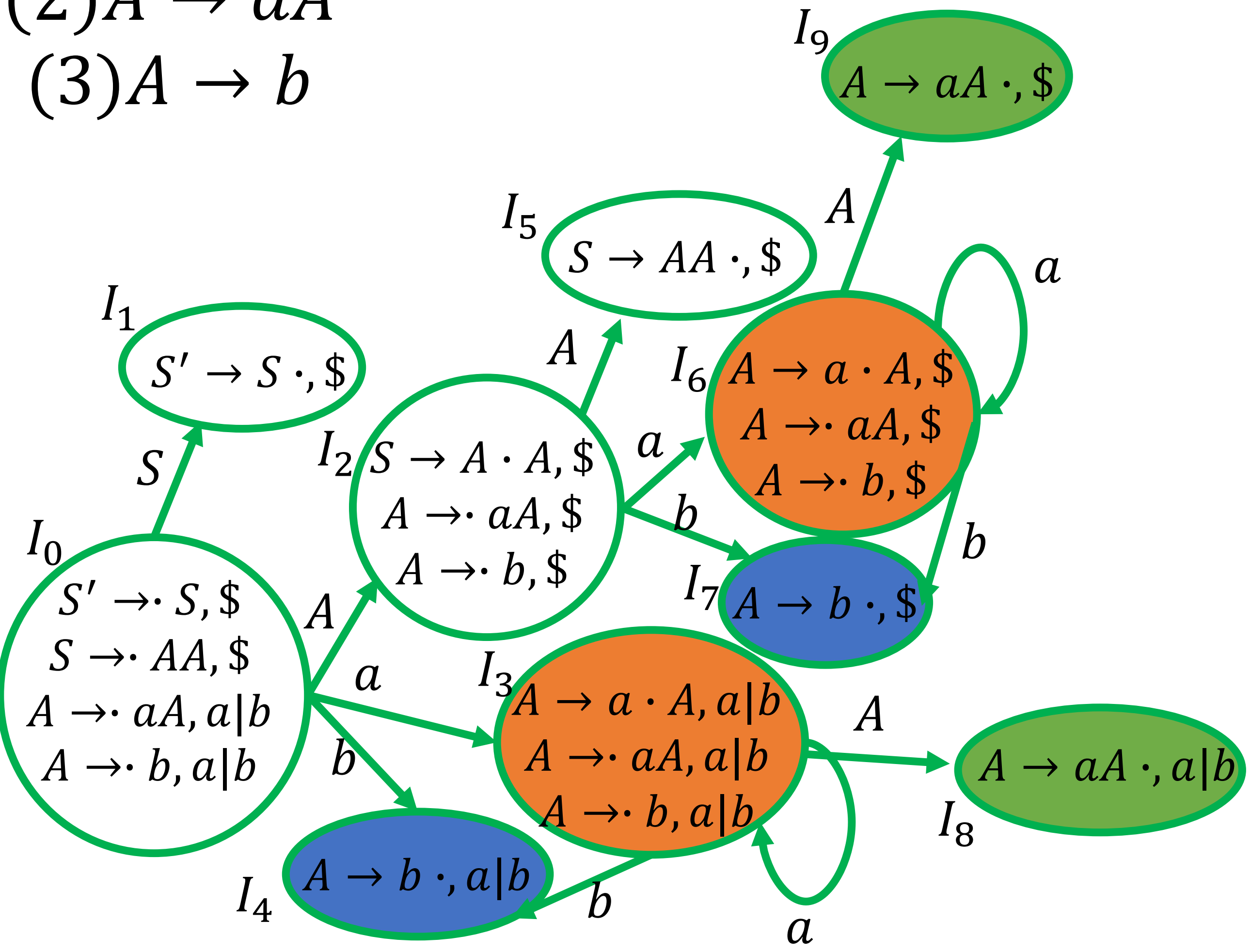
$$S' \to S$$
$$(1)\ S \to AA$$
$$(2) A \to aA$$
$$(3) A \to b$$



