Compiler - GATE

* Symbol table is updated during lexical analysis, syntax analysis and semantic analysis

* Top down parser - Leftmost Derivation

  Classification

  Predictive Parser {
  1. LL(1)
  2. Recursive Descent Parsing
  }
  3. Non Recursive Predictive Parsing
  4. Backtracking

* Bottom up parser - Reverse of Rightmost Derivation

  also called Shift Reduce (SR) Parser

  Classification

  1. Operator Precedence Parser - no 2 consecutive variables
  2. (Shift Reduce Parsers) LR(K) parsing → LR(0), SLR(1), LALR(1), CLR(1)

* $LR(0) \subseteq SLR(1) \subseteq LALR(1) \subseteq CLR(1)$

  $LL(1) \subseteq LALR(1)$

  CLR(1) ↓ also called **LR(1)**

	High level language ↓	
DFA minimization, Key word recognition, Regular Expression, Finite Automata (Front End)	Lexical Analyzer	- INPUT - Character Stream; - read the program character by character
PDA	Syntax Analyzer	- INPUT - Token stream
Type checking	Semantic Analyzer	- INPUT - Syntax tree
	Intermediate code generation	
Dataflow Analysis, DAG	Code optimizer	
Register Allocation (Graph coloring), Syntax Tree (Back end)	Target Code Generation	- INPUT - Intermediate Representation
	↓ Assembly Code / Machine code	

Expression evaluation	Post order traversal	Activation record	Runtime environment
Abelian Group	Push Down Automata	Leftmost derivation	Top Down Parsing
Type checking	Syntax Directed Translation		Symbol Table - Info about variables and their attributes
Production Tree	Parsing		

PTO

* LL(1) Parser / Predictive Parser

- most widely used Top down parser
- parses the given input string with the help of LL(1) parsing table
- two function First( ) and Follow() are used to construct LL(1) parsing table.

FIRST (X) for a grammar symbol X, is the set of terminals that begin the strings derivable from X.

Rules to compute FIRST set -

1. If x is a terminal, then FIRST (x) = $\{ 'x' \}$
2. If $x \rightarrow \varepsilon$ is a production rule, then add ε to FIRST (x).
3. If $X \rightarrow Y_1 Y_2 Y_3 \ldots Y_n$ is a production
   1. FIRST (X) = FIRST (Y_1)
   2. If FIRST (Y_1) contains ε, then
      FIRST (X) = $\{$ FIRST $(Y_1) - \varepsilon \} \cup \{$ FIRST $(Y_2) \}$
   3. If FIRST (Y_i) contains ε for all $i = 1$ to n, then add ε to FIRST(X)

FOLLOW(X) to be the set of terminals that can appear immediately to the right of Non-Terminal X in some sentential form. any string derivable from start symbol

Rules to compute FOLLOW set -

1. If B is start symbol of G Grammar
   FOLLOW(B) = $\{ \$ \}$

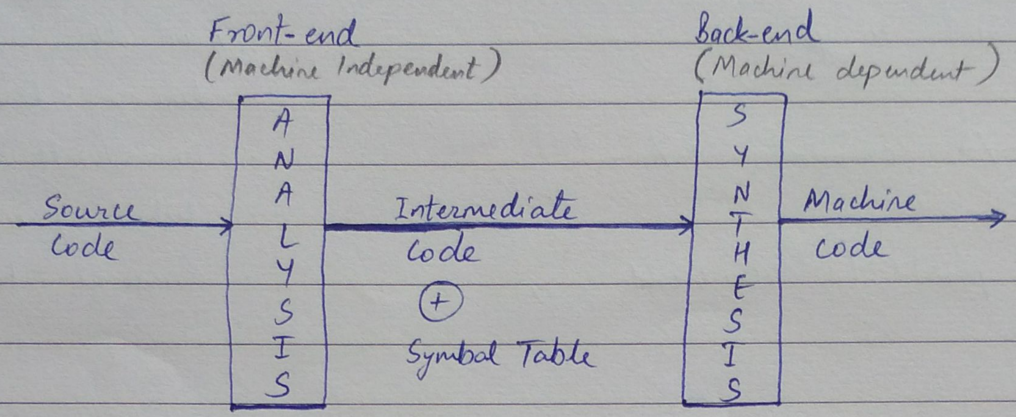
• Follow(A) ⊆ Follow(B)
• However reverse may **not** be true

2. If $A \rightarrow pB$ is a production, then everything in FOLLOW (A) is in FOLLOW (B)
3. If $A \rightarrow pBq$ is a production, where p, B and q are any grammar symbols, then everything in FIRST(q) except ε is in FOLLOW(B)
4. If $A \rightarrow pBq$ is a production and FIRST (q) contains ε, then FOLLOW(B) contains $\{$ FIRST$(q) - \varepsilon \} \cup$ FOLLOW(A)

ε never comes in FOLLOW set

Compiler (Unit 1)

