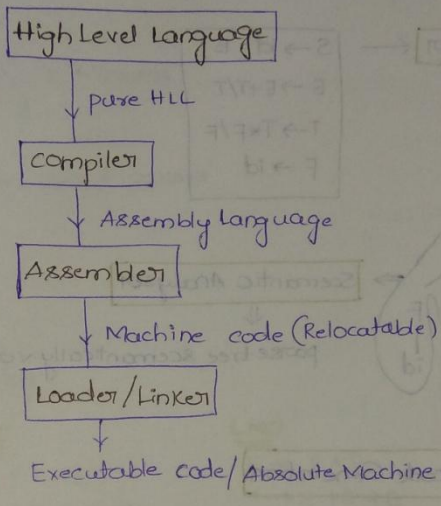
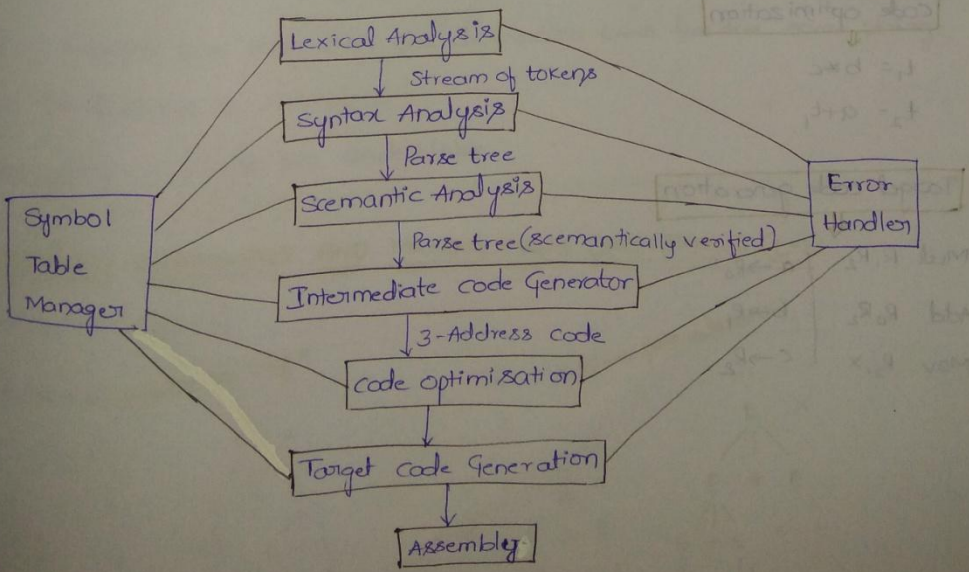
INTRODUCTION AND VARIOUS PHASES OF COMPILER (L-1)

⇒ The main aim of the compiler is to convert a High Level Language to Low level language.



⇒ Removing #include by including a specific file is called "File Inclusion"

⇒ "Assembler" is dependent on the platform [Hardware + OS].



Example :

$x = a + b * c$

⇓

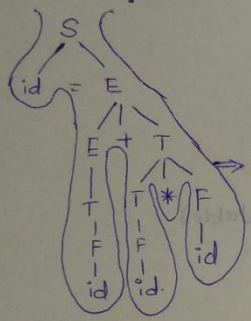
Lexical Analysis ⇒ The Lexical Analyzer identifies the "identifiers" tokens" is using some Regular expressions called patterns $l(l+d)^*$

$id = id + id * id$

⇓

Syntax Analyser ←

$S \rightarrow id = E;$
$E \rightarrow E+T/T$
$T \rightarrow T*F/F$
$F \rightarrow id$



⇒ **Semantic Analyser**

⇓

parse tree semantically verified

Intermediate code Generation

⇓

$t_1 = b * c$

$t_2 = a + t_1$

$x = t_2$

Code optimization

⇓

$t_1 = b * c$

$t_2 = a + t_1$

Target code generation

⇓

Mul R_1, R_2	$a \rightarrow R_0$
Add R_0, R_2	$b \rightarrow R_1$
Mov R_2, x	$c \rightarrow R_2$

⇒ This is the only phase that Reads the Input character by character

⇒
```
int max(x,y)
  int x,y;
  /* find max of x and y */
  {
     return (x>y ? x : y);
  }
```
25 Tokens are present

int = 1 token
max = 1 token
return = 1 token

⇒ printf("%d Hai", x); ⇒ 8 Tokens

(1 Token: "%d Hai")

1 2 3 4 5 6 7 8

⇒ Syntax Analyser is also called parser

Grammar:

E → E+E / E*E / id

⇒ id+id*id = Given string

LMD
E → E+E
→ id + E*E
→ id + id*E
→ id + id*id

RMD
E → E+E
→ E + E*E
→ E + E*id
→ E + id*id
→ id + id*id

LMD
E → E*E
→ E + E*E
→ id + E*E
→ id + id*E
→ id + id*id

RMD
E → E*E
→ E* id
→ E+E*id
→ E+id*id
→ id+id*id

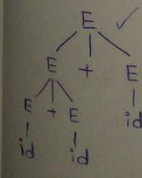
⇒ If a Gramar has more than one derivation trees for the same string then the Gramar is Ambigious

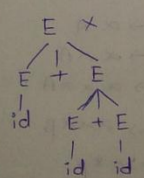
⇒ Ambiguity problems are undecidable

AMBIGIOUS GRAMMARS AND MAKING THEM UNAMBIGIOUS (L-3)

E → E+E / E*E / id

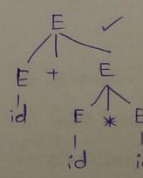
id+id+id ⇒ Associativity ✗

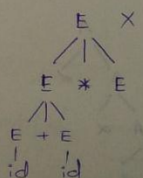




Leftmost plus should be evaluated first.