States	Action			GoTo	
	a	b	\$	S	A
I_0	s_3	s_4		1	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)

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



States	Action			GoTo	
	a	b	$	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		

LALR(1)

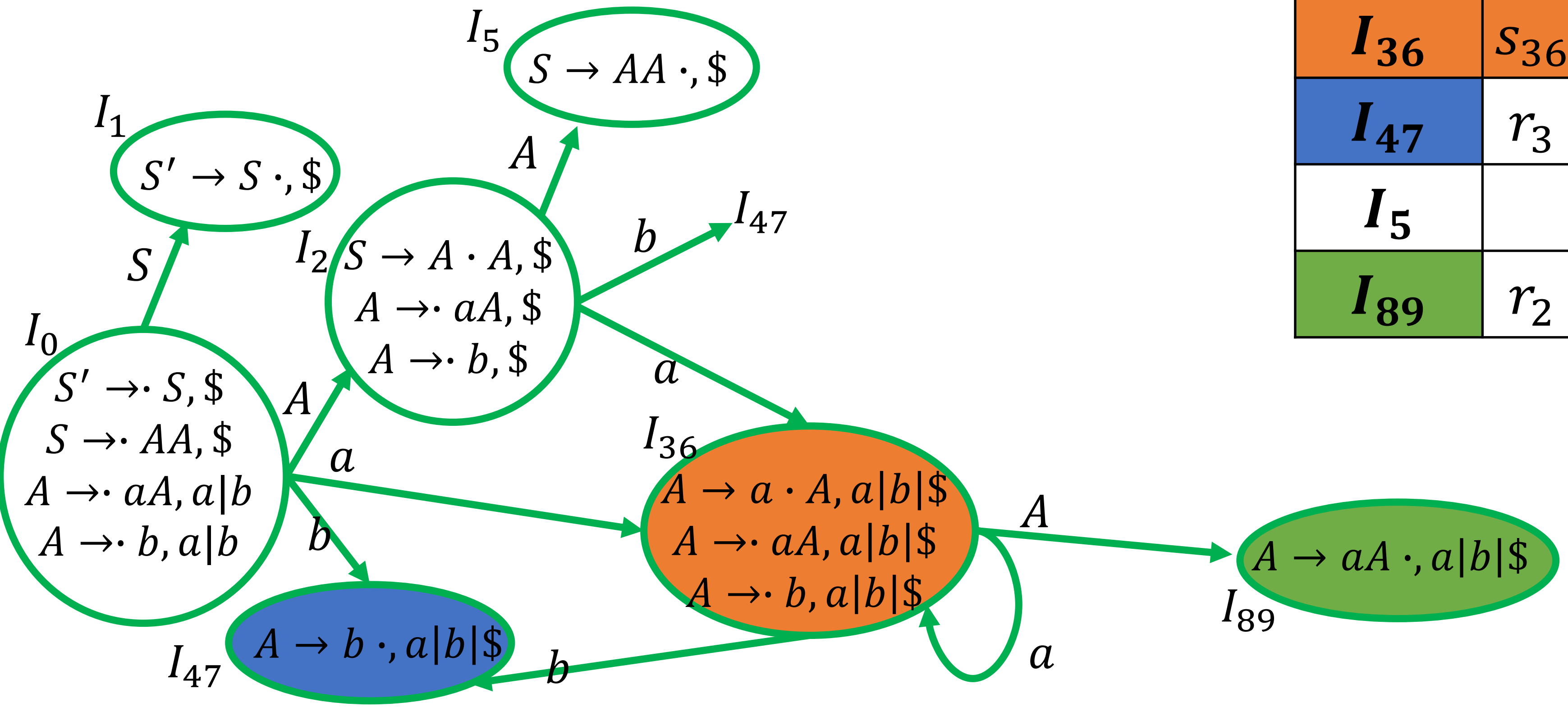
$S' \to S$

$(1)\ S \to AA$

$(2)\ A \to aA$

$(3)\ A \to b$

States	Action			GoTo	
	a	b	\$	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		