Compiler design Architecture

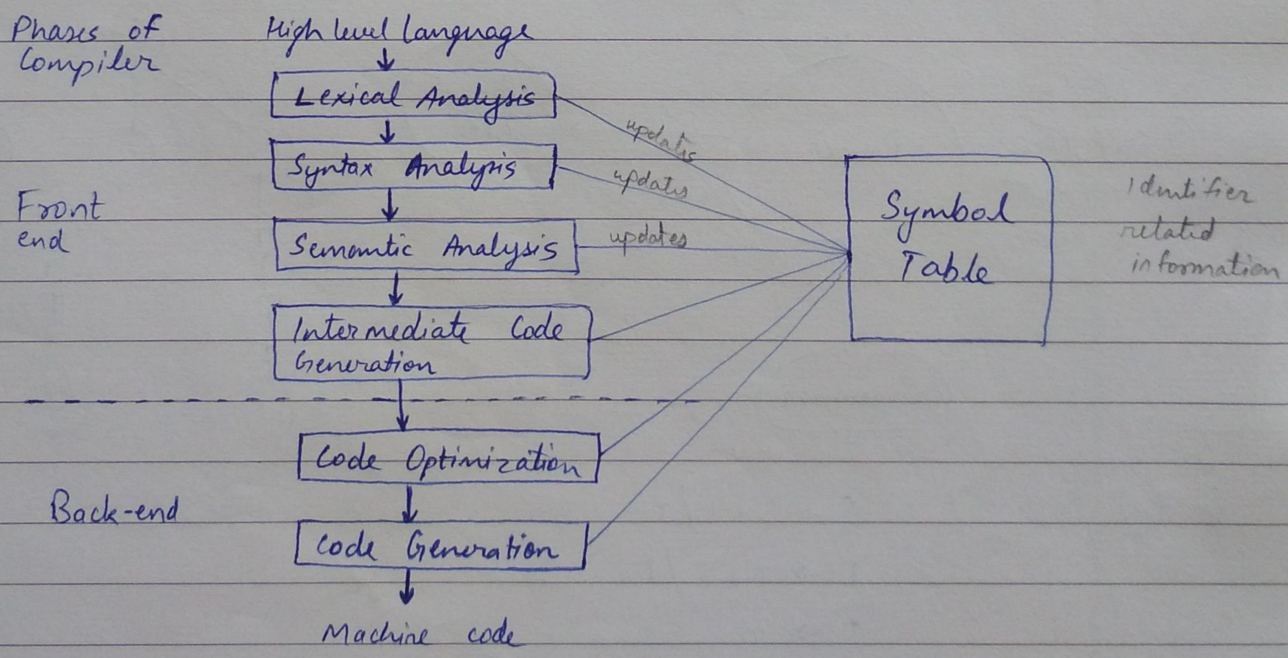


- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Intermediate Code Generation

- Code Optimization
- Code Generator

Compiler – without changing the meaning of the high level language, it converts the source code into machine code

Phases of Compiler



Lexical Analysis

5 tokens

int | x | , | y | ; |
1 | 2 | 3 | 4 | 5

~~/* comment */~~

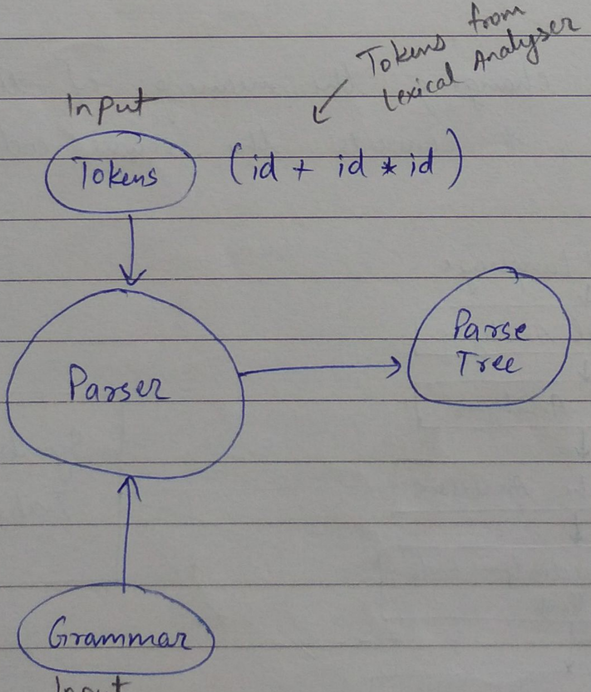
printf | (| "Hello World" |) | ; |
1 | 2 | 3 | 4 | 5

Lexical Analyser -

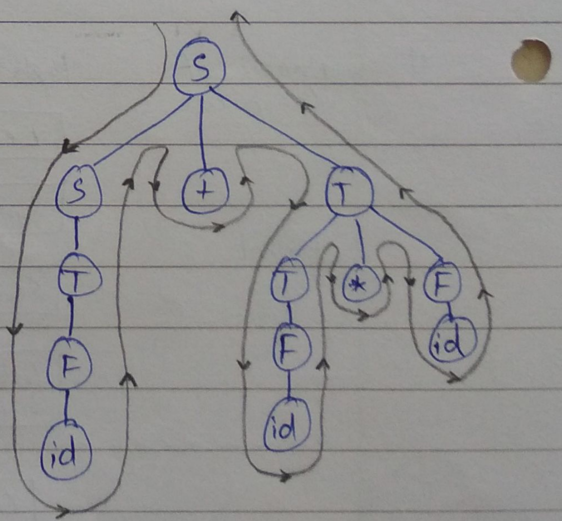
- Read the High Level language (HLL) line by line
- Discards all comments
- White space elimination
- Generate tokens

Syntax Analysis

Syntax Analyser / Parser - evaluation of parse tree must map/match to the input tokens. Evalution is top-down, left-to right