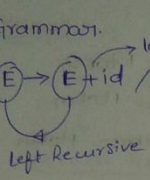
id+id*id ⇒ precedence (✗)

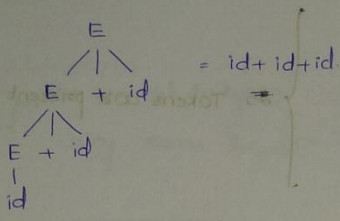




Highest precedence one should be evaluated first (* should be evaluated first)

To avoid the above Ambiguity we have to restrict the growth of the grammar.





→ The Grammar is said to be left recursive if the leftmost symbol in the RHS = LHS

⇒ Inorder to overcome the Associativity we need to define the Grammar to be Left Recursive

⇒ E → E+T/T
T → T*F/F
F → id

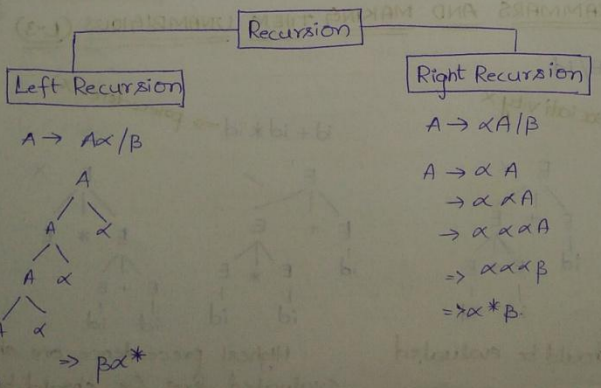
Associativity = Recursion
Precedence = Levels

⇒ 2↑3↑2 = 2^{3^2} = $(2)^{(3^2)}$ = 2^9

A → A $B/B
B → B#C/C
C → C@D/D
D → d

$, #, @ = Left Associative
$ > $, # > #, @ > @
$ < # < @

LEFT RECURSION ELIMINATION AND LEFT FACTORING OF GRAMMARS. (L-4)



$A \to \beta A'$
$A' \to \alpha A' / \epsilon$ $\Leftrightarrow$ $A \to A\alpha / \beta$

If the grammar is of the form $A \to A\alpha / \beta$ then eliminate Left Recursion by $A \to \beta A'$, $A' \to \alpha A' / \epsilon$

1) $E \to E+T / T$ ($A \to A\alpha / \beta$)

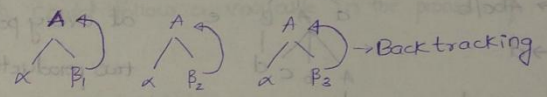
$E \to TE'$
$E' \to \epsilon / +TE'$ = Left Recursion Eliminated

2) $A \to \beta_1 A' / \beta_2 A' / \beta_3 A'$
$A' \to \alpha_1 A' / \alpha_2 A' / \alpha_3 A'$ } is the eliminated LR of the Grammar $A \to A\alpha_1 / A\alpha_2 / A\alpha_3 / \cdots / \beta_1 / \beta_2 / \beta_3 \cdots$

3) Grammars can be classified into various categories

G

Ambigious	unambigious
Left Recursive	Right Recursive
Deterministic	Non-Deterministic ($A \to \alpha\beta / \alpha\beta_2 / \alpha\beta_3$)



$\Rightarrow A \to \alpha A'$
$A' \to \beta_1 / \beta_2 / \beta_3$

① $S \to iEtS / iEtSeS / a$
$E \to b$

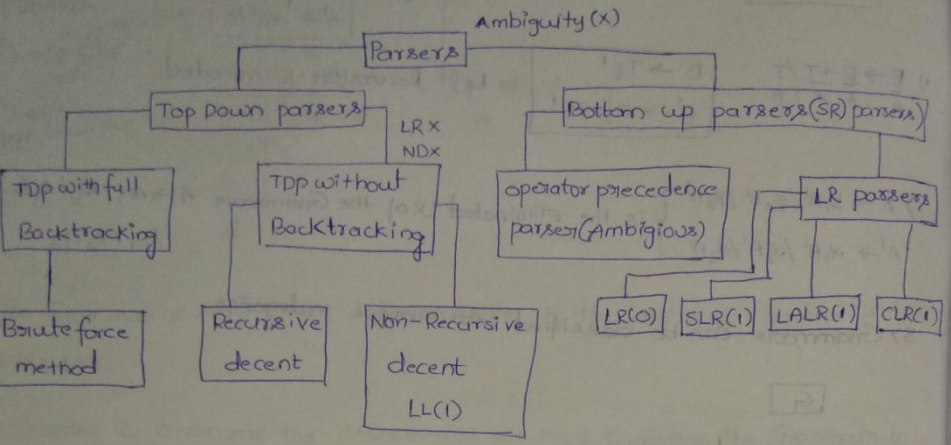
$S \to iEtSS' / a$
$S' \to eS / \epsilon$
$E \to b$

$\Rightarrow$ The elimination of Non-determinism doesnot guarentee the elimination of Ambiguity.