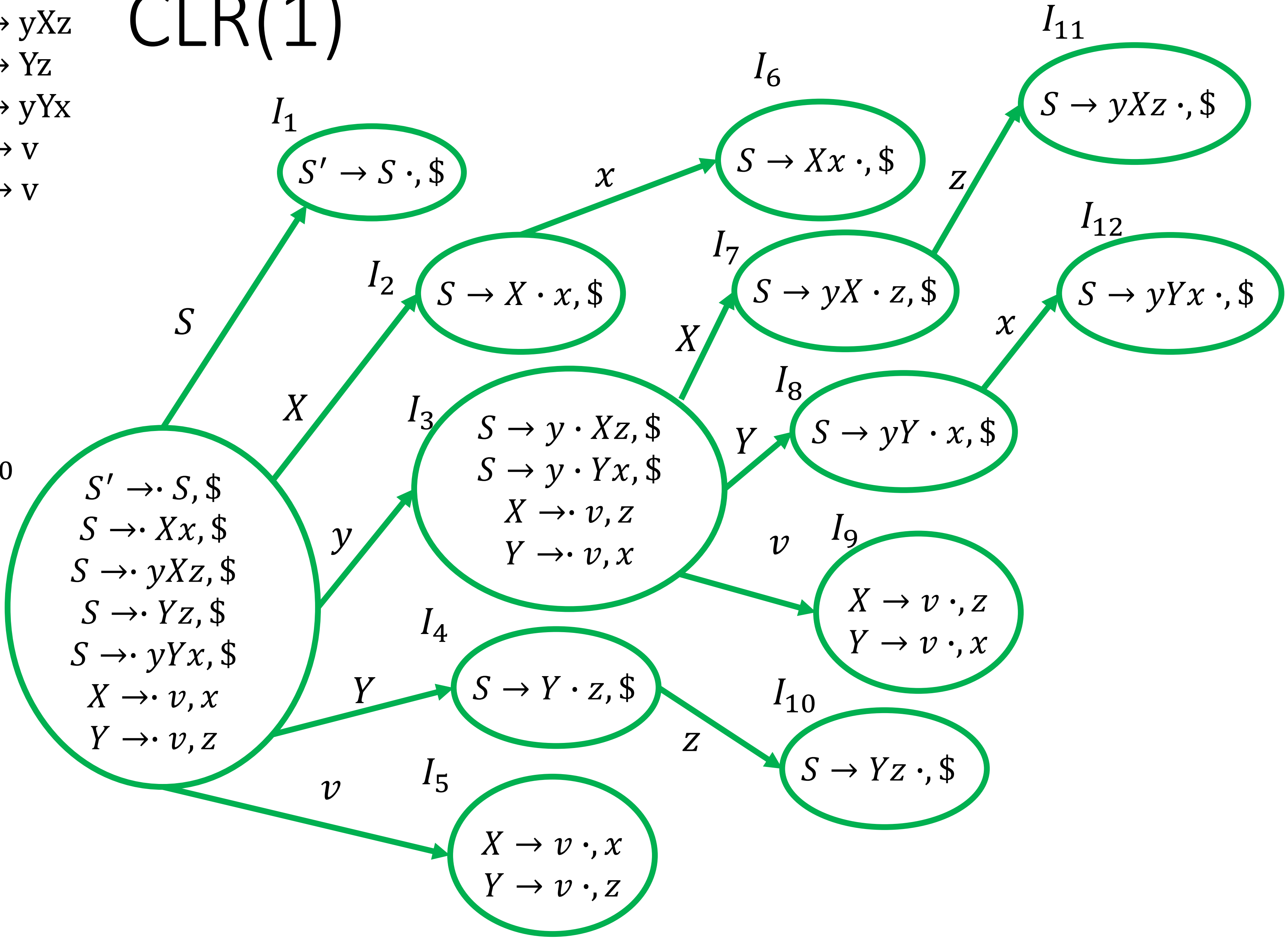


CLR(1) Example

- $S \rightarrow Xx \mid yXz \mid Yz \mid yYx$
- $X \rightarrow v$
- $Y \rightarrow v$

