Parul® University

COMPILER DESIGN
SUBJECT CODE: 203105351

Prof. Kapil Dev Raghuwanshi, Assistant Professor
Computer Science & Engineering

Parul®
University

DIGITAL LEARNING CONTENT

CHAPTER-3

Top-down parsing

What is Parsing

Parsing: is that phase of compiler which takes token string as input and with the help of existing grammar, converts it into the corresponding parse tree.
-> Parser is also known as Syntax Analyzer

➢Parsing technique is divided in two Types.

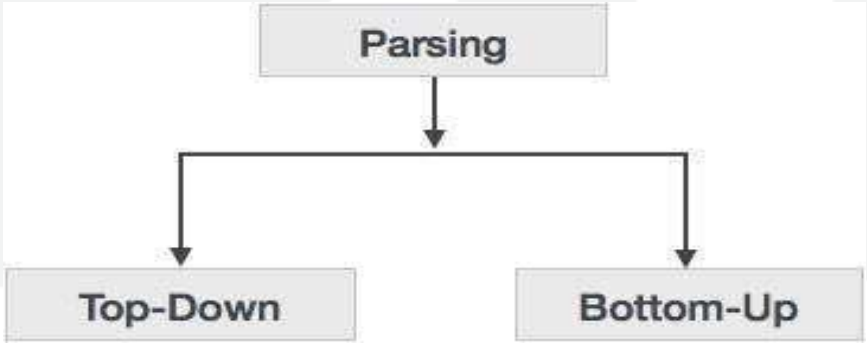
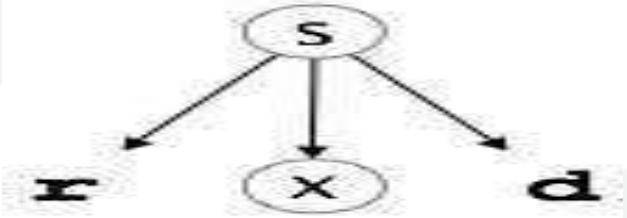


Image source : Google

Top - Down Parsing:

Top-down parser: is the parser which generates parse for the given input string with the help of grammar productions by expanding the non-terminals i.e. it starts from the start symbol and ends on the terminals.

- It uses left most derivation



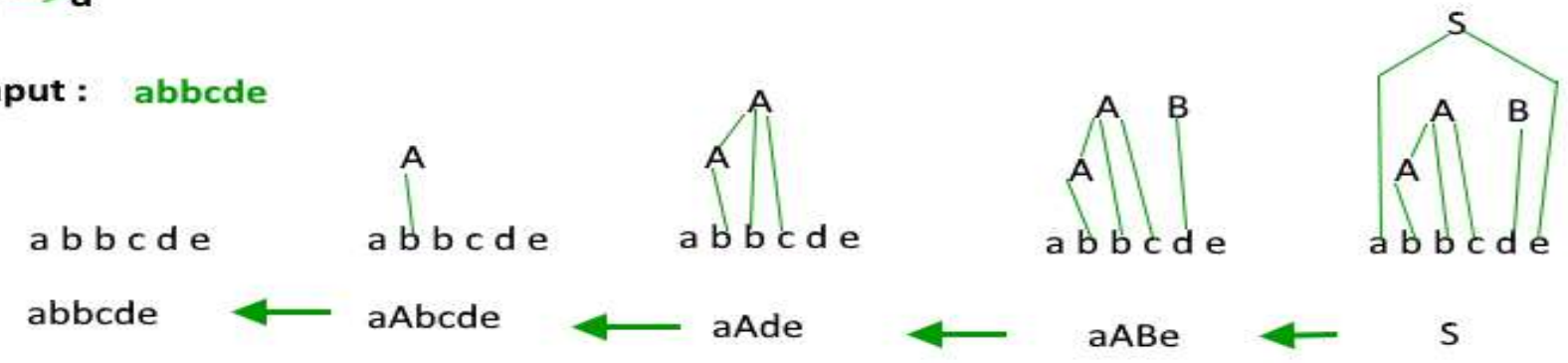
Bottom-Up/Shift Reduce parser : Bottom-up Parser is the parser which generates the parse tree for the given input string with the help of grammar productions by compressing the non-terminals i.e. it starts from non-terminals and ends on the stat symbol

- It uses reverse of the right most derivation.

