Contents

Module 4

Push down Automata: Definition of Pushdown Automata, The languages of PDA

Syntax Analysis Phase of the compiler:part-2- Bottom-up parsing, Introduction to LR Parsing's, More Powerful LR Parsers

4 Push down Automata

4.1 Introduction

Pushdown automata (PDA) are a type of automaton used in computer science and theoretical computer science to model and analyze processes involving stack-based memory. They are an extension of finite automata, which have limited memory, allowing PDAs to recognize a broader class of languages, including context-free languages.

Why pushdown automata are important:

Modeling Context-Free Languages: Context-free grammars are widely used in programming languages, parsing, and natural language processing. Pushdown automata provide a formalism for recognizing and generating strings in these languages, making them essential for understanding the structure of context-free languages.

Parsing: PDAs are used in parsing algorithms, such as the famous LR and LL parsers. These parsers are crucial components of compiler construction and other language-processing tasks. They help in analyzing the syntax of programming languages and ensuring that programs are correctly structured.

Expressive Power: Pushdown automata can recognize more languages than finite automata while maintaining a relatively simple structure. They strike a balance between expressiveness and ease of understanding, making them a valuable tool for studying computational complexity and language recognition.

Formal Language Theory: Pushdown automata play a central role in formal language theory, providing insights into the computational capabilities of various language classes. They are used to prove the properties about context-free languages and to establish relationships between different classes of languages.

Applications in Software Engineering: Beyond theoretical applications, pushdown automata have practical implications in software engineering. They are used in static code analysis, verification of software correctness, and designing efficient algorithms for language processing tasks.

Overall, pushdown automata are essential in both theoretical and practical aspects of computer science, providing a foundation for understanding and solving a wide range of problems related to formal languages and automata theory.

4.2 PDA Components

A pushdown automaton can be viewed as an extension of a finite state machine (FSM) with an added memory component called a **stack**. While a finite state machine has limited memory, a pushdown automaton can remember an unbounded amount of information using its stack.

In essence, a pushdown automaton combines the capabilities of a finite state machine with those of a stack, allowing it to recognize more complex languages than a finite state machine alone.

A pushdown automaton is − "Finite state machine" + "a stack"

A pushdown automaton has three components − **an input tape**, a **control unit**, and a **stack** with infinite size.

The stack head scans the top symbol of the stack.

A stack does two operations −

Push − a new symbol is added at the top.

Pop − the top symbol is read and removed.

A PDA may or may not read an input symbol, but it has to read the top of the stack in every transition.

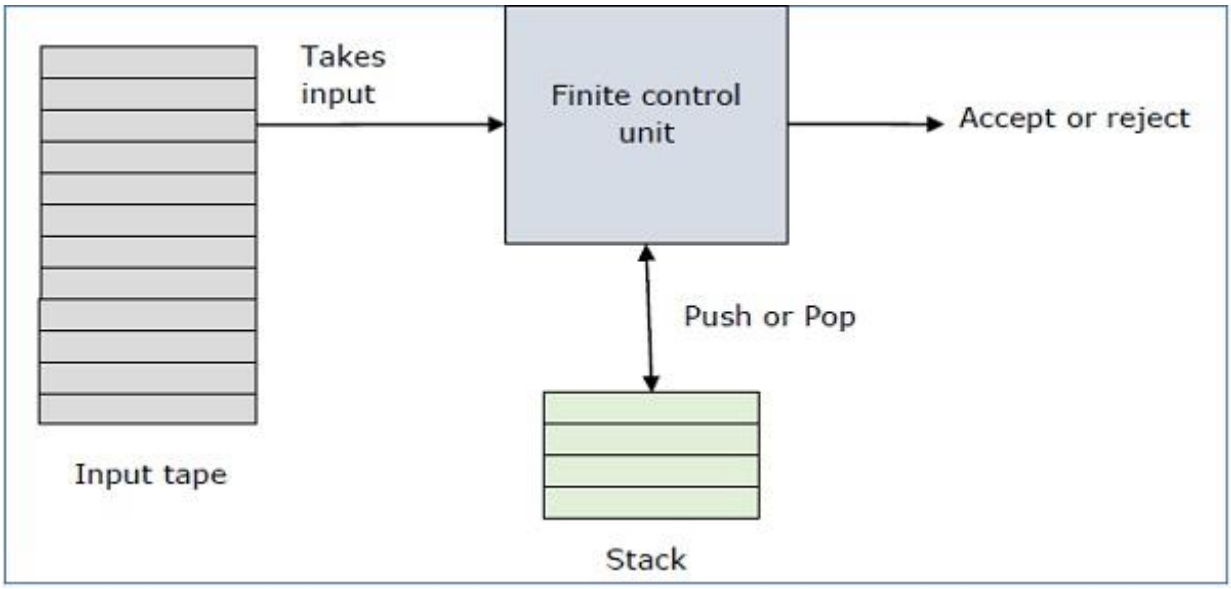


Figure: PDA Components

4.3 Formal Definition of PDA

A Pushdown Automaton (PDA) is formally defined as a **7-tuple (Q, Σ, Γ, δ, q0, Z, F),**

where:

Q is a finite set of states.

Σ is a finite set of input symbols (the input alphabet).

Γ is a finite set of stack symbols (the stack alphabet).

δ : **Q × (Σ ∪ {ε}) × Γ → (Q × Γ*)** is the transition function, where ε is the empty string and Γ* is the set of all strings over the stack alphabet. Each transition is of the form δ(q, a, X) = {(p, w)}, where q is the current state, a is the current input symbol (or ε for ε-transitions), X is the symbol at the top of the stack, p is the next state, and w is a string of stack symbols to be pushed onto the stack.

q0 ∈ Q is the initial state.

Z ∈ Γ is the initial stack symbol.

F ⊆ Q is the set of accepting states.

The PDA operates as follows:

At each step, it reads an input symbol from the input string.

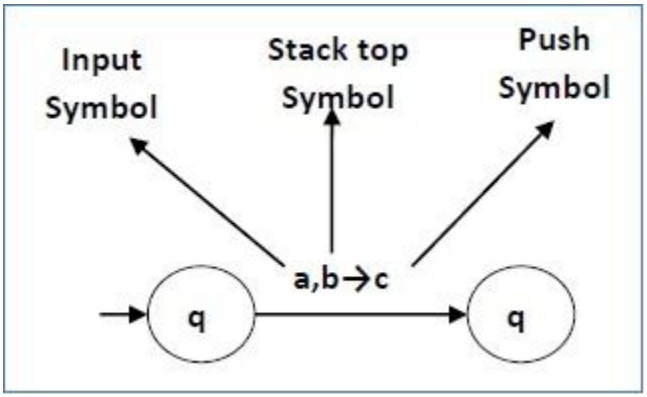
It consults the current state, the input symbol, and the symbol on top of the stack to determine the next state and the string of symbols to push onto the stack.

The stack operates in a last-in-first-out (LIFO) manner: symbols are pushed onto the stack, and the PDA can perform operations such as push, pop, or no operation based on the transition function.