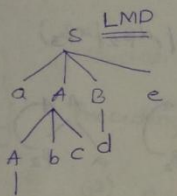
② $S \to aSSbS / aSaSb / abb / b$ $\Rightarrow$ $S \to aS' / b$
$S' \to SSbS / SaSb / bb$
$S' \to SS''$
$S'' \to SbS / aSb$

③ $S \to bSSaaS / bSSaSb / bSb / a$ $\Rightarrow$ $S \to bSS' / a$
$S' \to SaaS / SaSb / b$
$S' \to SaS'' / b$
$S'' \to aS / SbS$

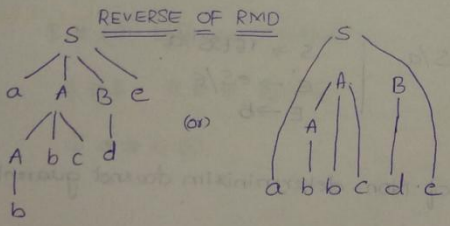
⇒ parsers are nothing but Syntax Analysers. (L-5)



→aABe
→ Abc/b
→d
: abbcde



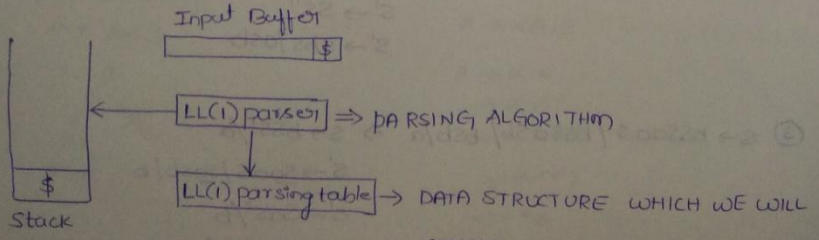
The main thing that we must assure is at every point when we have more than two productions to be choosen, "which one to choose"



⇒ The main decision that should be made is if I see a terminal when to Reduce

(1) PARSER / NON RECURSIVE DECENT PARSER

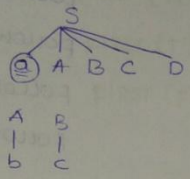
ft to Right, Left most Derivation. (1) = No. of look aheads



Now, before constructing the LL(1) parsing table we should know two functions they are FIRST AND FOLLOW

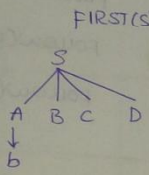
FIRST():

S→a ABCD
A→b
B→C
C→d
D→e



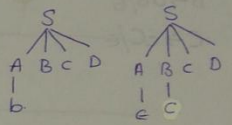
FIRST(S) = a
FIRST(A) = b
FIRST(B) = C
FIRST(C) = d
FIRST(D) = e

S→ ABCD
A→b
B→C
C→d
D→e



S→ABCD
A→ b/ϵ
B→ c (small c)
C→d
D→e

FIRST(S) = {b, c}



FOLLOW():

⇒ what is the terminal which could follow a variable in the process of derivation.

⇒ follow of start symbol always contain '$'

S→ABCD
A→b/ϵ
B→c/ϵ
C→d
D→e

FOLLOW(S) = {$}
FOLLOW(A) = FIRST(B,C,D) = {c,d,e}
FOLLOW(B) = FIRST(C,D) = {d}
FOLLOW(C) = FIRST(D) = {e}

FOLLOW NEVER CONTAIN "EPSILON"

⇒ Now, If nothing is following a variable i.e If nothing is at the RHS of a variable then the follow of that variable is the follow of LHS

∴ FOLLOW(D) = FOLLOW(S) = {$}

1) S → ABCDE
A → a/ϵ
B → b/ϵ
C → c
D → d/ϵ
E → e/ϵ

FIRST(S) = {a, b, c}
FIRST(A) = {a, ϵ}
FIRST(B) = {b, ϵ}
FIRST(C) = {c}
FIRST(D) = {d, ϵ}
FIRST(E) = {e, ϵ}

FOLLOW(S) = {$}
FOLLOW(A) = {b, c}
FOLLOW(B) = {c}
FOLLOW(C) = {d, e, $}
FOLLOW(D) = {e, $}
FOLLOW(E) = {$}

S → Bb/cd
B → aB/ϵ
C → cC/ϵ

FIRST(S) = {a, b, c, d}
FIRST(B) = {a, ϵ}
FIRST(C) = {c, ϵ}

FOLLOW(S) = {$}
FOLLOW(B) = {b}
FOLLOW(C) = {d}

E → TE'
E' → +TE'/ϵ
T → FT'
T' → *FT'/ϵ
F → id/(E)