eg - LR, CLR, SLR, LALR
Top Down parser
Botton Up parser



evaluation = id + id * id

Grammar

$G = (V, T, P, S)$

Variables, Terminals, Productions, Symbol

$S \rightarrow S + T / T$

$T \rightarrow T * F / F$

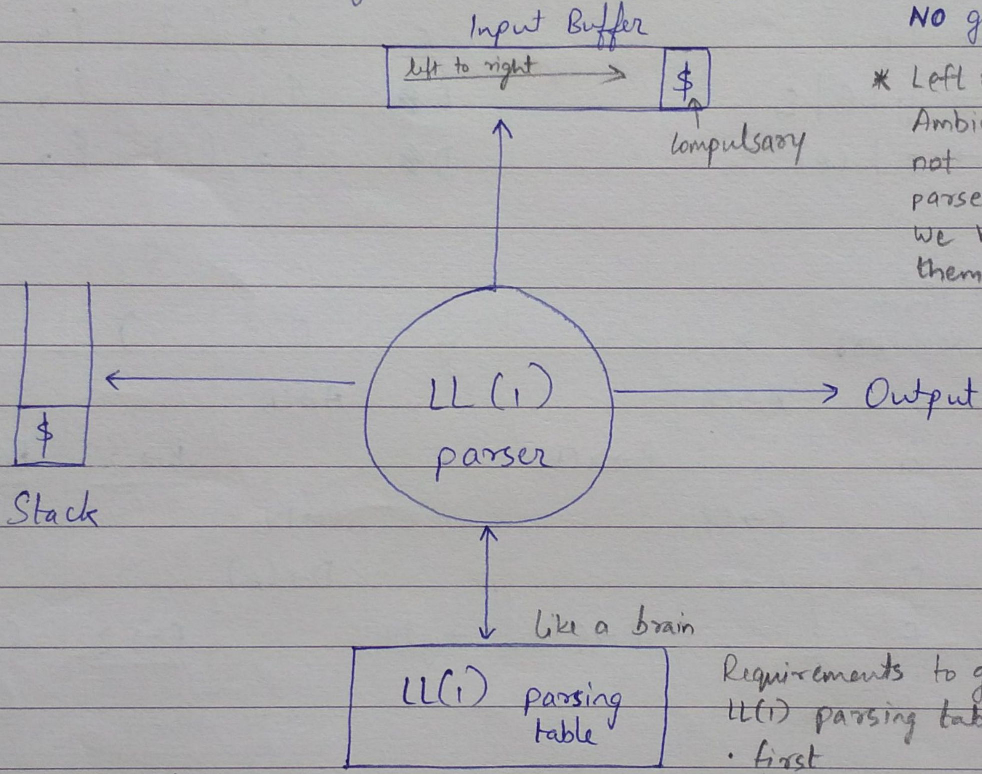
$F \rightarrow id$

(Unit 2)

only 1 symbol look ahead

LL(1) Parser / Predictive Parser

- Top down parser
- Left to Right parsing
- Left most derivation - left most variable is expanded first
- follows LL grammar - subset of CFG, under LL grammar NO grammar is ambiguous



* Left recursion and Ambiguous grammar is not allowed for LL(1) parser.
We have to first remove them and then solve

Requirements to generate LL(1) parsing table
- first
- follow

Always Variable

First (α) and Follow (α) $\xrightarrow{\text{output}}$ set of terminals only

First (α): It is a set of terminal symbols that begin in strings derived by "α"

$A \rightarrow ab \mid de \mid gh$

First (A) = { a, d, g }

Follow (α): It is a set of terminal symbols that appear immediately to the right of "α"
Tip - always try to locate α on the RHS of other productions

$S \rightarrow aAb$

$A \rightarrow c$

Follow of start symbol

Follow (S) = { \$ }

Follow (A) = { b }

For $A \rightarrow pB$, Follow(A) $\subseteq$ Follow(B)

Construction of LL(1) parsing ~~tag~~ table

Production rule	Variables	First	Follow
$A \to CB$	A	{ z, (}	{ $,) }
$B \to xCB \mid \varepsilon$	B	{ x, ε }	{ $,) }
$C \to DE$	C	{ z, (}	{ x,), $ }
$E \to yDE \mid \varepsilon$	E	{ y, ε }	{ x,), $ }
$D \to z \mid (A)$	D	{ z, (}	{ x, y,), $ }

Terminals → / Variables ↓	z	x	y	(	)	$
A	A→CB			A→CB		
B		B→xCB			B→ε	B→ε
C	C→DE			C→DE		
D	D→z			D→(A)		
E		E→ε	E→yDE		E→ε	E→ε

Synchronizing entries - those entries which can be obtained with the help of FOLLOW set. All synchronizing entries are underlined in the above table.

- All productions of "A" will be in the row of "A". Same applies for other variables.
- Each production will go into the column of terminals which are present in the set generated using first(X) where X is the first variable/terminal of that respective production (i.e. $A \to X\alpha_1\alpha_2 \ldots$). Same applies for all production of that particular variables productions, and for all variables as well, eg- for A→CB, row is A & columns = first(C)
- For production like B→ε, first of ε is ε.
  ∴ We use follow(B) to get the column for ~~[illegible]~~ B→ε
  We do the same for E→ε

$A \to X\alpha_1\alpha_2$
↑ row ↑ FIRST(X) columns

How to check if a Grammar is LL(1) or not

- For a Grammar to be LL(1), each cell or LL(1) parsing table should have only one production
- If a cell has two productions in it, then it is ambiguous and NOT a LL(1) grammar.
- Example - the table on the previous page has only one production in each cell. Hence the respective grammar is a LL(1) grammar.
- Example - the following grammar is not LL(1) grammar.

Productions
$X \to Y \mid a$
$Y \to a$

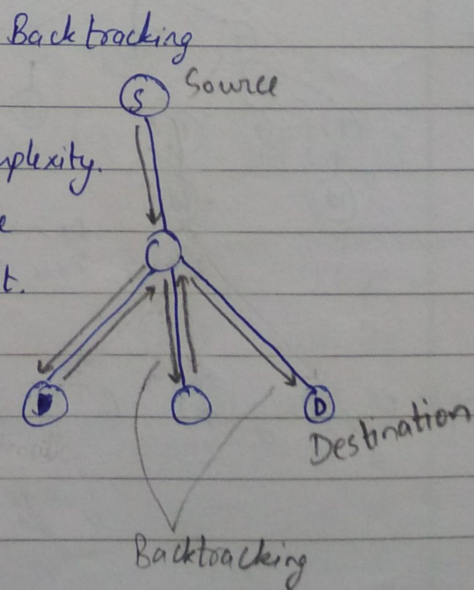
Variable	first	follow
X	{a}	{$}
Y	{a}	{$}

Terminals → / Variables ↓	a	$
X	$X \to Y$ $X \to a$	
Y	$Y \to a$	

As two productions in one cell, this is not a valid LL(1) parse table

Recursive Descent Parser

- Top-down
- Left to right
- Disadvantage - exponential time complexity.
- Uses match(...) function to see if the path choosen is correct or not.