The PDA accepts the input string if, after processing the entire input, it enters an accepting state.

Pushdown automata are more powerful than finite automata because they have access to an additional memory structure (the stack), allowing them to recognize languages that cannot be recognized by finite automata alone, such as context-free languages.

The following diagram shows a transition in a PDA from a state q1 to state q2, labeled as a,b → c



This means at state q1, if we encounter an input string 'a' and top symbol of the stack is 'b', then we pop 'b', push 'c' on top of the stack and move to state q2.

Fig: PDA Transitions

4.4 What is instantaneous description?

An Instantaneous Description (ID) in the context of a Pushdown Automaton (PDA) is a snapshot of the PDA's configuration at a particular point during its operation. It represents the state of the PDA, including the current state, the remaining input string to be processed, and the contents of the stack.

Formally, an Instantaneous Description of a PDA is typically denoted as a tuple ⟨q, w, γ⟩, where:

q is the current state of the PDA.

w is the remaining input string to be processed.

γ is the content of the stack.

An Instantaneous Description provides a way to track the progress of the PDA as it reads input symbols, transitions between states, and manipulates the stack. It is used to analyze the behavior of the PDA during its computation and to determine whether it accepts or rejects a given input string.

$$ID = (q, w, \gamma)$$

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

$$\delta(q_0, a, z_0) = (q_0, a, z_0)$$

$$\delta(q_0, a, a) = (q_0, a\,a\,z_0)$$

$$\delta(q_1, b, a) = (q, \epsilon)$$

$$\delta(q, \epsilon, z_0) = (q_2, \epsilon)$$

$(q_0, aabb, z_0) \vdash (q_0, abb, az_0)$
$\vdash (q_0, bb, aaz_0)$
$\vdash (q_1, b, az_0)$
$\vdash (q_1, \epsilon, z_0)$
$\vdash (q_2, \epsilon)$

find State $\leftarrow q_2$
Since we set a find state "aa bb" is accepted.

4.5 Representation of Push-down Automata (PDA)

The pushdown automaton is represented by the following types;

1. Transaction function of pushdown automata

2. Graphical Notation of pushdown automata (PDA)

4.5.1 Transaction function of push-down automata

The transaction function of pushdown automata has the following form;

$\delta: (Q \times (\Sigma \cup \{\varepsilon\}) \times X) \rightarrow (p, \gamma)$

δ is now a function of three arguments.

The first two arguments are the same as before:

(i) The state
(ii) The symbol of input alphabet or either 'ε' or 'λ'.
(iii) The third argument is the symbol on top of the stack. Just as the input symbol is 'consumed' when the function is applied, the stack symbol is also "consumed" (removed from the stack).

Note:

While the second argument may be ε, rather than a member of the input alphabet (so that no input symbol is consumed), there is no such option for the third argument.

δ always consumes a symbol from the stack, no move is possible if the stack is empty.

There may also be a ε-transition, where the second argument may be ε, which means that a move that does not consume an input symbol is possible. No move is possible if the stack is empty.

For example, let us consider the set of transition rules of a pushdown automaton given by

$\delta(q1, a, b) = \{(q2 , cd), (q3 , \varepsilon)\}$

If at any time the control unit is in state q1, the input symbol read is 'a', and the symbol on the top of stack is 'b', then one of the following two cases can occur:

- The control unit tends to go into the state q2 and the string 'cd' replaces 'b' on top of the stack.
- The control unit goes into state q3 with the symbol b removed from the top of the stack.

In the deterministic case, when the function δ is applied, the automaton moves to a new state $q \in Q$ and pushes a new string of symbols $x \in \Gamma^*$ onto the stack. As we are dealing with a nondeterministic pushdown automaton, the result of applying δ is a finite set of (q, x) pairs.

4.5.2 Graphical Notation of pushdown automata (PDA)

Pushdown automata are not usually drawn. However, with a few minor extensions, we can draw an PDA similar to the way we draw an finite automata. The graphical representation of the PDA's consists;

- The node corresponds to the states of the PDA.
- An arrow labeled Start indicates the start state, and doubly circled states are final or accepting as for finite automata.
- The arcs correspond to transition of the PDA in the following sense. An arc labeled $a|X, u$ from state q to state p means that $\delta(q, a, X)$ contains the pair (p, u), perhaps among other pairs. That is, the arc label tells what input is used, and also gives the old and new tops of the stack.

In short instead of labeling an arc with an element of ∑, we can label arcs with $a|x, y$ where $a \in \sum$, $x \in \Gamma$ and $y \in \Gamma^*$.

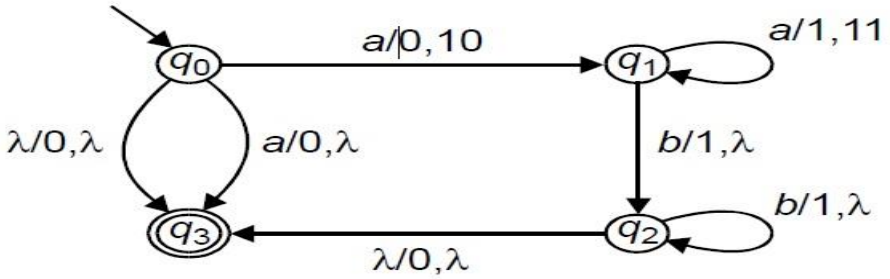
Let us consider the pushdown given by

$M = (Q=\{q0, q1, q2, q3\}, \sum=\{a,b\}, \Gamma=\{0,1\}, \delta, q0 , z0 = 0, F=\{q3\})$

Where;

$\delta(q0, a, 0) = \{(q1, 10), (q3, \varepsilon)\}$
$\delta(q0, \varepsilon, 0) = \{(q3, \varepsilon)\}$
$\delta(q1, a, 1) = \{(q1, 11)\}$
$\delta(q1, b, 1) = \{(q2, \varepsilon)\}$
$\delta(q1, b, 1) = \{(q2, \varepsilon)\}$
$\delta(q2, \varepsilon, 0) = \{(q3, \varepsilon)\}$

The Pushdown is drawn as follows;



Note: ε represented as lambda("λ")

(1) write a graphical notation for PDA transition

for $M = (Q, \epsilon, T, \delta, q_0, z_0, F)$

$\delta(q_0, a, z_0) = (q_0, a, z_0)$

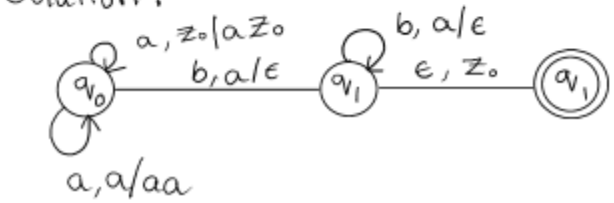
$\delta(q_0, a, a) = (q_0, aa)$

$\delta(q_0, b, a) = (q_1, \epsilon)$

$\delta(q_1, b, a) = (q_1, \epsilon)$

$\delta(q_1, \epsilon, z_0) = (q_\epsilon, z_0)$

Solution :-



(2) Construct PDA for a language $L=\{a^n b^n \mid n \geq 1\}$

$M=(Q, \Sigma, T, \delta, q_0, z_0, F)$

Solution :-

Procedure :-

Step 1 : Initially push all 'a's to the stack.