FIRST(E) = {id, (}
FIRST(E') = {+, ϵ}
FIRST(T) = {id, (}
FIRST(T') = {*, ϵ}
FIRST(F) = {id, (}

FOLLOW(E) = {$,)}
FOLLOW(E') = {$,)}
FOLLOW(T) = {+, $,)}
FOLLOW(T') = {+, $,)}
FOLLOW(F) = {*, +, $,)}

S → ACB/CbB/Ba
A → da/BC
B → g/ϵ
C → h/ϵ

FIRST(S) = {ϵ, d, g, h, b, a}
FIRST(A) = {d, g, h, ϵ}
FIRST(B) = {g, ϵ}
FIRST(C) = {h, ϵ}

FOLLOW(S) = {$}
FOLLOW(A) = {h, g, $}
FOLLOW(B) = {$, a, h, g}
FOLLOW(C) = {g, $, b, h}

S → aABb
A → C/ϵ
B → d/ϵ

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

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

S → aBDh
B → cC
C → bC/ϵ
D → EF
E → g/ϵ
F → f/ϵ

FIRST(S) = {a}
FIRST(B) = {c}
FIRST(C) = {b, ϵ}
FIRST(D) = {g, f, ϵ}
FIRST(E) = {g, ϵ}
FIRST(F) = {f, ϵ}

FOLLOW(S) = {$}
FOLLOW(B) = {g, f, h}
FOLLOW(C) = {g, f, h}
FOLLOW(D) = {h}
FOLLOW(E) = {f, h}
FOLLOW(F) = {h}

$E \to TE'$

$E' \to +TE'/\epsilon$

$T \to FT'$

$T' \to *FT'/\epsilon$

$F \to id/(E)$

FIRST(E) = {id, (}

FIRST(E') = {+, ϵ}

FIRST(T) = {id, (}

FIRST(T') = {*, ϵ}

FIRST(F) = {id, (}

FOLLOW(E) = {$,)}

FOLLOW(E') = {$,)}

FOLLOW(T) = {+, $,)}

FOLLOW(T') = {+, $,)}

FOLLOW(F) = {*, +, $,)}

NOW THE LL(1) PARSING TABLE LOOKS LIKE

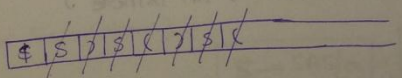
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$F \to (E)$		

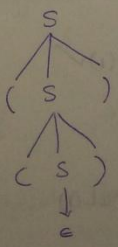
2) $S \to (S)/\epsilon$

	(	)	$
S	$S \to (S)$	$S \to \epsilon$	$S \to \epsilon$

All Grammars are not feasable for LL(1) parsing.

Now, w = (())$





ANY GRAMMAR WHICH IS LEFT RECURSIVE/ND CANNOT BE USED FOR LL(1) PARSING

Non-deterministic

) $S \to aSbS / bSaS / \epsilon$

{a} {b} {a,b,$} ⇒ Not LL(1)

↳ which means the production must be placed under 'S' row and {a} column.

↳ since the $S \to \epsilon$ production should be placed under the follow(s) which are {a,b}, we have already placed production in S row and a column which means a single CELL has more than one entry, so the grammar is not LL(1) ✗

) $S \to aABb$ ⇒ {a}

$A \to c/\epsilon$ ⇒ {c} and follow(A) = {d, b} ($A \to c$, $A \to \epsilon$)

$B \to d/\epsilon$ ⇒ {d} and {b} ⇒ IS LL(1) ✓

) $S \to A/a$ {a} ∩ {a} = Not LL(1) ✗ (conflict)

$A \to a$ {a}

) $S \to aB/\epsilon$ {a} {$} ⇒ Not x^n

$B \to bC/\epsilon$ {b} {$} ⇒ No x^n

$C \to cS/\epsilon$ {c} {$} ⇒ No x^n

} LL(1) ✓

) $S \to AB$

$A \to a/\epsilon$ ⇒ {a} {b,$} ⇒ No x^n

$B \to b/\epsilon$ ⇒ {b} {$} ⇒ No x^n

} LL(1) ✓

$S \to aSA/\epsilon$ ⇒ {a} {c,$} ⇒ No x^n

$A \to c/\epsilon$ ⇒ {c} {c} ⇒ x^n is there

} ✗

$S \to A$

$A \to Bb/cd$ ⇒ {a,b} {c,d}

$B \to aB/\epsilon$ ⇒ {a} {b}

$C \to cC/\epsilon$ ⇒ {c} {d}

} LL(1) ✓

$S \to aAa/\epsilon$ {a} {$,a}

$A \to abS/\epsilon$

} x Not LL(1)