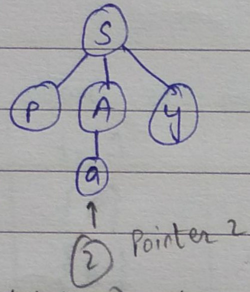
- example -

Grammar:

$S \to pAy$

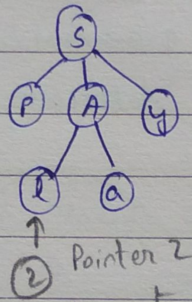
$A \to a \mid la$

Input: play
↑
① pointer 1



② Pointer 2

Match(...) returns false. Hence backtrack



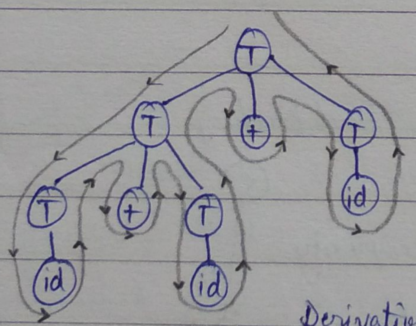
② Pointer 2

match returns true. Hence proceed

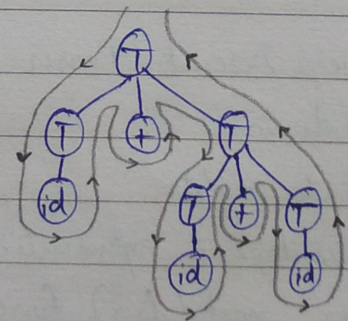
Ambiguous Grammar - when there exists more than 1 derivation tree for a given valid input string.

Production : $T \to T+T \mid id$

Input : $id + id + id$



Derivation Tree 1



Derivation Tree 2

Unambiguous Grammar - there exists a unique derivation tree for every given valid input string.

Operator Precedence Parser

- Bottom-up parser
- Reduction operation is performed on the input string to get the starting symbol
- Restriction on Operator Precedence Grammar -
  (1) ε should **not** be on RHS of the productions
  (2) 2 non-terminals (variables) can NOT be adjacent to each other

$A \rightarrow AB$
$A \rightarrow BB$ } Not valid

{ $S \rightarrow SAS \mid a$
{ $A \rightarrow bSb \mid b$

↓ fix the problem

$T \rightarrow T+T \mid T*T \mid id$ } Valid

{ $S \rightarrow SbSbS \mid a \mid SbS$
{ $A \rightarrow bSb \mid b$

Operator Precedence Relation

$T \rightarrow T+T \mid T*T \mid id$

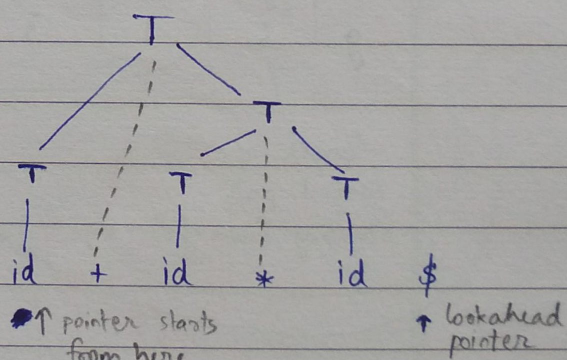
Precedence : id ·> * ·> + ·> $

Terminals = { id, +, *, $ }

Disadvantage of this relation - its size is n^2

use "<·" if right associative

	id	+	*	$
id	-	·>	·>	·>
left associative +	<·	·>	<·	·>
left associative *	<·	·>	·>	·>
$	<·	<·	<·	- parse tree successfully built



Stack | $ | id | + | id | * | id |

New Stack	Comparisons	
\$ | id	$ <· id	push & increment
\$	id ·> +	pop & reduce
\$ | +	$ <· +	push & increment
	+ <· id	
\$ | + | id		push & increment
\$ | +	id ·> *	pop & reduce
\$ | + | *	+ <· *	push & increment
\$ | + | * | id	* <· id	push & increment
\$	id ·> $, * ·> $, + ·> $	pop pop pop

if (precedence of stack symbol <· precedence of operator pointed by lookahead pointer) { push } increment pointer

if (precedence of stack symbol ·> operator pointed by lookahead pointer) { pop } reduce the popped element

Successfully built parse tree ⇐ stack [$] $ pointer ←

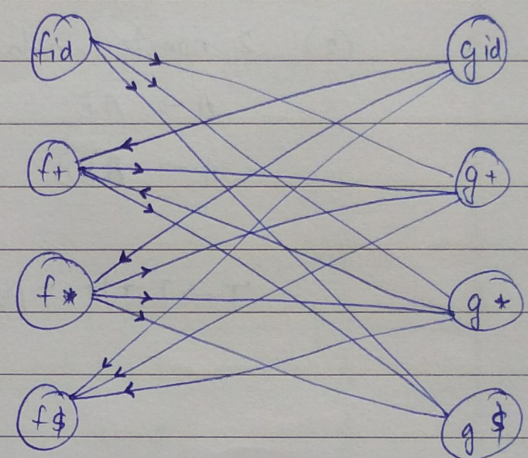
Operator Function Table

- There should not be any cycle in the graph generated. If there is any cycle, then the function table can not be generated for that ~~gra~~ graph
- New size = $2n$

space = n^2

g

f	id	+	*	$
id	-	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	-



space = $2n$

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

length of longest path starting from g(id)

fid → g* → f+ → g+ → f$

gid → f* → g* → f+ → g+ → f$

Compare

fid, g* → 4 > 3

f+, gid → 2 < 5

Shift Reduce Parser

- Buttom up parser
- Actions
  - Shift - push current input symbol to stack
  - Reduce - replace symbols by non-terminals (variables) i.e replace RHS of a production by LHS

- Example-

* Productions-

$E \to E + E$

$E \to a$

$E \to b$

* Input string-

$a + b$

Stack	Input String	Action
\$	a+b\$	Shift a
\$a	+b\$	Reduce $E \to a$
\$E	+b\$	Shift +
\$E+	b\$	Shift b
\$E+b	\$	Reduce $E \to b$
\$E+E	\$	Reduce $E \to E+E$
\$E	\$	Accept