Step 2 : whenever 'b' occurs change the state & pop 'a' from the stack.

Step 3 : Repeat step 2 initial stack is empty.

Consider the language

Input string = | a | a | a | b | b | b | ∈ |

$L = \{ab, aabb, aaabbb \ldots\}$

$\delta(q_0, a, z_0) = (q_0, a, z_0)$

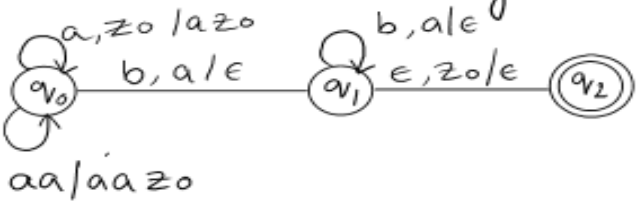
$\delta(q_0, a, a) = (q_0, aa\ z_0)$

$\delta(q_0, b, a) = (q_1, \epsilon)$

$\delta(q_1, b, a) = (q_1, \epsilon)$

$\delta(q_1, \epsilon, z_0) = (q_2, \epsilon)$

"aaabbb" accepted by PDA



$M=\{\{q_1, q_2, q_3\}, \{a, b\}, \{z_0, a\}\}$

(3) Construct PDA for the language

$L = \{a^n b^{2n} / n \geq 1\}$ $M = (Q, \epsilon, T, \delta, q_0, Z, F)$

Solution :-

The given language is

$L = \{abb, aabbbb, \ldots\}$

a	a	b	b	b	ϵ

Procedure :-

Step 1 : Initially if you read 'a' push 2b is to stack.

Step 2 : whenever 'b' occurs perform the pop 'a' from the stack and the change the State for 1st pop Operation.

Step 3 : Repeat step 2 initial stack in empty.

Consider Transition Function

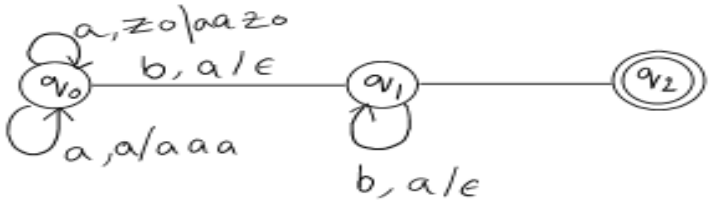
$\delta(q_0, a, Z_0) = (q_0, aa, Z_0)$

$\delta(q_0, a, a) = (q_0, aaa\, Z_0)$

$\delta(q_0, b, a) = (q_1, \epsilon)$

$\delta(q_1, b, a) = (q_1, \epsilon)$

$\delta(q_1, \epsilon, Z_0) = (q_2, \epsilon)$



$M = \{\{q_0, q_1, q_2\}, \{a, b\}, \{a, z_0\}, \delta, q_0, z_0, q_2\}$

(4) Construct PDA for a language

$L = \{wcw^R \mid w \in (a+b)^*\}$

Solution :-

The $M = \{a, \epsilon, T, \delta, q_0, z_0, f\}$

$L = \{abacaba, ababcbaba \ldots\}$

Procedure :-

Step 1 : push all symbols to the stack till 'c'

Step 2 : Don't perform any operation when 'c' exists

Step 3 : Perform pop operation after 'c' whenever 'a' matches 'a' and 'b' matches 'b'

Step 4 : Repeat step '3' Untill the stock empty

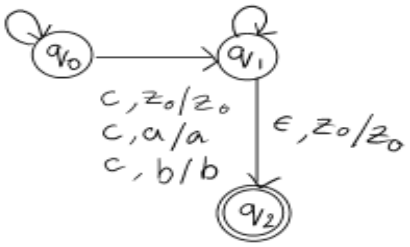
a b c c a b a

Transition functions

$\delta(q_0, a, z_0) = (q_0, az_0)$
$\delta(q_0, b, z_0) = (q_0, bz_0)$
$\delta(q_0, a, a) = (q_0, aa)$
$\delta(q_0, b, a) = (q_0, ba)$
$\delta(q_0, a, b) = (q_0, ab)$
$\delta(q_0, b, b) = (q_0, bb)$
$\delta(q_0, c, z_0) = (q_1, z_0)$
$\delta(q_0, c, a) = (q_1, a)$
$\delta(q_0, c, b) = (q_1, b)$
$\delta(q_0, a, a) = (q_1, \epsilon)$
$\delta(q_0, b, b) = (q_1, \epsilon)$
$\delta(q_1, a, a) = (q_1, \epsilon)$
$\delta(q_1, b, b) = (q_1, \epsilon)$
$\delta(q_1, \epsilon, z_0) = (q_1, z_0)$

$a, z_0/az_0$
$b, z_0/bz_0$
$a, a/aa$
$b, a/ba$
$a, b/ab$
$b, b/bb$

$a, a/\epsilon$
$b, b/\epsilon$



$M = \{q_0, q_1, q_2\}$,
$\{a, b, c\}$,
$\{a, b, z_0\}$
δ,
$q_0, z_0, q_2\}$

4.5.3 Acceptance of PDA

Acceptance by Final State: The PDA is said to accept its input by the final state if it enters any final state in zero or more moves after reading the entire input.

Let P = (Q, ∑, Γ, δ, q0, Z, F) be a PDA. The language acceptable by the final state can be defined as:

L(PDA) = {w | (q0, w, Z) ⊢* (p, ε, ε), q ∈ F}

Acceptance by Final stage

$M = (Q, \epsilon, T, \delta, q_0, z_0, F)$

Example :- $L = \{a^n b^n \mid n \geq 1\}$

For which

$\delta(q_0, a, z_0) = (q_0, a, z_0)$

$\delta(q_0, a, a) = (q_0, aa\, z_0)$

$\delta(q_0, b, a) = (q_1, \epsilon)$

$\delta(q_1, b, a) = (q_1, \epsilon)$

$\delta(q_1, \epsilon, z_0) = (q_2, z_0)$ or (q_1, ϵ)