(9) $S \to iEtSS'/a$ {i} {a}

$S' \to es/\epsilon$ {e} {e}

$E \to b$

} Not LL(1) ✗

$E \rightarrow iE'$

$E' \rightarrow +iE'/\epsilon$

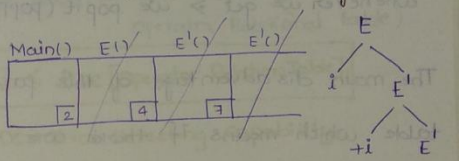
⇒ The parser is called Recursive Decent parser because for every variable we are going to write Recursive functions.

```
E()
{
 1 if (l==='i')
 2{
      match('i');
   }
 3 E'();
}4
l=getchar();
```

```
E'()
{
 1) if(l=='+')
 {
   2) match('+');
   3) match('i');
    4) E'();
 }
5)else 6)return;
}7)
```

```
match(char t)
{
 if (l==t)
   l=getchar();
 else
   printf("error");
}
```

{(i+i $)}is generated from above grammar




```
main()
{
 1) E()
 2) if(l==='$')
 3) printf("parsing success");
}
```

⇒ This parser uses Recursion stack for parsing.

⇒ Here l- look Ahead

OPERATOR GRAMMAR AND OPERATOR PRECEDENCE PARSER (L-9)

OPERATOR GRAMMAR

A Grammar that is used to define the mathematical operations is called operator Grammar (with some Restrictions)

⇒ No Two variables must be Adjacent

⇒ No epsilon productions

$E \rightarrow E+E/E*E/id$ (✓)

$E \rightarrow EAE/id$ } Not operator Grammar

$A \rightarrow +/*$

$S \rightarrow SAS/a$, $A \rightarrow bSb/b$ ⇒ $S \rightarrow SbSbS/SbS/a$, $A \rightarrow bSb/b$

Not operator precedence ⇒ Operator precedence Grammar.

⇒ This parser parses the Ambigious grammars by Creating operator Relation Table

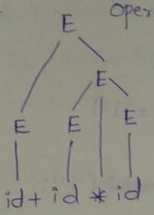
	id	+	*	$
id	–	>	>	>
+	<	>	<	<
*	<	>	>	>
$	<	<	<	–

The given grammar is

$E \to E+E / E*E / id$

⇒ id will have higher precedence than any other operator

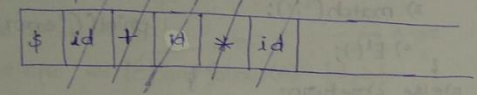
⇒ '$' will have least precedence than any other operator



⇒ Top of the stack will be $.

W = id + id * id $

look head



Now The Algorithm goes like this

1) when the top of the Stack is < than the lookAhead then push it and whenever we get > we pop it (popping means actually we Reduced it)

The main disadvantage of this parser is the size of the operator Relational table which means if there are 4 operators then Size of the table is $16(4^2)$ and if there are 5 operators then there would be 25 entries(5^2). So Generally if there are 'n' operators, the size of the table is $O(n^2)$

Now, To reduce the size of the table we use operator function table

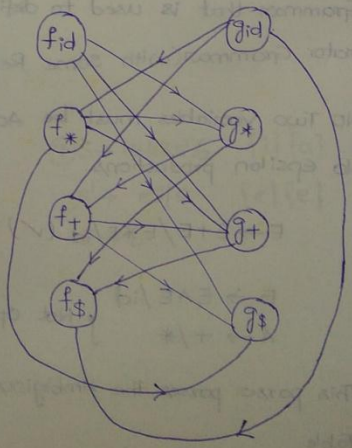
	id	+	*	$ → function 'G'
id	–	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	<

↓

Function (F)

$> = F_{id} \to g_{id}$ (F → G)

$< = g_{id} \to f_{id}$ (G → F)



Now, the Longest path from g_{id} is $g_{id} \to f_* \to g_* \to f_+ \to g_+ \to f_\$$

Similarly find the longest paths from each node the function table looks like

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

$f_{id} \xrightarrow{1} g_* \xrightarrow{3} f_+ \xrightarrow{3} g_+ \xrightarrow{4} f_\$ = 4$

$f_+ \xrightarrow{1} g_+ \xrightarrow{2} f_\$ = 2$

Now if incead to compare (+, +) $\Rightarrow f_+ \quad g_+$

$\Rightarrow \quad 2 \quad 1 \quad \Rightarrow 2 > 1 \Rightarrow (+ \gtrdot +)$

∴ The Size of the table = $\boxed{O(2n)}$ n = no. of operators.

⇒ In the functional table we dont have Blank entries (Black Entries are nothing but errors) so, the error detecting capability of the functional table is less than that of operator Relation table (we have Blank entries in Operator Relational table)

$$\boxed{EDC\,[\text{Operator Functional Table}] < EDC\,[\text{Operator Relation Table}]}$$

EDC = Error Detecting Capability.