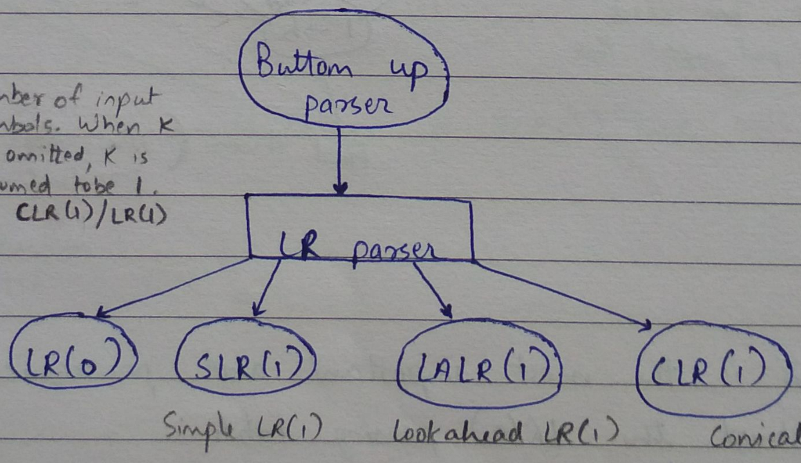
LR parser

LR(K)

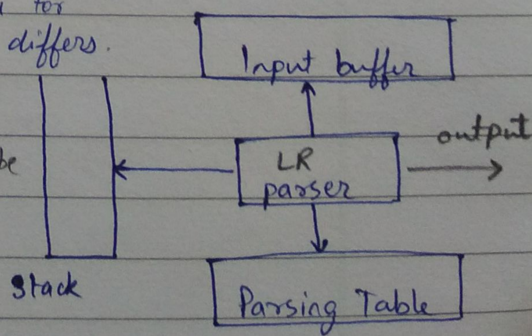
Left to right scanning

Rightmost derivation in reverse

number of input symbols. When K is omitted, K is assumed to be 1. i.e CLR(1)/LR(1)



Note - Parser, stack & Input buffer are same for all parsers
- Only the parsing table construction differs.



→ LR(0) items (LR(0), SLR(1))

Dot means that the input has been read till that point.

$E \to .aA$

Requirement - 'dot "." should be present on RHS

→ LR(1) items (LALR(1), CLR(1))

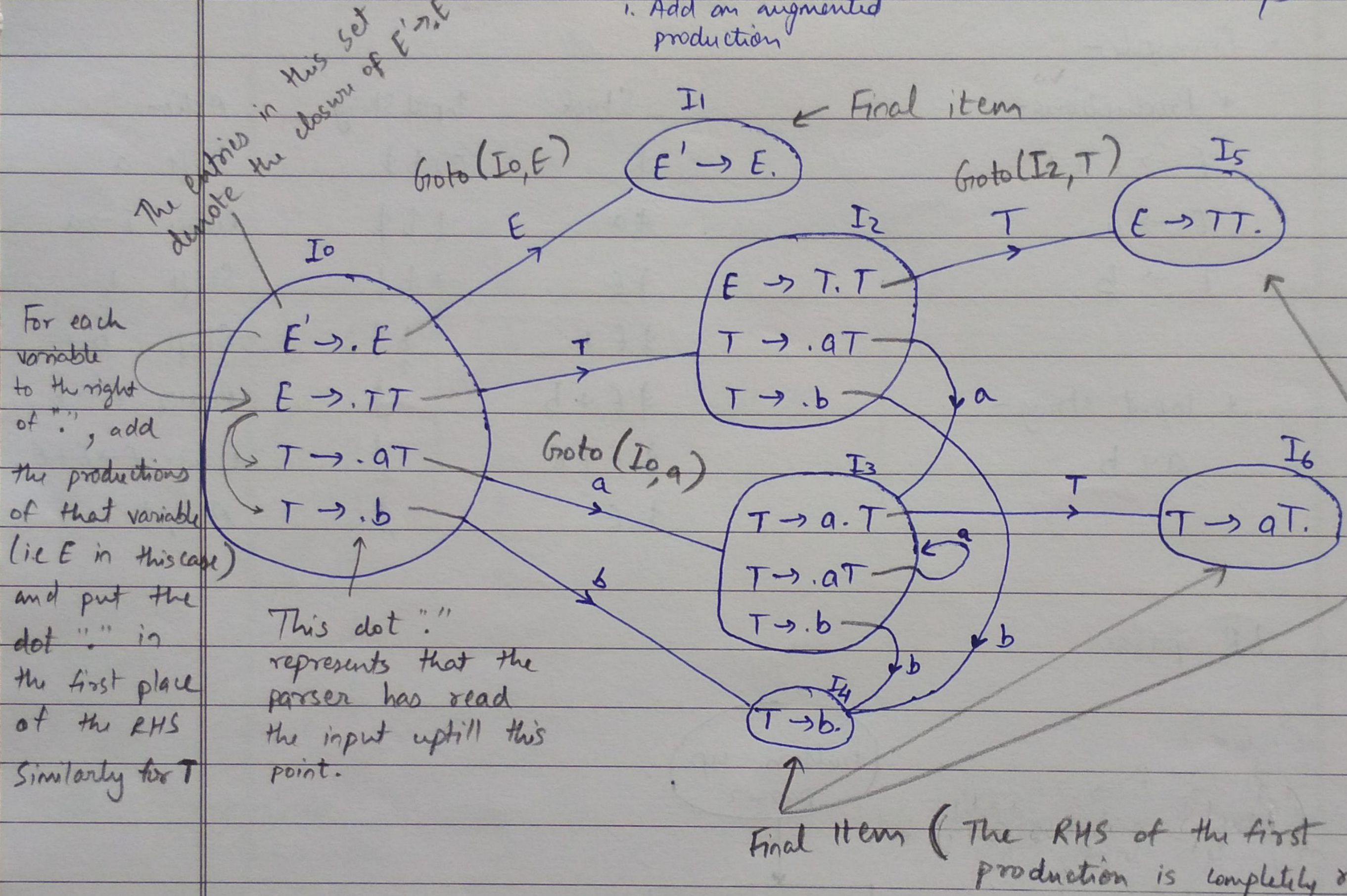
$\underbrace{E \to .aA}_{\text{Same as LR(0)}}$, $\underbrace{a/b}_{\text{Lookahead}}$