$\delta(q_0, aabb, z_0) \vdash (q_0, abb, az_0)$

$\vdash (q_0, bb, aaz_0)$

$\vdash (q_1, b, az_a)$

$\vdash (q_1, b, z_0)$

$\vdash (q_2, z_0)$

$(q, w, z_0) \vdash^* (p, \epsilon, r)$

$P \in$ Final stage

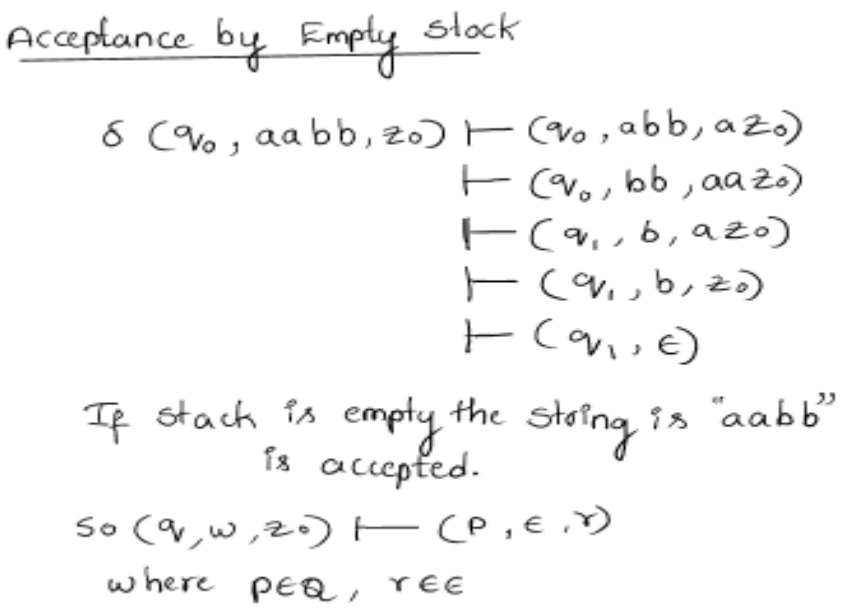
$r \in \vdash^* (P, \epsilon, r)$

∴ The language Accepted by PDA

Acceptance by Empty Stack: On reading the input string from the initial configuration for some PDA, the stack of PDA gets empty.

Let P =(Q, ∑, Γ, δ, q0, Z, F) be a PDA. The language acceptable by empty stack can be defined as:

N(PDA) = {w | (q0, w, Z) ⊢* (p, ε, ε), q ∈ Q}



4.6 Syntax Analysis Phase of the compiler

4.6.1 Bottom-up Parsing

As the name suggests, bottom up parsing works in the opposite direction from top down. A top-down parser begins with the start symbol at the top of the parse tree and works downward, driving productions in forward order until it gets to the terminal leaves. A bottom-up parse starts with the string of terminals itself and builds from the leaves upward, working backward to the start symbol by applying the productions in reverse. Along the way, a bottom-up parser searches for substrings of the working string that match the right side of some production. When it finds such a substring, it reduces it, i.e., substitutes the left side nonterminal for the matching right side. The goal is to reduce all the way up to the start symbol and report a successful parse.

In general, bottom-up parsing algorithms are more powerful than top-down methods, but not surprisingly, the constructions required are also more complex. It is difficult to write a bottom-up parser by hand for anything but trivial grammars, but fortunately, there are excellent parser generator tools like **Bison** that build a parser from an input specification

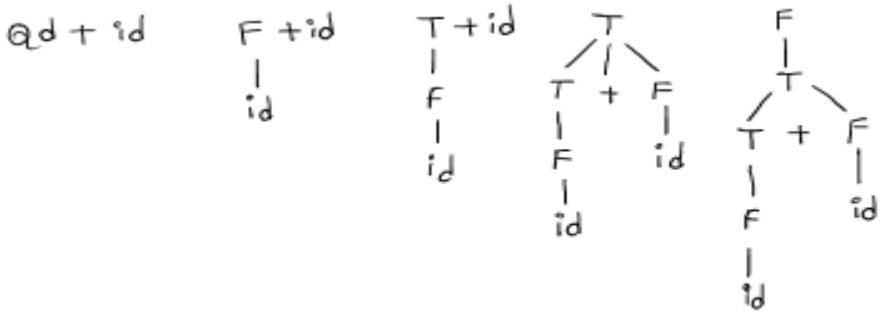


Figure :- Bottom – up parse for id + id

Example 1

$S \rightarrow aABe$

$A \rightarrow Abc \mid e$

$B \rightarrow d$

Input string "abbcde"

Solution :-

	Right Most Derivation
abbcde (A → b) ↳ Handle	
aAbcde (A → Abc)	S → aABe
aAde (B → d)	→ aAde
aABe (S → aABe)	→ aAbcde
S	→ abbcde

∴ The process of reducing input string to start Symbol is called as Bottom up parsing.

Example 2

$E \rightarrow E + T / T$

$Y \rightarrow T R F / F$

$F \rightarrow (E) / id$

Solution : id * id

id * id (F → id)

F * id (T → F)

T * id (F → id)

T * F (T → TAF)

T (F → T)

E → start symbol

4.6.1.1 Handle and Handle Pruning

In compiler design, a "handle" refers to the right-hand side of a production rule that is used during the process of constructing a parse tree or a derivation in the parsing phase, particularly in context-free grammars (CFGs). When performing bottom-up parsing, a handle is the substring of the input string that matches the right-hand side of a production, and it represents a portion of the input that can be reduced to the non-terminal on the left-hand side of that production.

Handle pruning, on the other hand, is a technique used in bottom-up parsing algorithms, such as LR (Left-to-right, Rightmost derivation) parsing, to identify and replace handles with the corresponding non-terminals. This process continues until the entire input string is reduced to the start symbol of the grammar, effectively constructing a parse tree or derivation.

Handle pruning involves identifying handles in the input string and replacing them with the non-terminals from the corresponding production rule. This step is crucial in the parsing process to ensure that the input string is correctly recognized according to the grammar rules defined by the language.

Example : 1

$S \rightarrow aABe$

$A \rightarrow Abc / b$

$B \rightarrow d$

Solution : Input string (abbcde)

Right Sentential Form	Handle	Reducing Production	Right most derivation
abbcde	b	A → b	S → a ABe
aAbcde	Abe	A → Abe	→ a Ade
aAde	d	B → d	→ a Abcde
a ABe	aABe	S → aABe	→ abbcde
S			

Example 2

E → E+T/T
T → T * F/F
F → (E)/id

Solution :- Input string = id * id

Right sential Form	Handle	Reducing Production
id * id	id	F → id
F * id	F	T → F
T * id	id	F → id
T * F	T×d	T → T * F
T	T	F → T
E		

Handle pruning

(The Process of right most derivation in reverse order)

4.6.1.2 Shift Reduce Parser

A shift-reduce parser is a type of bottom-up parsing algorithm used in compiler design to parse context-free grammars. It works by shifting input symbols onto a stack until it can apply a reduction (or handle pruning) using a production rule from the grammar. Shift-reduce parsers are particularly efficient and widely used because they can handle a broad class of grammars.

a shift-reduce parser works in terms of the four main steps: **shift**, **reduce**, **accept**, and **error**.

Shift:

- In the shift step, the parser reads the next input symbol from the input string.
- It then pushes this symbol onto the top of the stack.
- This step continues until the parser encounters a situation where it cannot proceed by shifting, typically because there are no applicable rules to reduce the symbols on the stack.