S → Xx
S → yXz
S → Yz
S → yYx
X → v
Y → v

CLR(1)



CLR(1)

1. S → Xx
2. S → yXz
3. S → Yz
4. S → yYx
5. X → v
6. Y → v

States	Action					GoTo		
	v	x	y	z	$	S	X	Y
I_0	s_5		s_3			1	2	4
I_1					Accept			
I_2		s_6						
I_3	s_9						7	8
I_4				s_{10}				
I_5		r_5		r_6				
I_6					r_1			
I_7				s_{11}				
I_8		s_{12}						
I_9		r_6		r_5				
I_{10}					r_3			
I_{11}					r_2			
I_{12}					r_4			

CLR(1)

Stack	Symbols	Input	Action
$\$\boldsymbol{I_0}$		yvz$	Shift
$\$\boldsymbol{I_0 I_3}$	y	vz$	Shift
$\$\boldsymbol{I_0 I_3 I_9}$	yv	z$	Reduce by $\mathrm{X} \rightarrow \mathrm{v}$
$\$\boldsymbol{I_0 I_3 I_7}$	yX	z$	Shift
$\$\boldsymbol{I_0 I_3 I_7 I_{11}}$	yXz	$	Reduce by $\mathrm{S} \rightarrow \mathrm{yXz}$
$\$\boldsymbol{I_0 I_1}$	S	$	Accept

CLR(1)

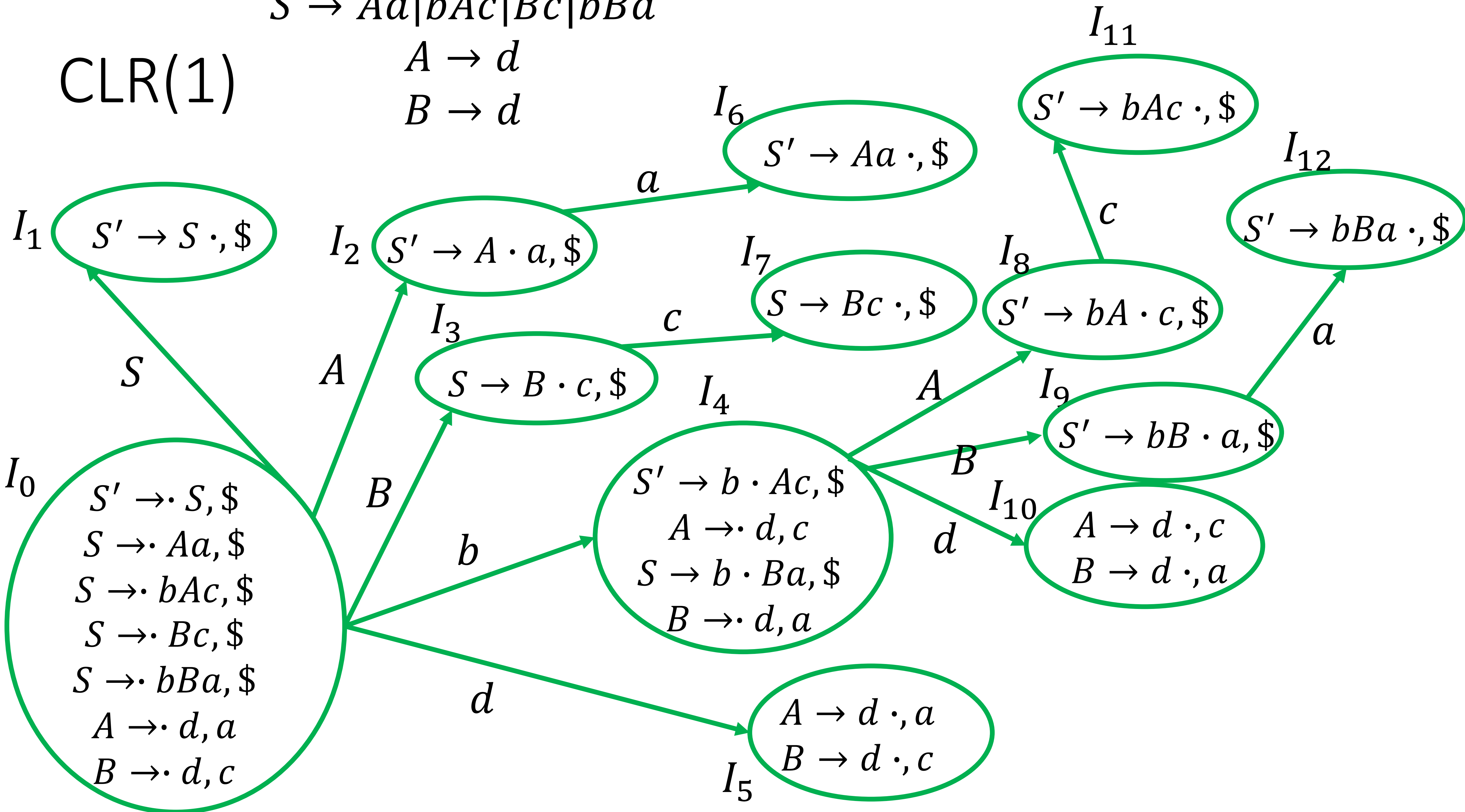
$$S \to Aa|bAc|Bc|bBa$$
$$A \to d$$
$$B \to d$$