LR(0) item

$E \rightarrow TT$
$T \rightarrow aT / b$

Convert normal grammar to ~~augmented~~ augmented grammar. Steps –
1. Add an augmented production

$E' \rightarrow E$ ← Augmented Production
$E \rightarrow TT$
$T \rightarrow aT/b$



This is the most important and prerequisite structure to construct the LR(0) parsing table

Construction of LR(0) parsing table

Refer the diagram and grammar on pg 10

$E' \to E$

$E \to TT$ ①

$T \to aT \,/\, b$
② ③

Necessary-
We number the productions for use in the table when denoting the reduce operation

The state in which the augmented production becomes the final item, that is when we use "accept"

Perform shift on seeing "a" when in I_0, and move to I_3

On seeing E when in I_0, move to I_1

Represents states $I_0, I_1, \dots$

This is augmented relation

Final Items

	Action(terminals)			Goto(variables)	
	a	b	\$	E	T
0	S_3	S_4		1	2
1			accept		
2	S_3	S_4			5
3	S_3	S_4			6
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

Note - we fill all columns under Action with r_n where n is the production number in the augmented grammar

only this will change in SLR(1) table

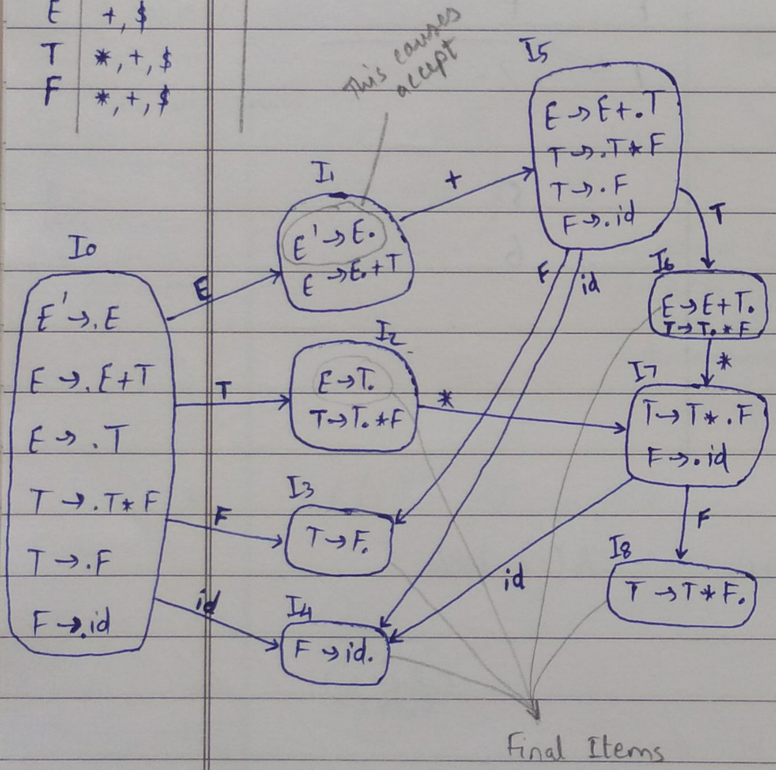
Construction of SLR(1) parsing table

Production:

$E \to E + T$		$E' \to .E$
$E \to T$	Convert to augmented grammar **and** make them LR(0) items by adding "." to the leftmost side of RHS	$E \to .E + T$ ①
$T \to T * F$		$E \to .T$ ②
$T \to F$		$T \to .T * F$ ③
$F \to id$		$T \to .F$ ④
		$F \to .id$ ⑤

Necessary - we number the productions / LR(0) items for use in the table while writing the reduce operation.

	Follow(X)
E	+, \$
T	*, +, \$
F	*, +, \$



	Action				Goto		
	id	+	*	\$	E	T	F
0	S_4				1	2	3
1		S_5		accept			
2		r_2	S_7	r_2			
3		r_4	r_4	r_4			
4		r_5	r_5	r_5			
5	S_4					6	3
6		r_1	S_7	r_1			
7	S_4						8
8		r_3	r_3	r_3			

Steps

1. Find all the states where final item has come. Except the state with augmented production $E' \to E.$
2. For example, consider I_4 item/state in the transition diagram → row 4

$F \to id$ — this is 5th production in the converted grammar. Hence write r_5

$\text{Follow}(F) = \{ *, +, \$ \}$

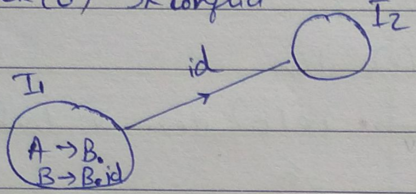
Write r_5 under these terminals for the 4th row

SR conflict and RR conflict

(SR = Shift Reduce, RR = Reduce Reduce)

* LR(0) - SR conflict

Assume $A \to B$ is 3^{rd} production in the augmented grammar.



SR conflict

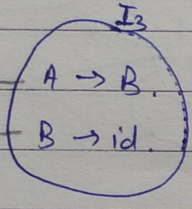
	Action				Goto
	id	+	*	$ ...	......
1	s_2/r_3	r_3	r_3	r_3 ...	.....
					.....

(S_2 Because I_1 has a transition to I_2 for "id")

(r_3 Because I_1 has a final item)

* LR(0) - RR conflict

Assume - $A \to B$ as 2^{nd} production & $B \to id$ as 3^{rd} production in the augmented grammar



RR conflict

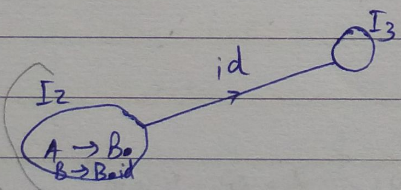
	Action			Goto
	t_1	t_2	t_3	.....
3	r_2/r_3	r_2/r_3	r_2/r_3	.....

r_2 because I_3 has $A \to B.$ as final state which is 2^{nd} production in the augmented grammar

r_3 because I_3 has $B \to id.$ as final state which is 3^{rd} production in the augmented grammar

* SLR(1) - SR conflict

Assume $A \to B$ is 3^{rd} production in the augmented grammar & Follow(A) = {id}



SR conflict

	Action				Goto
	id	+	*	$	.....
2	s_3/r_3				.....