Reduce (Handle Pruning):

- When the parser cannot shift anymore, it tries to reduce the symbols currently on the stack if they match the right-hand side of any production rule in the grammar.
- If a match is found, the parser applies the corresponding production rule, replacing the symbols on the stack with the non-terminal symbol on the left-hand side of the production rule.
- This reduction step continues until the parser cannot apply any more reduction rules.

Accept:

- After processing the entire input string and successfully reducing it to the start symbol of the grammar, the parser accepts the input.
- This means that the input string is syntactically correct according to the grammar rules defined.

Error:

- If the parser encounters a situation where it cannot shift or reduce and there are still symbols remaining on the stack, or if it reaches the end of the input string without successfully reducing it to the start symbol, it indicates a syntax error.
- Common syntax errors include unexpected symbols, missing symbols, or invalid combinations of symbols according to the grammar rules.

Overall, the shift-reduce parsing process involves a combination of shifting input symbols onto the stack and reducing symbols on the stack using production rules until the input string is either accepted as syntactically correct or an error is encountered.

Example 1

$E \rightarrow E + T$

$T \rightarrow T * F / F$

$F \rightarrow (E) / id$

Solution :- id * id

Stack	Input Buffer	Action
$	id * id $	Shift
$ id	* id $	Reduced by F → id
$ F	* id $	Reduced by T → F
$ T	* id $	shift
$ T *	id $	shift
$ T * id	$	Reduced F → id
$ T * F	$	Reduced T → F * F
$ T	$	Reduced T → ∈
$ E	$	Accept

Example 2

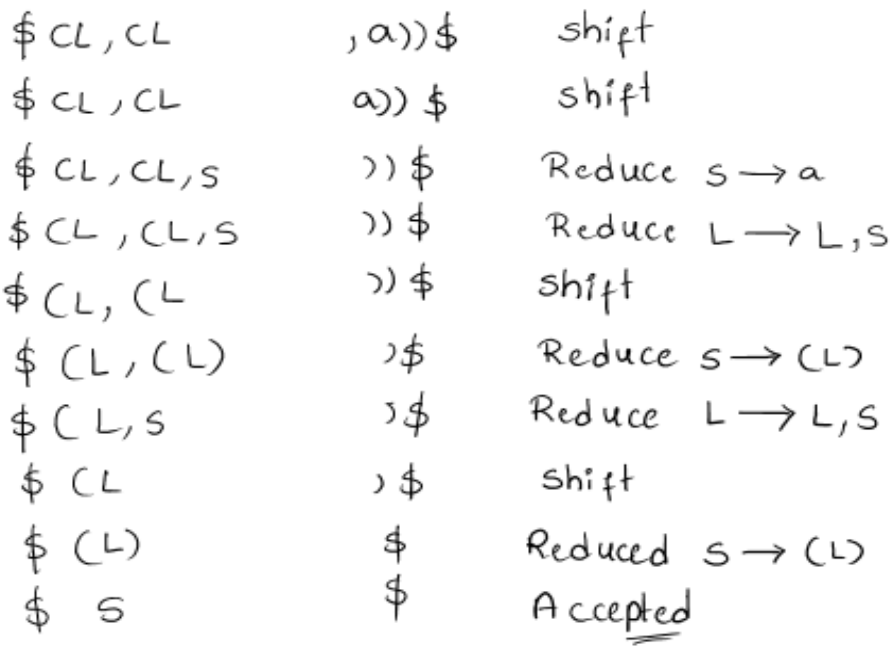
Consider the following grammer

S → (L) / a

L → L, S / S

- Parse the input string (a, (a, a)) using shift reduce parser

$ (L, (L	, a)) $	shift
$ (L, (L	a)) $	shift
$ (L, (L, S	)) $	Reduce S → a
$ (L, (L, S	)) $	Reduce L → L, S
$ (L, (L	)) $	shift
$ (L, (L)	) $	Reduce S → (L)
$ (L, S	) $	Reduce L → L, S
$ (L	) $	shift
$ (L)	$	Reduced S → (L)
$ S	$	Accepted



4.6.1.3 shift-reduce parser conflict

A shift-reduce conflict occurs in a shift-reduce parser when there is ambiguity in the parsing table, leading to uncertainty about whether to shift an input symbol onto the stack or to reduce symbols on the stack using a production rule. In simpler terms, it's a situation where the parser cannot decide whether to shift or reduce based solely on the current state and lookahead symbol.

Here's an example to illustrate a shift-reduce conflict:

Consider a grammar with the following production rules:

E → E + E

E → id

Now, let's say we have the input string "id + id + id".

At some point during parsing, the parser may encounter a shift-reduce conflict. Let's look at a specific situation:

The parser has already shifted "id + id" onto the stack.

The next symbol to consider is "+", indicating a possible addition operation.

At this point, the parser faces a choice:

It can either shift the "+" onto the stack, expecting to reduce it later.

Or it can reduce "id + id" to "E", following the production rule E → E + E.

The conflict arises because the parser cannot determine whether to shift the "+" symbol onto the stack or to reduce "id + id" to "E".

To resolve shift-reduce conflicts, parser generator tools may use precedence and associativity rules specified in the grammar. Additionally, manual adjustments to the grammar or the introduction of explicit disambiguation rules can help resolve conflicts. These adjustments guide the parser in making the correct decisions when faced with ambiguous situations during parsing.

4.6.1.4 reduce-reduce parser conflict

A reduce-reduce conflict is another type of conflict that can occur in a shift-reduce parser. Unlike a shift-reduce conflict, which involves a choice between shifting an input symbol onto the stack or reducing symbols on the stack, a reduce-reduce conflict arises when the parser encounters ambiguity about which production rule to use for reduction.

Here's an example to illustrate a reduce-reduce conflict:

Consider a grammar with the following production rules:

E → id + id

E → id * id

Now, let's say we have the input string "id + id * id".

At some point during parsing, the parser may encounter a reduce-reduce conflict. Let's look at a specific situation:

The parser has already shifted "id + id" onto the stack.

The next symbol to consider is "*", indicating a possible multiplication operation.

At this point, the parser faces a choice:

It can reduce "id + id" to "E" using the production rule E → id + id.

Or it can reduce "id * id" to "E" using the production rule E → id * id.

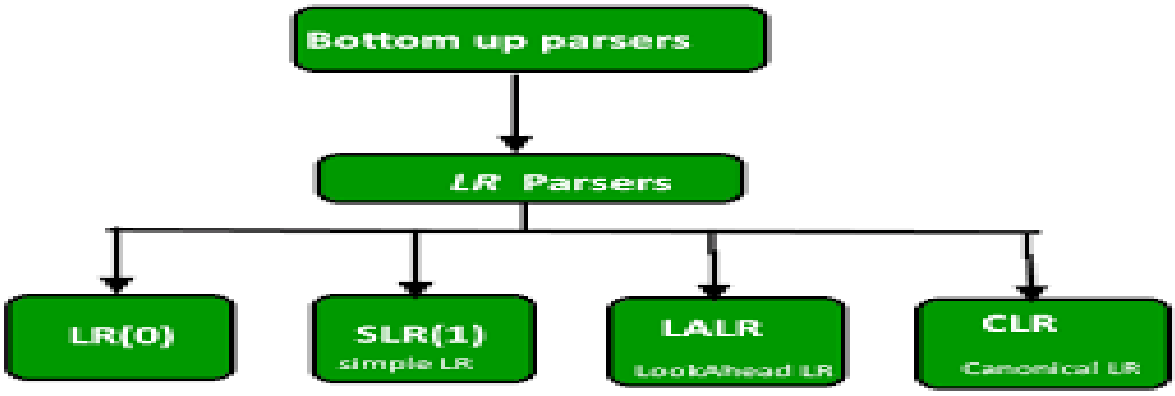
The conflict arises because the parser cannot determine which production rule to use for reduction, leading to ambiguity about how to proceed.