Bottom-Up parser

S → aABe
A → Abc/b
B → d

Input : abbcde



Top-Down and Bottom-Up parser are further divided in following types

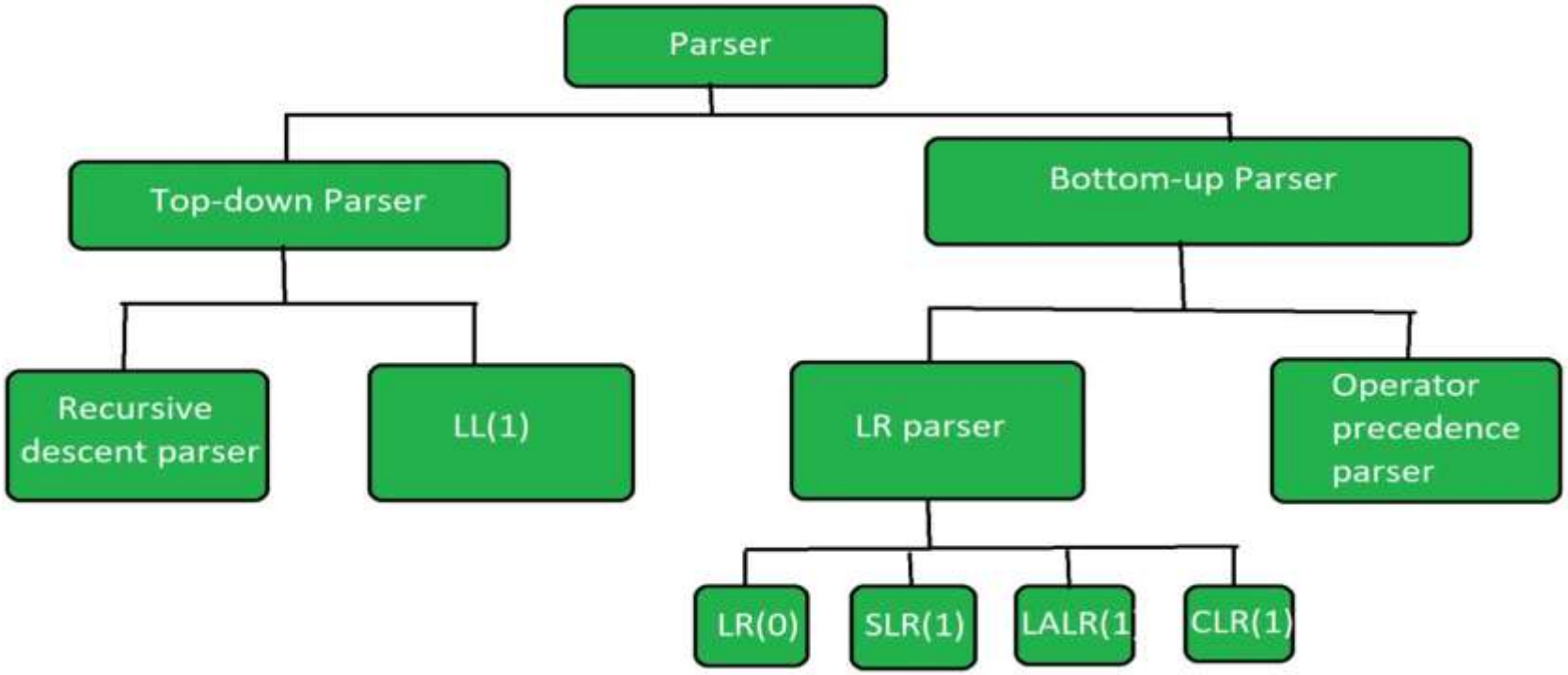


Image source : Google

Recursive Descent parser:

It is also known as Brute force parser or the with backtracking parser. It basically generates the parse tree by using brute force and backtracking.

Non Recursive Descent Parser OR LL (1):

It is also known as LL(1) parser or predictive parser or without backtracking parser or dynamic parser. It uses parsing table to generate the parse tree instead of backtracking.

LR parser:

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

Operator Precedence Parser:

It generates the parse tree form given grammar and string but the only condition is two consecutive non-terminals and epsilon never appears in the right-hand side of any production.

Parul®
University

LR-Parser:

- A general shift reduce parsing is LR parsing.
- The L stands for scanning the input from left to right and R stands for constructing a rightmost derivation in reverse

LR –parser are of following types:

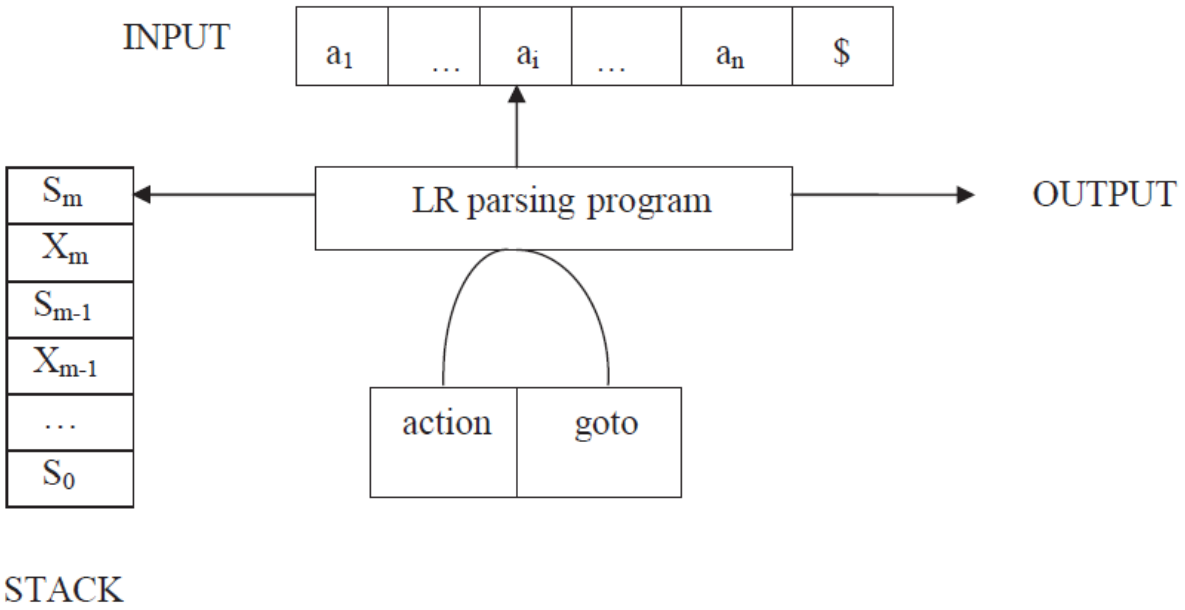
(a). LR(0)
(b). SLR(1)
(c). LALR(1)
(d). CLR(1)

Parul® University

LR-Parser:

The LR parsing algorithm:

The schematic form of an LR parser is as follows



Parul® University

LR-Parser:

It consists of : an input, an output, a stack, a driver program, and a parsing table that has two parts (*action* and *goto*).

The driver program is the same for all LR parser.

The parsing program reads characters from an input buffer one at a time.

The program uses a stack to store a string of the form $s_0X_1s_1X_2s_2\ldots X_ms_m$, where s_m is on top. Each X_i is a grammar symbol and each s_i is a state.

The parsing table consists of two parts :*action* and *goto* functions.

Parul® University

LR-Parser:

Action: The parsing program determines s_m, the state currently on top of stack, and a_i, the current input symbol. It then consults *action*[s_m,a_i] in the action table which can have one of four values :

shift s, where s is a state,

reduce by a grammar production $A \rightarrow \beta$,

accept, and

error.

Goto: The function goto takes a state and grammar symbol as arguments and produces a state.

Parul® University

LR Parsing algorithm:

Input: An input string*w*and an LR parsing table with functions*action*and*goto*for grammar G.

Output: If*w*is in L(G), a bottom-up-parse for*w*; otherwise, an error indication.

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

Parul®
University

LR Parsing algorithm:

Set *ip* to point to the first input symbol of *w*$;
repeat forever begin
Let *s* be the state on top of the stack and
A the symbol pointed to by *ip*;
If *action*[*s*,*a*] = shifts' **then begin** push *a* thens' on top of the stack; advance *ip* to the next input symbol
end
else if *action*[*s*,*a*] = reduce A→β **then begin**
pop 2* | β | symbols off the stack;
lets' be the state now on top of the stack; push A then *goto*[*s*', A] on top of the stack;
output the production A→ β
end
else if *action*[*s*,*a*] = accept **then return**
Else *error*( )
end

Parul®
University

LR-Parser:

CONSTRUCTING SLR(1) PARSING TABLE:

To perform SLR parsing, take grammar as input and do the following:

1. Find LR(0) items.
2. Completing the closure.
3. Compute *goto*(I,X), where, I is set of items and X is grammar symbol.

Parul® University

LR-Parser:

LR(O) items:

An *LR(O) item* of a grammar G is a production of G with a dot at some position of the right side. For example, production A → XYZ yields the four items :

A →**.**XYZ

A → X**.**YZ

A → XY**.**Z

A → XYZ**.**

Parul® University

LR-Parser:

Closure operation:

If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the two rules:

Initially, every item in I is added to closure(I).

If A → a . Bβ is in closure(I) and B → y is a production, then add the item B → . y to I , if it is not already there.

We apply this rule until no more new items can be added to closure(I).

Parul®
University

LR-Parser:

Goto operation:

Goto(I, X) is defined to be the closure of the set of all items [A→ aX . β] such that [A→ a . Xβ] is in I.

Parul® University

LR-Parser:

Steps to construct SLR parsing table for grammar G are:

1. Augment G and produce G'
2. Construct the canonical collection of set of items C for G'
3. Construct the parsing action function action and goto using the following algorithm that requires FOLLOW(A) for each non-terminal of grammar

Parul® University

LR-Parser:

Algorithm for construction of SLR parsing table:

Input: An augmented grammar G'
Output: The SLR parsing table function s*action* and *goto*for G'
Method:
Construct C = {$I_0, I_1, \ldots I_n$}, the collection of sets of LR(0) items for G'.
State*i*is constructed from I_i. The parsing functions for state *I* are determined as follows:
If [A→a·*a*β] is in I_i and goto(I_i,*a*) = I_j, then set*action*[*i*,*a*] to "shift j". Here *a* must be terminal.
If [A→a·] is in I_i , then set*action*[*i*,*a*] to "reduce A→a" for al l*a* in FOLLOW(A).
If [S'→S.] is in I_i, then set *action*[*i*,$] to "accept".