2> $P \to SR/S$

$R \to bSR/bS$

$S \to wbS/w$

$w \to L*w/L$

$L \to id$

$\Rightarrow$

$P \to SbP/SbS/S$

$S \to wbS/w$

$w \to L*w/L$

$L \to id$

	id	*	b	$
id	−	⋗	⋗	⋗
*	⋖	⋖	⋗	⋗
b	⋖	⋖	⋖	⋗
$	⋖	⋖	⋖	−

⇒ Here * is defined as Right Associative ($w \to L*w$) so the Right side star has highest precedence

∴ $* \lessdot *$

1) $S \to AA$
   $A \to aA/b$

} In LR parsers we have CLOSURE and GOTO Operations

The Augmented Grammar is $S' \to S$, $S \to AA$, $A \to aA/b$

> Any production with a dot in the RHS is called an item.

The LR(0) parsing Tree is,

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

The parsing table is

State	ACTION a	ACTION b	ACTION $	GOTO A	GOTO S
0	S_3	S_4		2	1
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		

CANONICAL COLLECTION OF LR(0) ITEMS

1. While writing the Reduce moves / while inspecting final items go to the productions and check

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

Now I_4 is $A \to b \cdot$ (3rd production)
$\Rightarrow I_4 \to R_3$

$S' \to S$

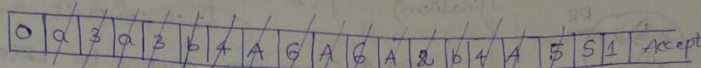
$S \to AA$ (1)

$A \to aA/b$

(2) (3)

Let the given string be aabb$

Input pointer → aabb $



⇒ Always the top of the stack contains state (and first state will be Zero)

⇒ Initially I_0 on 'a' is S_3 which means shift the Input you are looking at and as well as the state no. on to the stack ⇒ [Input = a, state = 3] = a|3 and increment the Input pointer [continue]

⇒ Now, top of the stack is '4' and Input pointer is b which means R_3 which states that Reduce the production no. 3. which means reduce the previous 'b'. (previous symbol). ↳ (A → b)

⇒

⇒ Now In the stack how could we make Reduce move is look the RHS of the production that must be Reduced, and find the length of RHS. In this example 'A → b' is the production ⇒ length of RHS = 1 (∵ |b| = 1) which means pop 2 symbols (1×2) from stack. (If the length of RHS is 'x' then pop '2x' elements from stack) and push the LHS symbol onto the stack, and see the stack for the last used state number it turns out that it is '3' and look what '3 on A' is generating = 6, so push '6' onto the stack. when we see reduce moves we don't increment Input pointer.

SLR(1)

	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			Accept		
2	S3	S4		5	
3	S3	S4		6	
4	R3	R3	R_3		
5			R_1		
6	R_2	R_2	R_2		

R_1 : S → AA

place R_1 in follow(S)

follow(S) = {$}

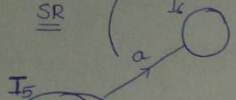
Now,

→ R_3 ⇒ A → b should be reduced

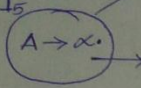
place R_3 in follow(A) = {a|b|$}

place the Reduce moves if the next Symbol is in the follow of current Symbol, this is the main difference between the LR(0) and SLR(1) parsers

SR



I_5



→ Tells Reduce operation (final item)

⇒ All the Gramars are not suitable for LR parsing due to shift Reduce conflicts.

	a	b
5	s/r_3	r_1

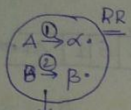
SR-CONFLICT

	a	b	c
5	r_1/r_2	r_1/r_2	r_1/r_2

↳ SR-CONFLICT

RR

I_5



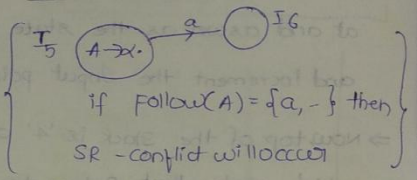
⇒ A Grammar cannot be LR(0) if i have SR/RR conflicts

Follow(A) ∩ follow(B) ≠ ϕ (RR - will occur)

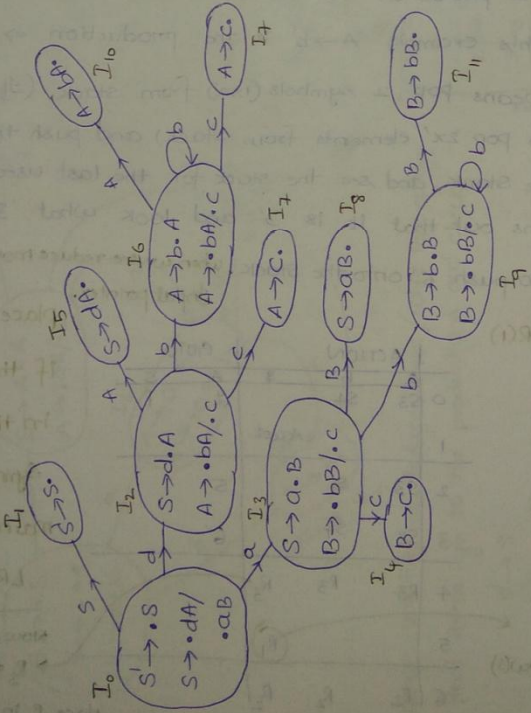
EXAMPLES OF LR(0) AND SLR(1) (L-12)