Reduce-reduce conflicts typically indicate a flaw in the grammar, where multiple production rules can be applied to the same sequence of symbols. To resolve reduce-reduce conflicts, you may need to revise the grammar to remove ambiguity, introduce additional context to disambiguate, or prioritize certain rules over others using precedence and associativity declarations.

4.6.1.5 LR PARSING

LR parsing is a bottom-up parsing technique used in compiler design to analyze and recognize the structure of input strings based on context-free grammar. It stands for "Left-to-right, Rightmost derivation" parsing. In LR parsing, the input string is scanned from left to right, and a rightmost derivation of the input string is built in reverse, starting from the start symbol of the grammar. At each step, the parser applies shift and reduce operations guided by a parsing table constructed from the grammar.

LR parsing is classified into different types based on the construction of the parsing table and the lookahead symbols used during parsing. Here are the main classifications:



- **LR(0):** In LR(0) parsing, the parser makes decisions based solely on the current state of the parser and does not consider any lookahead symbols. This can lead to conflicts, such as shift-reduce and reduce-reduce conflicts, in ambiguous grammars.
- **SLR (Simple LR):** SLR parsing improves upon LR(0) parsing by using a simplified parsing table that resolves some shift-reduce conflicts. It does this by considering only a subset of the lookahead symbols.
- **LALR (Look-Ahead LR):** LALR parsing further improves upon SLR parsing by using a more compact parsing table that merges states with similar cores, resulting in fewer states and a smaller table. It achieves this by sharing states that have identical cores but differ only in their lookahead sets.
- **CLR(1):** CLR(1), which stands for Canonical LR(1), is an extension of LR(1) parsing. It is a more formal and rigorous approach to LR(1) parsing, aiming to simplify and organize the parsing process. CLR(1) parsing utilizes a canonical collection of LR(1) parsing states, which are constructed to capture all possible LR(1) configurations of the parsing process.

Each type of LR parsing has its advantages and disadvantages in terms of table size, parsing efficiency, and the class of grammars it can handle. SLR and LALR parsers are commonly used in practice due to their balance between simplicity and power, while LR(1) parsers are used for more complex grammars where additional lookahead is necessary to resolve parsing ambiguities.

Why LR Parsing?

LR parsers are favored in compiler design and parsing for several reasons:

- **Powerful and General**: LR parsers are capable of parsing a wide range of context-free grammars, including those that cannot be parsed by simpler parsing techniques like LL parsers. This makes LR parsing suitable for parsing many programming languages, which often have complex grammatical structures.
- **Efficient**: LR parsing is typically more efficient than other parsing techniques, such as LL parsing, especially for larger grammars. LR parsers have linear-time complexity for parsing, meaning that the time taken to parse an input string is proportional to the length of the string.

- **Bottom-Up Parsing**: LR parsing is a bottom-up parsing technique, which means that it starts parsing from the input symbols and builds up to the start symbol of the grammar. Bottom-up parsing can often result in better error recovery and better handling of left-recursive grammars compared to top-down parsing techniques.
- **Automatic Construction**: LR parsers can be automatically generated from a given context-free grammar using parser generator tools like YACC (Yet Another Compiler Compiler) or Bison. These tools take a grammar specification as input and produce the corresponding LR parsing tables and parser code.
- **Error Reporting**: LR parsers can provide detailed error messages when syntax errors are encountered during parsing. This is particularly useful for compiler writers and developers, as it helps them quickly identify and fix issues in their code.
- **Widely Used**: LR parsing has been extensively studied and is widely used in practice. Many programming languages, such as C, C++, Java, and Python, are parsed using LR-based parsers.

Overall, LR parsers offer a powerful, efficient, and widely applicable approach to parsing context-free grammars, making them a popular choice in compiler design and related fields.

Figure: Block diagram of LR Parser

The input buffer is used to indicate end of input and it contains the string to be parsed followed by a $ Symbol.

A stack is used to contain a sequence of grammar symbols with a $ at the bottom of the stack.

The parsing table is a two-dimensional array. It contains two parts: The **action** part and **GoTo** part

LR(0) Parsing: The following are the steps of the LR Parser

Step 1: For the given input string write a context-free grammar.

Step 2: Check the ambiguity of the grammar.

Step 3: Add Augment production in the given grammar.

Step 4: Create a Canonical collection of LR (0) items.

Step 5: Draw a data flow diagram (DFA).

Step 6: Construct a LR (0) parsing table.

Note:

- ***All parsers having the same structure as LR Parser only difference in the Parsing Table(ACTION and GO)***
- ***To construct LR(0) and SLR(1) tables we can use canonical collection of LR(0) items***
- ***To construct LALR(1) and CLR(1) tables we can use canonical collection of LR(1) items***

Step 3: Augment Grammar

Augmenting grammar involves adding a new start symbol and a new production rule to the existing grammar. This process is typically done to ensure that the grammar has a unique start symbol and to make it easier to construct parsers using certain parsing techniques such as LR parsing.

Select a New Start Symbol:

- Choose a new non-terminal symbol that does not already exist in the original grammar. This symbol will be the new start symbol for the augmented grammar.

Add a New Production Rule:

- Create a new production rule that uses the new start symbol to derive the original start symbol of the grammar.
- This new rule effectively makes the new start symbol the parent of the original start symbol.

For example :

S -> A

A -> aA | ε

After Augmenting

S' -> S

S -> A

A -> aA | ε

Step 4: Canonical collection LR(0) items creation

An LR (0) item is a production G with dot at some position on the right side of the production.

LR(0) items is useful to indicate that how much of the input has been scanned up to a given point in the process of parsing.

In the LR (0), we place the reduced node in the entire row.