If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).

The *goto* transitions for state *I* are constructed for all non-terminals A using the rule: If *goto*(I_i,A) = I_j, then *goto*[i,A] =*j*.
All entries not defined by rules (2) and (3) are made "error"
The initial state of the parser is the one constructed from the set of items containing [S'→.S].

Parul® University

LR-Parser:

Example for SLR parsing:
Construct SLR parsing for the following grammar :
G : E → E + T | T
T → T * F | F F → (E) | id

The given grammar is :
G : E → E + T ------ (1)
E →T ------ (2)
T → T * F ------ (3)
T → F ------ (4)
F → (E) ------ (5)
F → id ------ (6)

Parul® University

LR-Parser:

Step 1 :Convert given grammar into augmented grammar.

Augmented grammar :

E’ → E

E → E + T

E → T

T → T * F T → F

F → (E)

F → id

Parul® University

Augmented Grammar &LR(0) Items

Let's consider the following grammar

S -> AA

A -> aA | b

The augmented grammar for the above grammar will be

S' -> S

S -> AA

A -> aA | b

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

S -> ABC -> Given production

LR(0) Item of given production

S -> .ABC

S -> A.BC

S -> AB.C

S -> ABC.

Parul® University

Closure() & Goto()

Closure():
If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the two rules:
1. Initially every item in I is added to closure(I).
2. If A -> α.Bβ is in closure(I) and B -> γ is a production then add the item B -> .γ to I, If it is not already there. We apply this rule until no more items can be added to closure(I).

Goto () :
Goto(I, X) = 1. Add I by moving dot after X.
2. Apply closure to first step.

Parul® University

Example of Closure

S' → S
S → AA
A → aA/b

Closure(S'→S) = S'→.S → non terminal. So add production of S
S→.AA → non terminal after dot. So add production of A
A→.aA/ .b

Closure(S→A.A) = S→A.A
A→.aA/ .b

Closure(A→aA) = A→a.A

Image source : Google

Parul® University

DIGITAL LEARNING CONTENT

Example of GO to()

S' → S

S → AA

A → aA/b

Goto(S'→.S,S) = S'→S . □ → nothing is present after dot, no closure

S →.A A → non terminal after dot.So add production of A

A →.aA / .b

Goto(S →.AA,A) = S→A.A → apply closure

A →.aA / .b

Goto(A →aA,a) = A→ a.A → apply closure

A →.aA / .b

Image source : Google

Parul® University

LR(0) Parser

Consider the following grammar

G:

S → AA ----------(1)

A → aA ----------(2)

A-> b ----------(3)

Step: 1- Covert this grammar in Augmented grammar G':

G':