CLR(1)

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

$A \rightarrow d$

$B \rightarrow d$



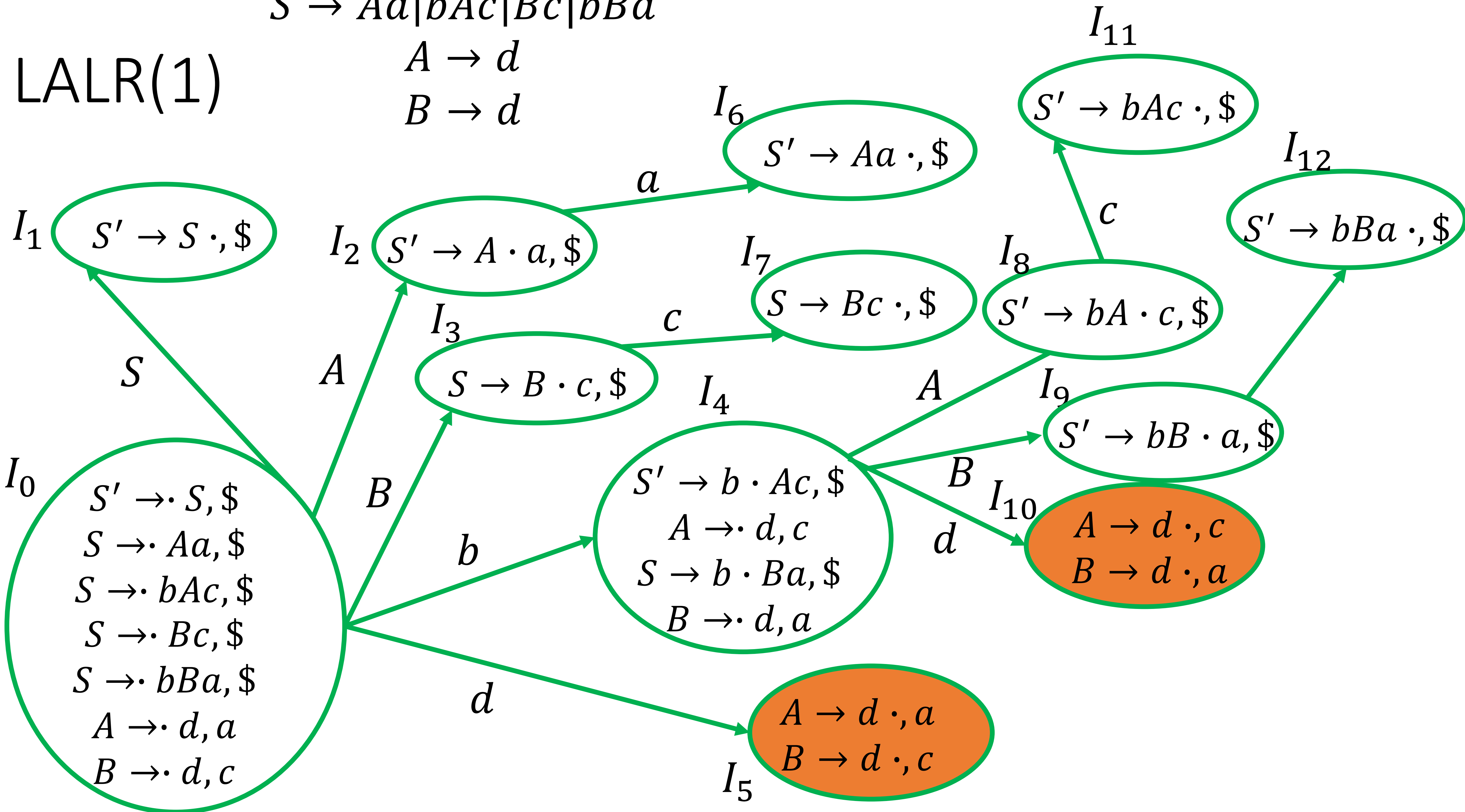
CLR(1)