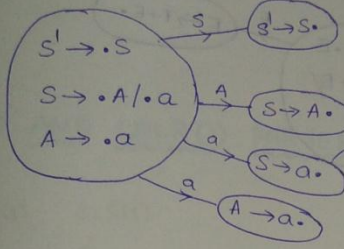
1) S → dA/aB {d,a}
   A → bA/c {b,c}
   B → bB/c {b,c}

1) LL(1) ✓
2) LR(0) ✓
3) SLR(1) ✓



CANONICAL COLLECTION OF LR(0) ITEMS





follow(S) = $ } Not SLR(1)
follow(A) = $

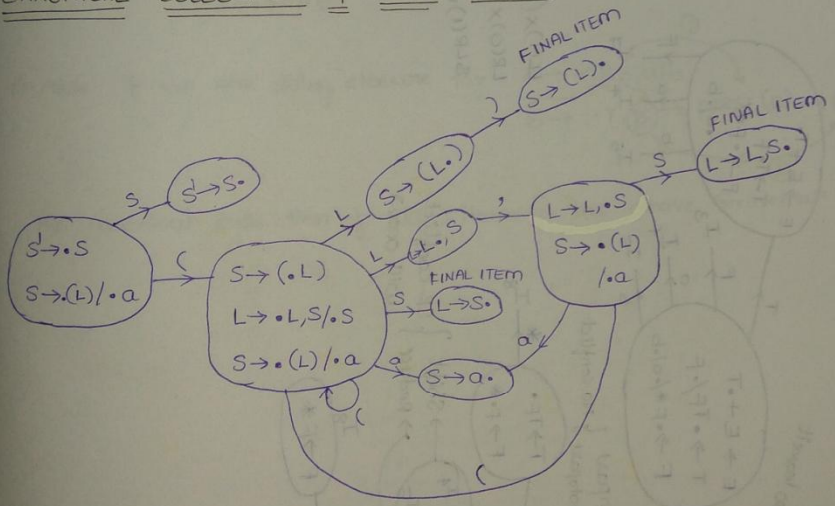
2 Reduce moves from the same state
RR-conflict so LR(0) ✗ (If we get 2 reduce moves from the same state then the grammar is not LR(0).

③ S→(L)/a
L→L,S/S

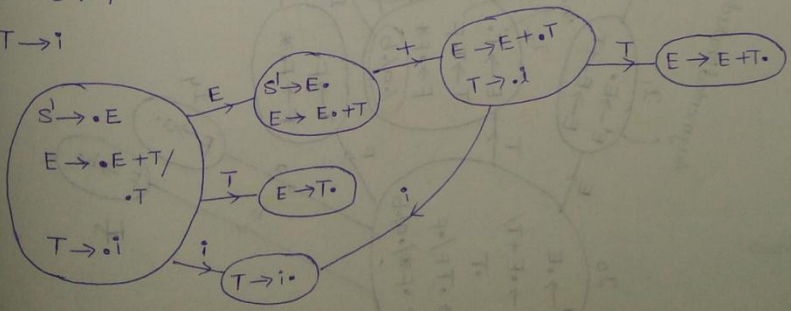
LL(1) ✗ (Left Recursive)
LR(0) ✓
SLR(1) ✓

⇒ Not

CANONICAL COLLECTION OF LR(0) ITEMS



④ E→E+T/T
T→i



L-13

⑥

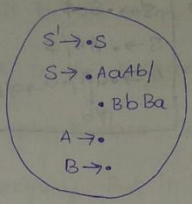
E→E+T
/T
T→TF/
F
F→F*/a/b

LL(1)× (Left Recursive)
LR(0)×
SLR(1)✓

17 $S \to AaAb / BbBa \{a,b\}$ LL(1) ✓

$A \to \epsilon$ LR(0) ×

$B \to \epsilon$ SLR(1) ×



	a	b	$
0	r_3/r_4	r_3/r_4	

RR- conflict

CLR(1) AND LALR(1) PARSERS L-14

LALR(1) CLR(1)

LR(1) Items = LR(0) Items + Lookhead

① $S \to AA$

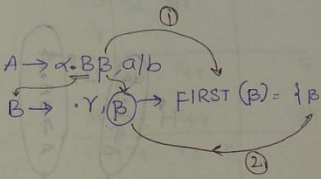
$A \to aA/b$

$S' \to \cdot S$ is the Augmented Grammar

$S \to \cdot AA$

$A \to \cdot aA/\cdot b$

In case if we are doing closure for $A \to \alpha \cdot B\beta, a/b$



The canonical collection of LR(1) items for the above Grammar is

Contd in next

CANONICAL COLLECTION OF LR(1) ITEMS

production.

tables, the main difference arises in the placement of the final items In the LR(0) and SLR(1) we are going to place in the entire row and the follow of LHS (in SLR(1)) respectively. But in CLR and LALR(1) we are going to place the reduce moves only in look ahead symbols.

⇒ From the above diagram $[I_3, I_6]$ have same LR(0) items but differ in look heads

⇒ Similarly $[I_4, I_7]$, $[I_8, I_9]$ also have same LR(0) items but differ in look aheads.