S` → •S

S → •AA

A → •aA

A → •b

Step-2:Find out the Closure() and Goto() for the given grammar

Step-3: Based up closure and Goto() construct the DFA of Augmented production

Parul® University

DFA Of LR(0) Parser

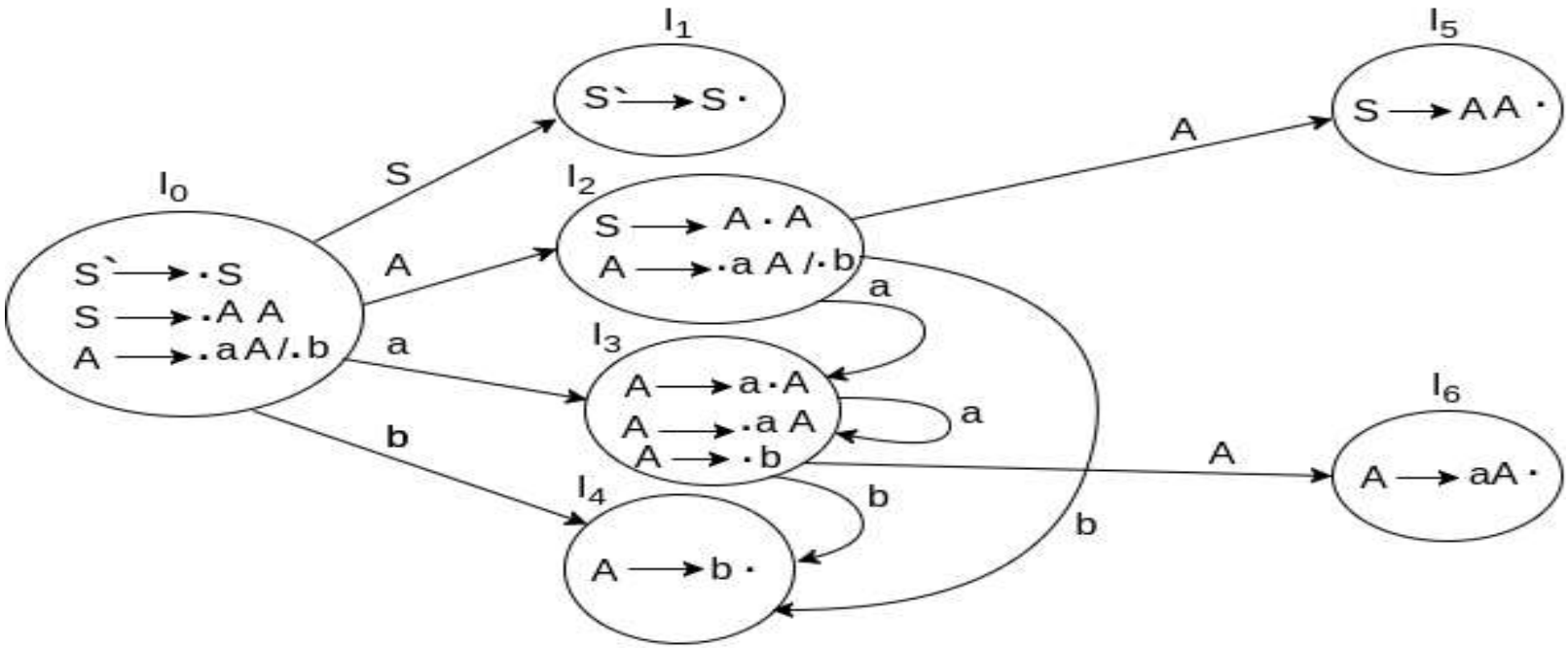


Image source : Google

Parul® University

LR(0) Parsing Table:

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		

Note: LR (0) table contain single entry in every row and column so this grammar will pass LR(0) parser. If any row or column contains multiple entry then the given grammar will not be pass LR(0) parser.

Parul® University

SLR(1) Parser

SLR(1) Parser:

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.

Example of SLR (1) Parsing :

Consider the following grammar G:

$S \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow id$

Parul® University

SLR(1) Parser

Step-1: Convert this Grammar in to Augmented Grammar G'

S` → •E
E → •E + T
E → •T
T → •T * F
T → •F
F → •id

Step-2:Find out the Closure() and Goto() for the given grammar
Step-3: Based up closure and Goto() construct the DFA of Augmented production

Parul®
University

DFA for SLR(1)