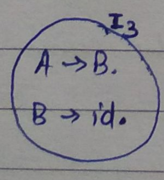
because $A \to B.$ is final item

* SLR(1) - RR conflict

Assume $A \to B.$ as 2^{nd} production & $B \to id$ as 3^{rd} production in the augmented grammar and

Follow(A) = $\{t_1, t_2\}$

Follow(B) = $\{t_2, t_3\}$



	Action			Goto
	t_1	t_2	t_3	.....
3	r_2	r_2/r_3	r_3	.....

RR conflict

CLR(1) / LR(1) parsing table - more states as compared to LR(0) and SLR(1)

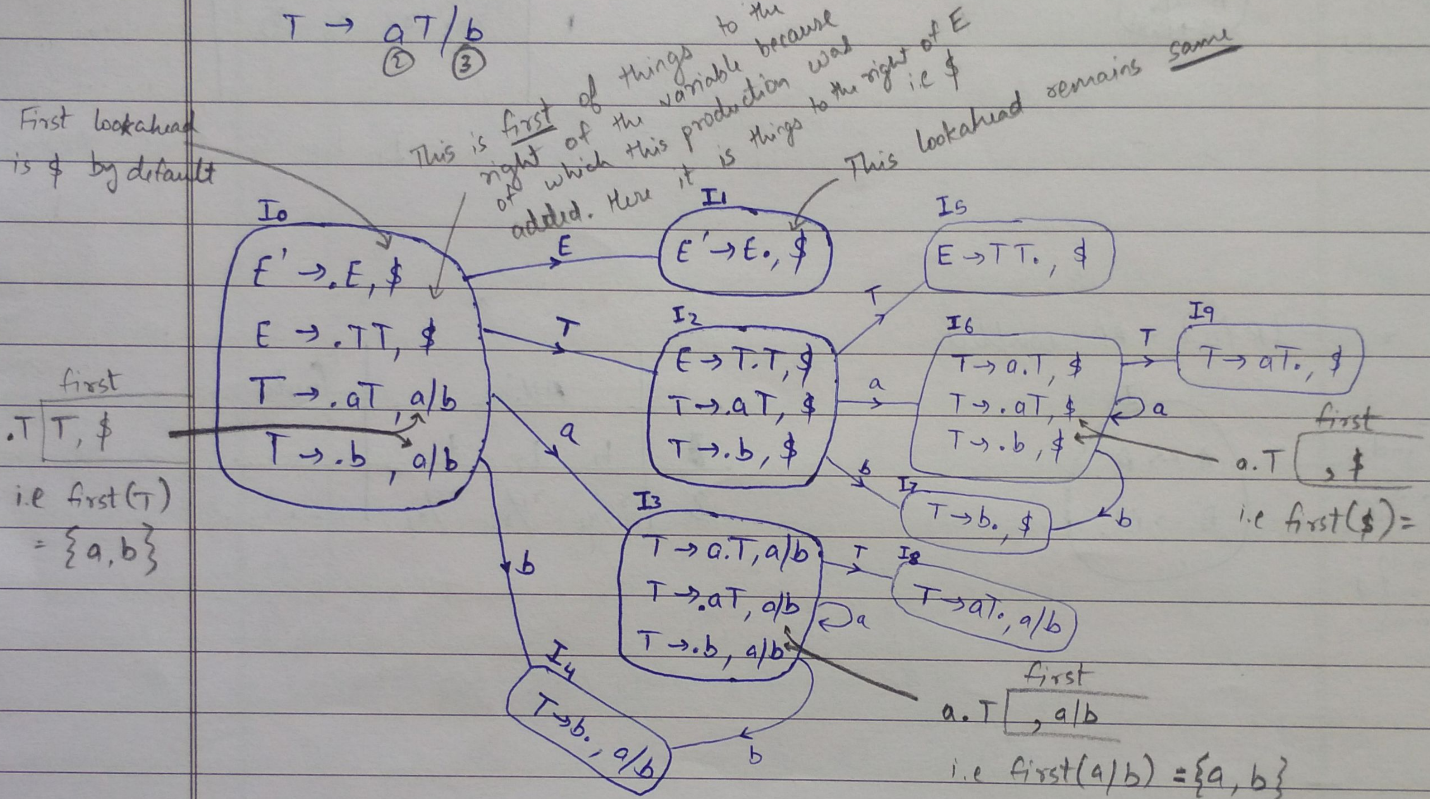
$E' \to E$

$E \to TT$ (1)

$T \to aT / b$ (2) (3)

LR(1) item

↳ LR(0) item + lookahead



Steps

1 Write the first production $E' \to .E, \$$, and add other production if there is a variable to the right of ".", which is same as LR(0) + SLR(1)

2 Lookahead remains the same after transition from one state to another, for the first production only. i.e only the dot shifts towards the right. The ~~lookahed look~~ lookahead may change for production generated due to first production of the same state.

The format of the parsing table remains the same.

Same as LR(0), SLR(1)

It is 3 because on "a", I_0 has transition to I_3

I_4
$T \to b., a/b$
Under a and b we write r_3 because $T \to b$ is 3rd production in the augmented grammar

	Action			Goto	
	a	b	$	E	T
0	S_3	S_4		1	2
1			accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

Note - filling of shift operation and values in Goto column is same as LR(0) and SLR(1) parsing table

like SLR(1)

Instead of using follow(X)$_A$, we use the lookahead to decide the columns.

LALR(1) parsing table

- We use the same example as CLR(1) (or LR(1))
- Take those states whose base content (i.e LR(0) items) is same but lookahead is different, and merge them to generate new states. Replace the old state numbers under Goto and shift operation with the new state numbers.
- Do NOT change numbers with reduce as they are production number from the augmented grammar.
- Merge the rows having same row number and take a union of their entries.