$(1) S \rightarrow Aa$
$(2) S \rightarrow bAc$
$(3) S \rightarrow Bc$
$(4) S \rightarrow bBa$
$(5) A \rightarrow d$
$(6) B \rightarrow d$

States	Action					GoTo		
	a	**b**	**c**	**d**	**$**	**S**	**A**	**B**
I_0		s_4		s_5		1	2	3
I_1					Accept			
I_2	s_6							
I_3		s_7						
I_4				s_{10}			8	9
I_5	r_5		r_6					
I_6					r_1			
I_7					r_3			
I_8			s_{11}					
I_9	s_{12}							
I_{10}	r_6		r_5					
I_{11}					r_2			
I_{12}					r_4			

LALR(1)

$$S \to Aa|bAc|Bc|bBa$$
$$A \to d$$
$$B \to d$$



LALR(1)

$(1) S \rightarrow Aa$
$(2) S \rightarrow bAc$
$(3) S \rightarrow Bc$
$(4) S \rightarrow bBa$
$(5) A \rightarrow d$
$(6) B \rightarrow d$

States	Action					GoTo		
	a	b	c	d	$	S	A	B
I_0		s_4		s_5		1	2	3
I_1					Accept			
I_2	s_6							
I_3		s_7						
I_4				s_{10}			8	9
I_5	r_5		r_6					
I_6					r_1			
I_7					r_3			
I_8			s_{11}					
I_9	s_{12}							
I_{10}	r_6		r_5					
I_{11}					r_2			
I_{12}					r_4			

Example

- $S \rightarrow AaAb|BbBa$
- $A \rightarrow \epsilon$
- $B \rightarrow \epsilon$

Example

- $S \rightarrow (S)|a$

LR

- All grammars used in the construction of LR-parsing tables must be un-ambiguous.
- Can we create LR-parsing tables for ambiguous grammars?

– Yes, but they will have conflicts.

– We can resolve these conflicts in favor of one of them to disambiguate the grammar

LALR Exercise

- $S \rightarrow L = R$
- $S \rightarrow R$
- $L \rightarrow * R$
- $L \rightarrow id$
- $R \rightarrow L$

LALR Exercise

- $E \rightarrow T + E|T$
- $T \rightarrow T * F|F$
- $F \rightarrow id$