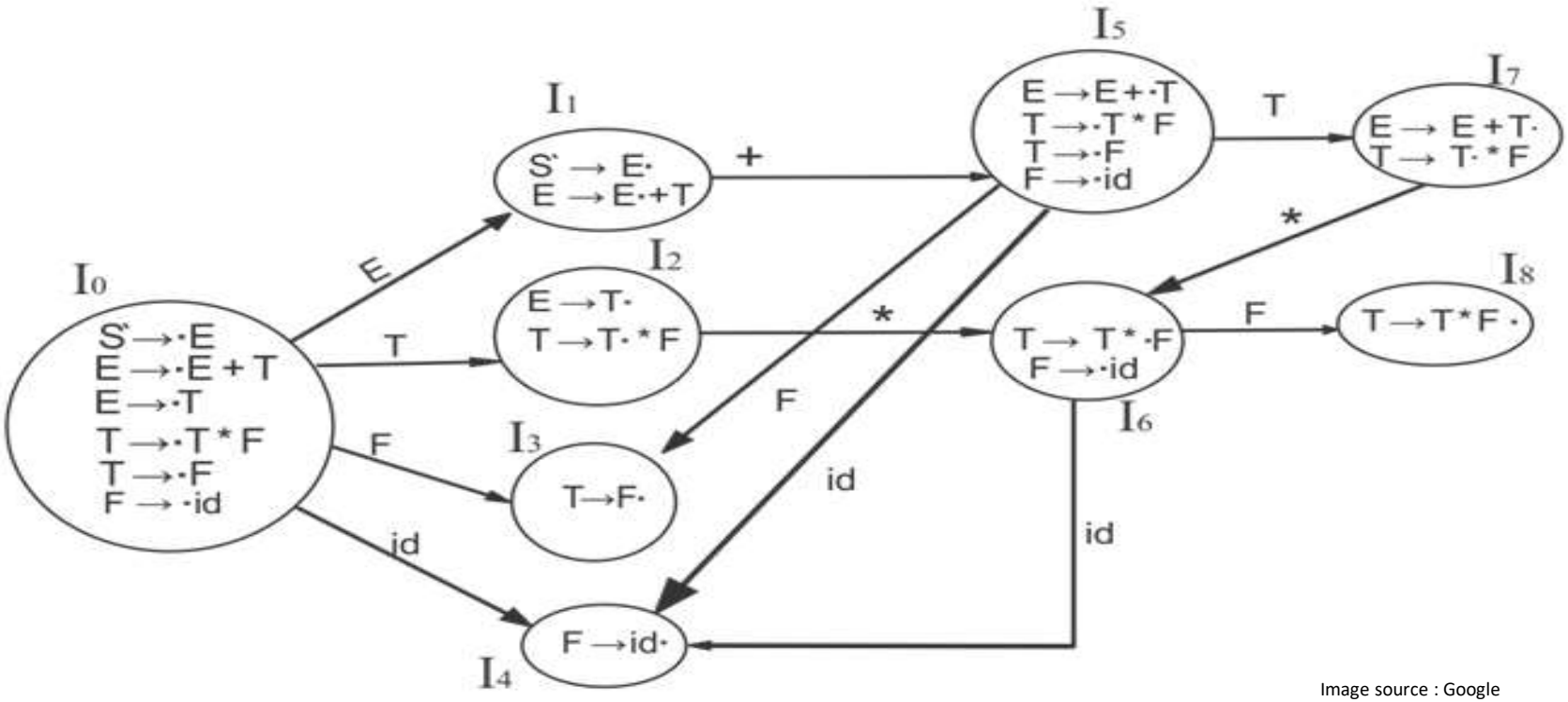


Image source : Google

Parul® University

SLR(1) Parsing Table:

1.If a state is going to some other state on a terminal then it correspond to a shift move.
2. If a state is going to some other state on a variable then it correspond to go to move.
3. Final entry of reduction can be done based on the FOLLOW() of that variable.

States	Action				Go to		
	id	+	*	$	E	T	F
I_0	S_4				1	2	3
I_1		S_5		Accept			
I_2		R_2	S_6	R2			
I_3		R_4	R_4	R4			
I_4		R_5	R5	R5			
I_5	S4					7	3
I_6	S4						8
I_7		R1	S6	R1			
I_8		R3	R3	R3			

Parul® University

CLR(1)/LR(1) Parser

CLR(1) /LR(1) Parsing:

CLR refers to canonical look ahead. CLR parsing use the canonical collection of LR (1) items to build the CLR (1) parsing table. CLR (1) parsing table produces the more number of states as compare to the SLR (1) parsing.

LR(1) Item= LR(0) Item + Look ahead Symbol

Parul® University

Example of CLR(1) Parser

Consider the following grammar to construct to LR(1) item

G:

S → AA

A → aA

A → b

Augmented grammar with LR(1) Item

S` → •S, $

S → •AA, $

A → •aA, a/b

A → •b, a/b

To construct the CLR(1) Parser

Step-1: construct Augmented grammar with LR(1) item

Step-2:Find out the Closure() and Goto() for the given grammar with Look ahead symbol

Parul®
University

DFA for CLR(1)