I_3
$T \to a.T, a/b$
$T \to .aT, a/b$
$T \to .b, a/b$

I_6
$T \to a.T, \$$
$T \to .aT, \$$
$T \to .b, \$$

I_{36}
$T \to a.T, a/b/\$$
$T \to .aT, a/b/\$$
$T \to .b, a/b/\$$

I_4
$T \to b., a/b$

I_7
$T \to b., \$$

I_{47}
$T \to b., a/b/\$$

I_8
$T \to aT., a/b$

I_9
$T \to aT., \$$

I_{89}
$T \to aT., a/b/\$$

	Action			Goto	
	a	b	$	E	T
0	S_{36}	S_{47}		1	2
1			accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

merge

	a	b	$
row 89	r_2	r_2	
row 89			r_2

SR conflict and RR conflict with LR(1) items

LR(1) item

SR conflict

RR conflict

I_0

$X \to \alpha A . b \; , \; c/d$

$Y \to a . \; , \; b$

$I_0 \xrightarrow{b} I_2$

Let $Y \to a$ be numbered 2 in augmented grammar

	Action				Goto
	b	c	d	\$	....
0	s_2/r_2				...

conflict

because of transition from I_0 to I_2

because I_0 has final item which is numbered 2 in augmented production and lookahead is b

I_0

$X \to B. \; , \; c/d$

$Y \to A. \; , \; c/d$

Let $X \to B$ and $Y \to A$ be 1st and 2nd production in the augmented grammar

	Action			Goto
	c	d	\$	...
0	r_1/r_2	r_1/r_2		...

conflict

(Unit 3)

Input - Parse Tree → Semantic Analysis → Output Semantically Verified Parse Tree

Syntax Directed Translation (SDT)

Grammar + Semantic Rule = SDT

Example -

These are semantic rules associated with the productions. They are executed during the REDUCTION phase.

$E \rightarrow E + T$ $\{ E.val = E.val + T.val \}$

$\quad | \; T$ $\{ E.val = T.val \}$

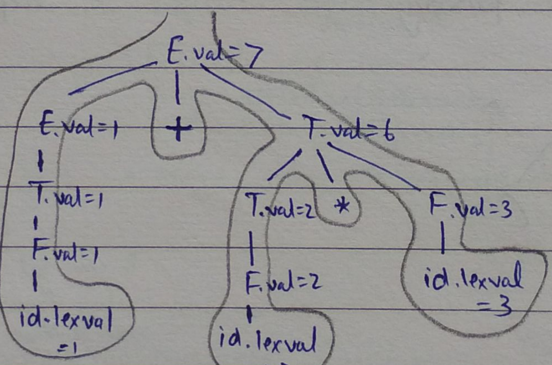
$T \rightarrow T * F$ $\{ T.val = T.val * F.val \}$

$\quad | \; F$ $\{ T.val = F.val \}$

$F \rightarrow id$ $\{ F.val = id.lexval \}$

"val" is an attribute associated with Non terminals (variables)

Lets solve for → 1 + 2 * 3



Synthesized attribute

- It is an attribute of a non-terminal (ie variable) which is at the LHS of the production rule.

$A \rightarrow BCD$ $\{ A.val = B.val, \; A.val = C.val, \; A.val = D.val \}$

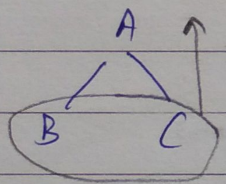
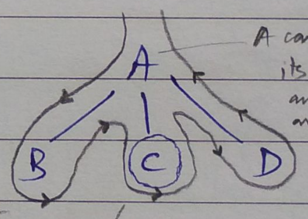
Attribute of A is synthesized attribute as it can get its value from its children B, C, D

Inherited attribute

- It is an attribute of a non-terminal (i.e variable) which is at the RHS of the production rule.

$A \rightarrow BCD$ $\{ C.val = A.val, \; C.val = B.val, \; C.val = D.val \}$

Can take its values from its siblings or its parent
Same goes for B and D.

# S-attributed SDT	L-attributed SDT
- Synthesized attributes are used (i.e parent can take its value from its children only)	- Both Synthesized attribute and inherited attribute (left side only) (for inherited attributes only) eg - A → B C D C can take its value from its parent or left siblings only ✓ C.val = B.val ✗ C.val = D.val
- Semantic actions can only be at the rightmost position A → B C { } B → C { }	- Semantic actions can be ~~any where~~ anywhere on the RHS A → B C D { } B → C { } D C → { } D
- Bottom up parsing 	- Depth first, left to right parsing 
- Every S-attributed definition is also L-attributed definition	- Bottom up evaluation is also possible

Intermediate code generation

3 address code - statement / sequence of variables / instruction which has atmost 3 address locations.

Other representations
(1) Graph & DAG / Syntax tree
(2) Postfix

Most popularly used forms of 3 address code are -
1. Quadruples
2. Triples

$-(a+b) * (c-d)$

$t_1 = a+b$

$t_2 = -t_1$

$t_3 = c-d$

$t_4 = t_2 * t_3$

Quadruples

	op	opr1	opr2	result
1)	+	a	b	t_1
2)	-	t_1		t_2
3)	-	c	d	t_3
4)	*	t_2	t_3	t_4

- Efficient optimization possible
- Flexible, i.e can swap instruction / statements ~~without any difficulty~~
- More space is required

Triples

number	op	opr1	opr2
1)	+	a	b
2)	-	(1)	
3)	-	c	d
4)	*	(2)	(3)

- NOT Efficient (in terms of ~~instruction swapping~~)
- Less flexible, i.e if we want to swap (1) & (3) above, we will have to change their references as well
- Less space required

Indirect Triples

number	op	opr1	opr2
1)	+	a	b
2)	-	(11)	
3)	-	c	d
4)	*	(12)	(13)

Statement	
1)	(11)
2)	(12)
3)	(13)
4)	(14)

Advantage
- Similar to quadruple
- Easy swapping of instruction as compared to Triples
- Space optimised as compared to quadruples.

Triple: references → var1. If var1 becomes var2, all references to be updated

Indirect Triple: reference → ptr1 — var1. if var1 becomes var2, only the ptr1 is updated