For example:

Given grammar:

S → AA

A → aA | b

Augment Production and insert '•' symbol at the first position for every production in G

S` → •S

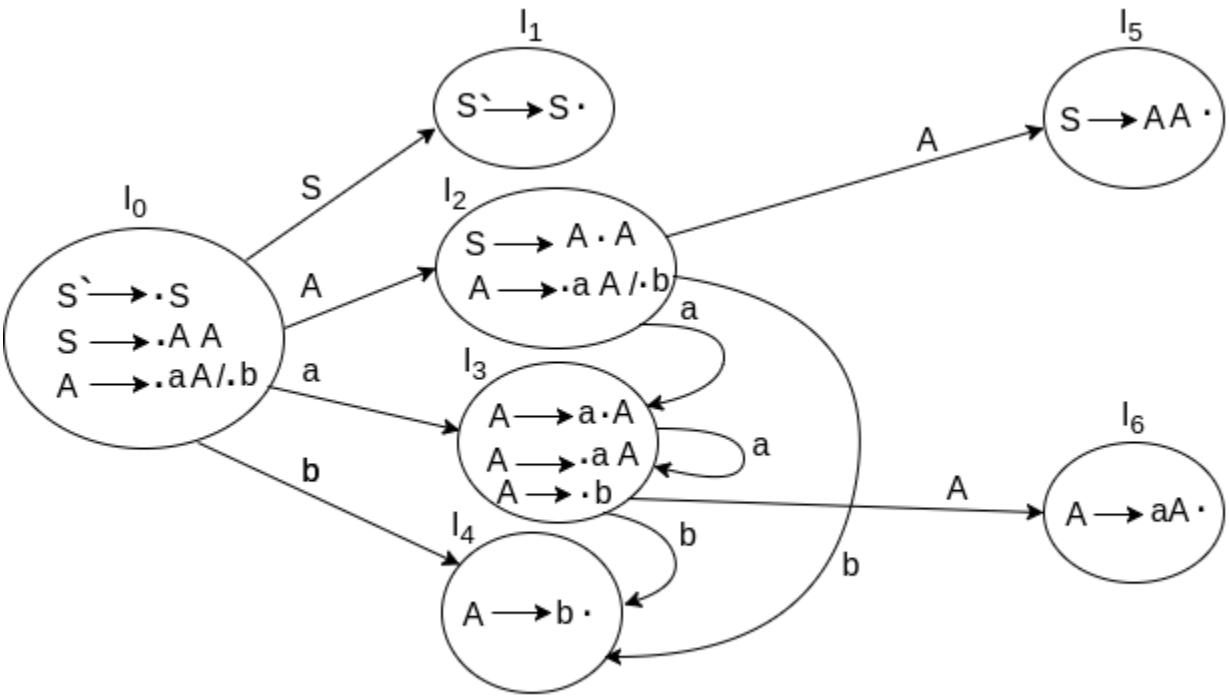
S → •AA

A → •aA

A → •b

Step 5: Drawing DFA

Assign state numbers to each set of LR(0) items.



Step 6: LR(0) Parsing Table

- If a state is going to some other state on a terminal then it corresponds to a shift move.
- If a state is going to some other state on a variable then it corresponds to go to move.
- If a state contains the final item in the particular row then write the reduce node completely.

States	Action			Go to	
	a	b	$	A	S
I_0	S3	S4		2	1
I_1			accept		
I_2	S3	S4		5	
I_3	S3	S4		6	
I_4	r3	r3	r3		
I_5	r1	r1	r1		
I_6	r2	r2	r2		

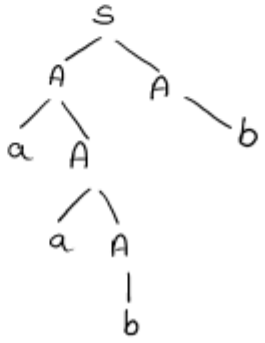
Action: shift (terminals)

GOTO: move(non-terminal/variables)

Parsing Input string (aabb)

Steps	Parsing stack	i/p	Action
1	$0	aabb$	shift a3
2	$0a3	abb$	shift a3
3	$0a3a3	bb$	Shift b4
4	$0a3a3b4	b$	reduce r_3 (A→b)
5	$0a3a3A6	b$	reduce r_2 (A→aA)
6	$0a3A6	b$	reduce r_2 (A→aA)
7	$0A2	b$	Shift b4
8	$0A2b4	$	reduce r_3 (A→b)
9	$0A2A5	$	reduce r_3 (A→AA)
10	$0S1	$	Accept

Bottom-up :- parse tree



4.6.1.6 SLR PARSING

SLR (1) refers to simple LR Parsing. It is same as LR(0) parsing. The only difference is in the parsing table. To construct SLR (1) parsing table, we use canonical collection of LR (0) item.

In the SLR (1) parsing, we place the reduce move only in the FOLLOW of left-hand side.

Various steps involved in the SLR (1) Parsing:

Step 1: For the given input string write a context free grammar

Step 2: Check the ambiguity of the grammar

Step 3: Add Augment production in the given grammar

Step 4: Create Canonical collection of LR (0) items

Step 5: Draw a data flow diagram (DFA)

Step 6: Construct a SLR (1) parsing table

① Construct CLR parsing table for below Grammar

$A \rightarrow (A)/a$

Solution :-

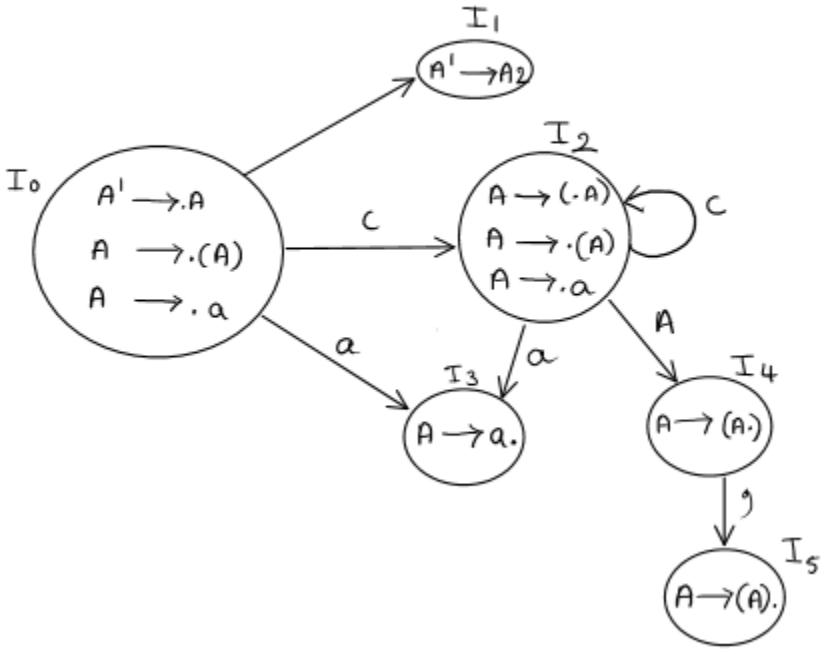
Augmented Grammar

$A' \rightarrow .A$

$A \rightarrow .(A)$

$A \rightarrow .a$

DFA Diagram



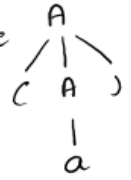
Parsing table

State	Action				GOTO
	a	(	)	$	A
I_0	S_3	S_2		Accept	1
I_1					
I_2	S_3	S_2			4
I_3			r_2	r_2	
I_4			S_5		
I_5			r_1	r_1	

Input string : (a) $

Step	parsing table	I/P	Action
1	$0 ↑	(a)$ ↑	shift (2
2	$0(2	a)$	Shift a3
3	$0(2a3	)$	Reduce r_2 ($A \to a$)
4	$0(2A4	)$	shift)5
5	$0(2A)5	$	reduced by r_1, ($A \to (A)$)
6	$0A	$	Accept

Parse



4.6.1.7 CLR(1) PARSING

- CLR refers to loose ahead. CLR parsing use the 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 SLR (1) parsing.
- In CLR(1), we place reduce node only in the look ahead symbols.