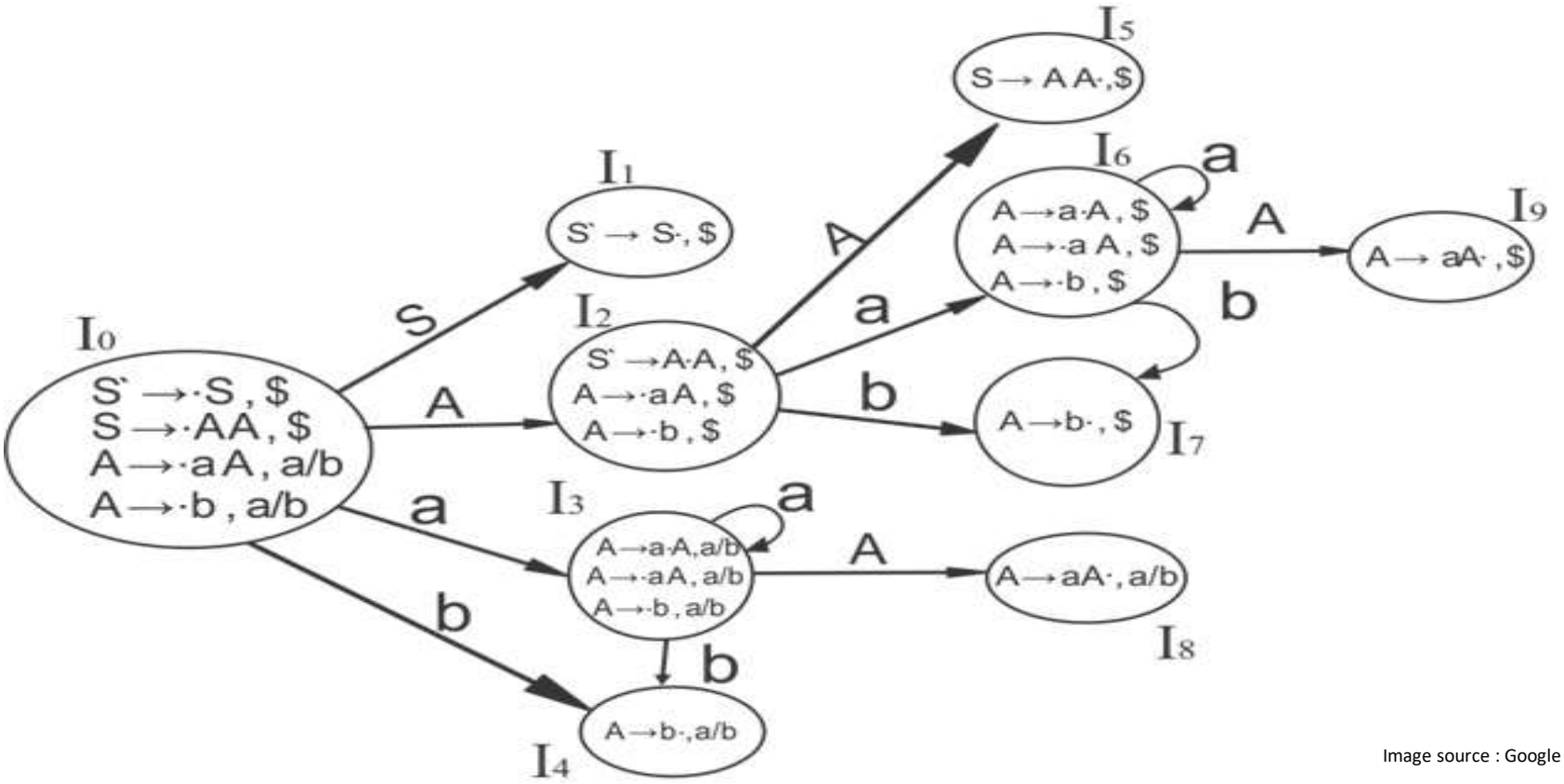


Image source : Google

Parul® University

CLR(1) parsing Table

Construction of CLR(1) Parsing Table:

1. If a state is going to some other state on a terminal then it correspond to a shift move.
2. If a state is going to some other state on a variable then it correspond to go to move.
3. Final entry of reduction can be done based on the look ahead symbol of that the production

Parul® University

CLR(1) Parsing Table

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

Parul® University

Example of LALR(1)

Consider the following grammar to construct to LR(1) item

G:

S → AA ,A → aA ,A → b

Augmented grammar with LR(1) Item

S` → •S, $

S → •AA, $

A → •aA, a/b

A → •b, a/b

To construct the LALR(1) Parser

Step-1: construct Augmented grammar with LR(1) item

Step-2:Find out the Closure() and Goto() for the given grammar with Look ahead symbol

Step-3: Based up closure and Goto() construct the DFA of Augmented production

Step-4: If two state of DAF is having same production but different look ahead symbol then merge these two step in single state.

Parul® University

LALR(1) Parsing

LALR(1) Parsing

LALR refers to the look ahead LR. To construct the LALR (1) parsing table, we use the canonical collection of LR (1) items.

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

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

Parul® University

DFA for LALR(1)

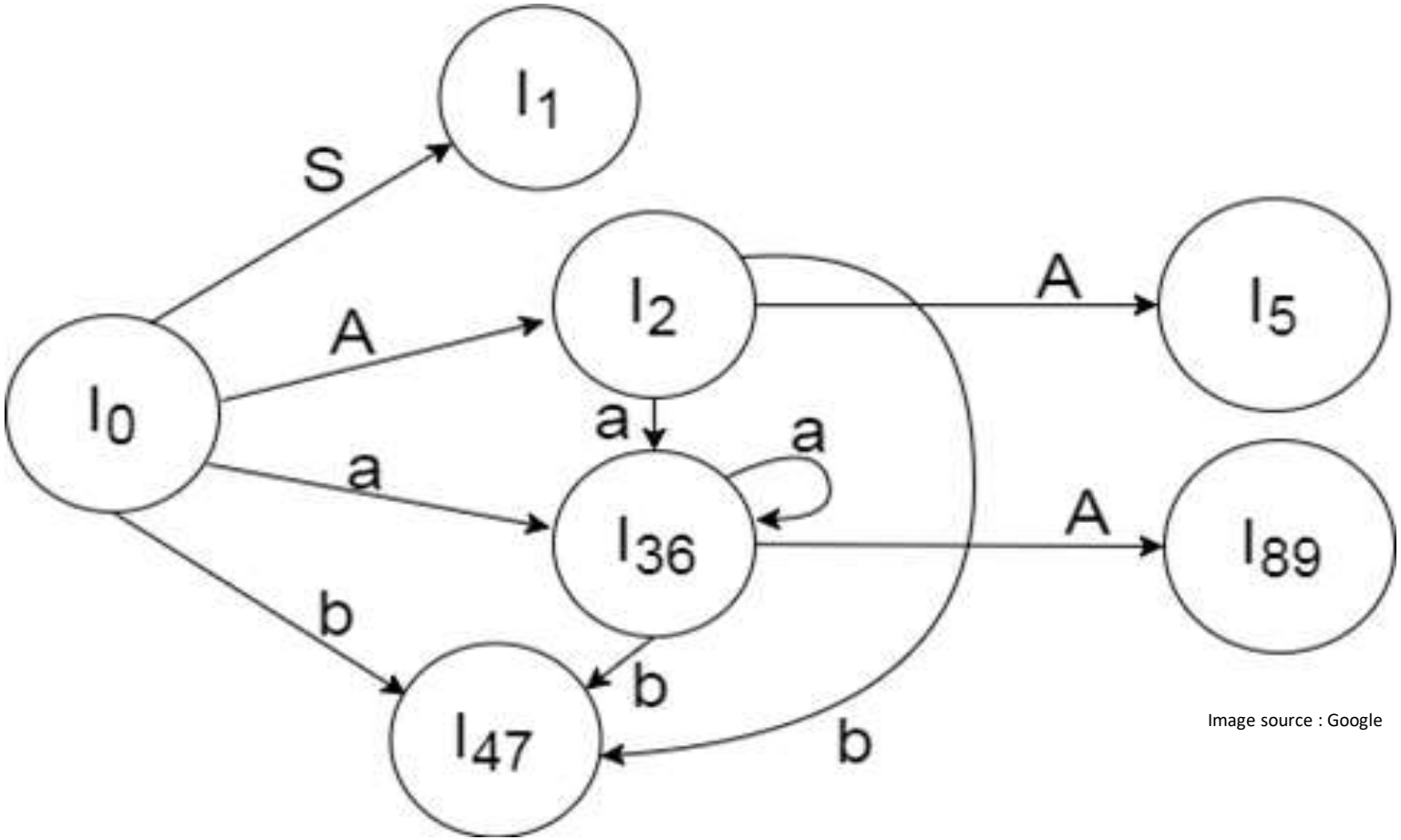


Image source : Google

Parul® University

LALR(1) Parsing Table

LALR(1) Parsing Table:

1. If a state is going to some other state on a terminal then it correspond to a shift move.
2. If a state is going to some other state on a variable then it correspond to go to move.
3. Final entry of reduction can be done based on the look ahead symbol of that the production

Parul® University

LALR(1) Parsing Table

States	a	b	$	S	A
I_0	S_{36}	S_{47}		12	
I_1		accept			
I_2	S_{36}	S_{47}			5
I_{36}	$S_{36}S_{47}$				89
I_{47}	R_3R_3	R_3			
I_5			R_1		
I_{89}	R_2	R_2	R_2		

Image source : Google

Parul®
University

Operator Precedence parsing:

- Operator precedence grammar is kinds of shift reduce parsing method. It is applied to a small class of operator grammars.
- A grammar is said to be operator precedence grammar if it has two properties:
- No R.H.S. of any production has a ∈.
- No two non-terminals are adjacent.
- Operator precedence can only established between the terminals of the grammar. It ignores the non-terminal.

Parul® University

Operator Precedence parsing

There are three operator precedence relations:

a ⋗ b means that terminal "a" has the higher precedence than terminal "b".

a ⋖ b means that terminal "a" has the lower precedence than terminal "b".

a ≐ b means that the terminal "a" and "b" both have same precedence.

Example of Operator precedence:

Grammar

E → E+T/T

T → T*F/F

F → id

Parul® University

Operator Precedence Table

	E	T	F	id	+	*	$
E	X	X	X	X	≐	X	⋗
T	X	X	X	X	⋗	≐	⋗
F	X	X	X	X	⋗	⋗	⋗
id	X	X	X	X	⋗	⋗	⋗
+	X	≐	⋖	⋖	X	X	X
*	X	X	≐	⋖	X	X	X
$	⋖	⋖	⋖	⋖	X	X	X

Image source : Google

Parul® University

For the given string W= id + id * id check whether this parser will parsed or not

$ ⋖ id1 ⋗ + id2 * id3 $

$ ⋖ F ⋗ + id2 * id3 $

$ ⋖ T ⋗ + id2 * id3 $

$ ⋖ E ≐ + ⋖ id2 ⋗ * id3 $

$ ⋖ E ≐ + ⋖ F ⋗ * id3 $

$ ⋖ E ≐ + ⋖ T ≐ * ⋖ id3 ⋗ $

$ ⋖ E ≐ + ⋖ T ≐ * ≐ F ⋗ $

$ ⋖ E ≐ + ≐ T ⋗ $

$ ⋖ E ≐ + ≐ T ⋗ $

$ ⋖ E ⋗ $

Accept.

Image source : Google

DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversity.ac.in