STEPS

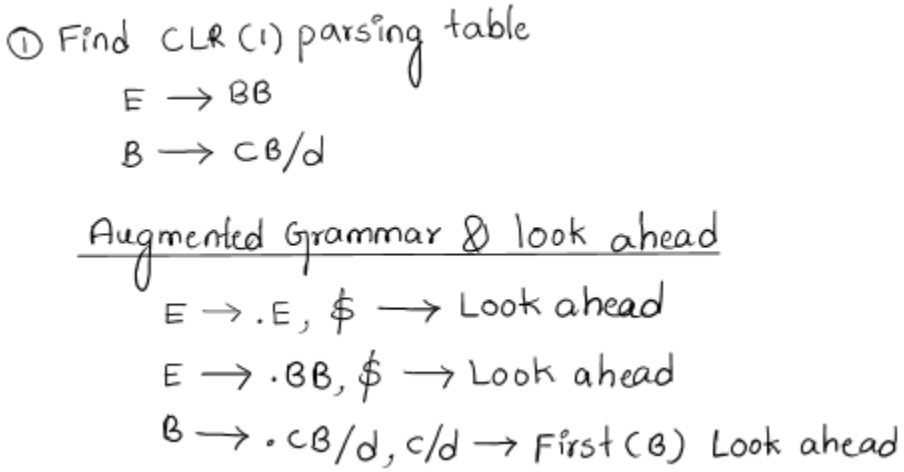
- For the given input string write a context free grammar
- Check ambiguity of the grammar
- Add augment production in the given grammar
- Erase canonical collection of LR (1) items
- Draw a data flow diagram (DFA)
- Construct a CLR (1) parsing table.

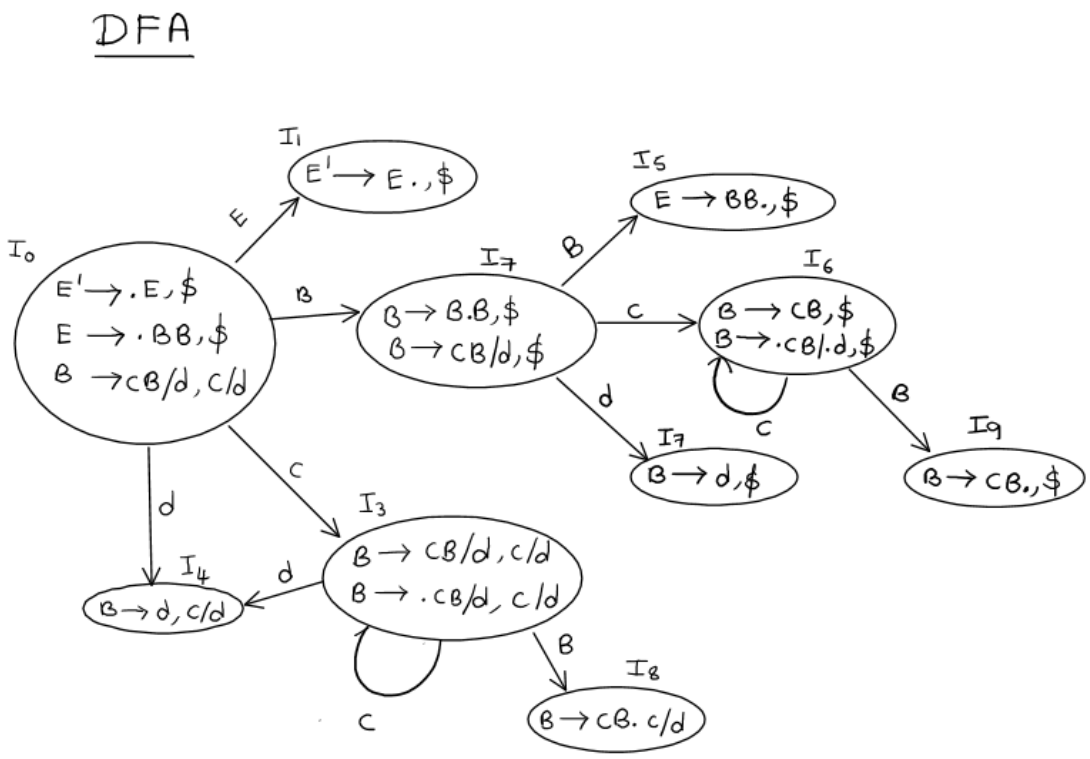
LR (1) items

LR (1) Item is a collection of LR (0) item and a look a head symbol.

LR (1) item = LR(0) item + look a head

- The look ahead is used to determine that where we place the final item.
- The look ahead always add & symbol for the augment production.





4.6.1.8 LALR PARSING

LALR (1)

LALR refers to the look ahead LR. To construct the LALR (!) parsing table, we use the canonical collection of LR (10 items.

In the LALR (1) parsing , the LR (1) items which have same productions but different look ahead are combined form a single set items.

LALR (1) is same as the CLR (1) parsing, only difference in the parsing in the parsing table

CLR(1) Parsing table

$E \rightarrow .BD$ — (1)
$B \rightarrow cB$ — (2)
$B \rightarrow d$ — (3)

STATE	ACTION			GO TO	
	c	d	$	E	B
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			
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		

Grammar

$E \rightarrow B.B$
$B \rightarrow cB/d$

$\Rightarrow$

$E' \rightarrow .E, \$$
$E \rightarrow .BB, \$$
$B \rightarrow .cB/d, c/d$

defference Look ahead value

If combine $I_3, I_6 \Rightarrow I_{36}$
$I_4, I_7 \Rightarrow I_{47}$
$I_8, I_9 \Rightarrow I_{89}$

} Same production + Difference in Look ahead

STATE	ACTION			GO TO	
	C	d	$	E	B
I_0	S_{36}	S_{47}		1	2
I_1			Accept		
I_2	S_{36}	S_{47}			5
I_3	S_{36}	S_{47}			89
I_4	r_{36}	r_3			
I_5			r_1		
I_{36}	S_6	S_{47}			89
I_{47}			r_3		
I_{89}	r_2	r_2			
I_{89}			r_2		

STATE	ACTION			GO TO	
	C	d	$	E	B
I_0	S_3	S_4		1	2
I_1			Accept		
I_{36}	S_{36}	S_{47}			89
I_{47}	r_3	r_3 r_3			
I_5			r_1		
I_{85}	r_2	r_2	r_2		