CLR(1) PARSING TABLE

	a	b	$	S	A
0	S_3	S_4			2
1					
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S6	S7			9
7			r_3		
8	r_2	r_2			
9			r_2		

LALR(1) TABLE

	a	b	$	S	A
0	S_{36}	S_{47}			2
1					
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		

21

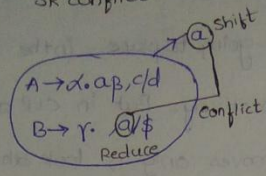
$[I_3, I_6] \Rightarrow I_{36}$

$[I_4, I_7] \Rightarrow I_{47}$

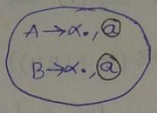
$[I_8, I_9] \Rightarrow I_{89}$

[No of states in CLR(1)] ≥ [[No of states in SLR(1)] = [No of states in LALR(1)] = [No of States in LR(0)]]

SR conflict

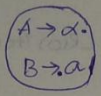


RR conflict



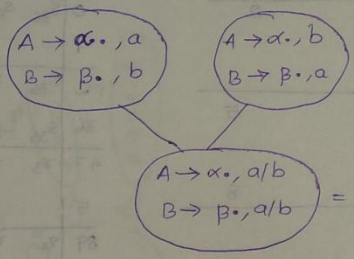
LR(1) Items

LR(0) Items



⇒ If the grammar is not CLR(1) then the Grammar is not LALR(1) because we reduce the size of the table but not the conflicts in LALR(1) parser

⇒ If the Grammar is CLR(1) then it may or maynot be LALR(1)



i) S→ AaAb / BbBa

A→ ∈

B→ ∈

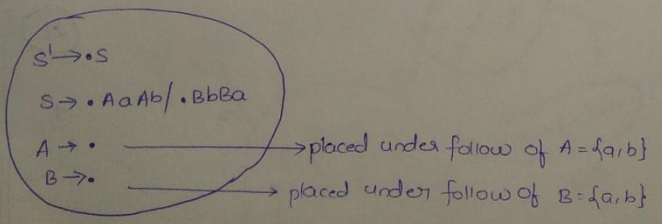
LL(1) ×

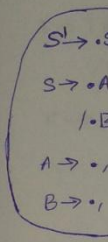
LR(0) ×

SLR(1) ×

CLR(1) ✓

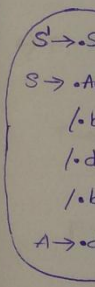
LALR(1) ✓

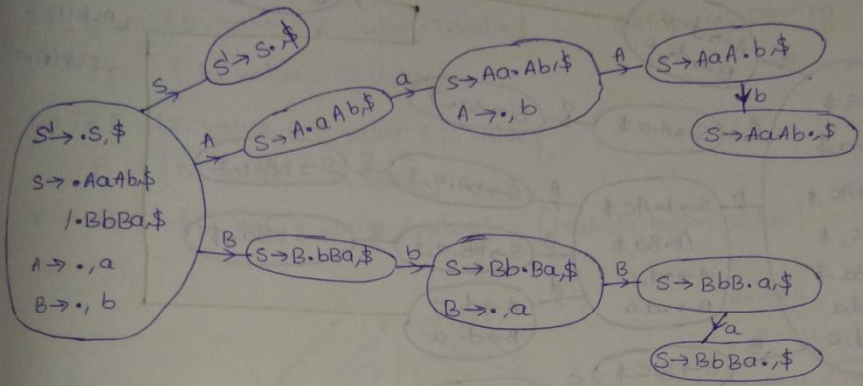




② S→ Aa

A→d



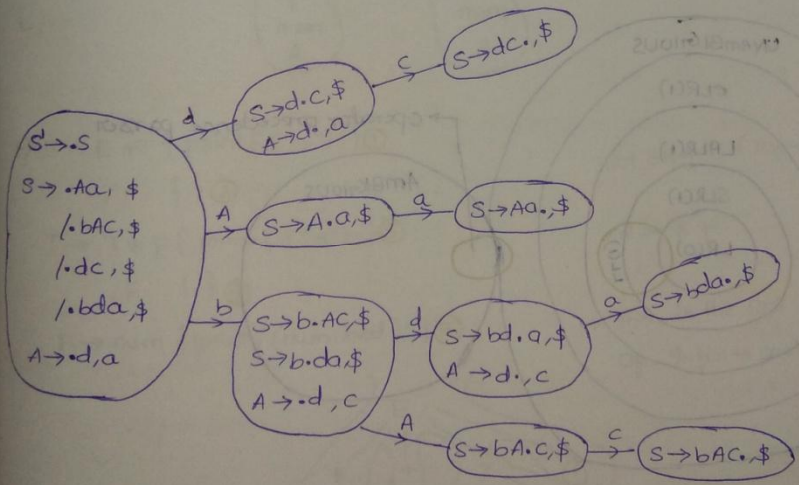


(2)

S→ Aa/bAC/dc/bda

A→d

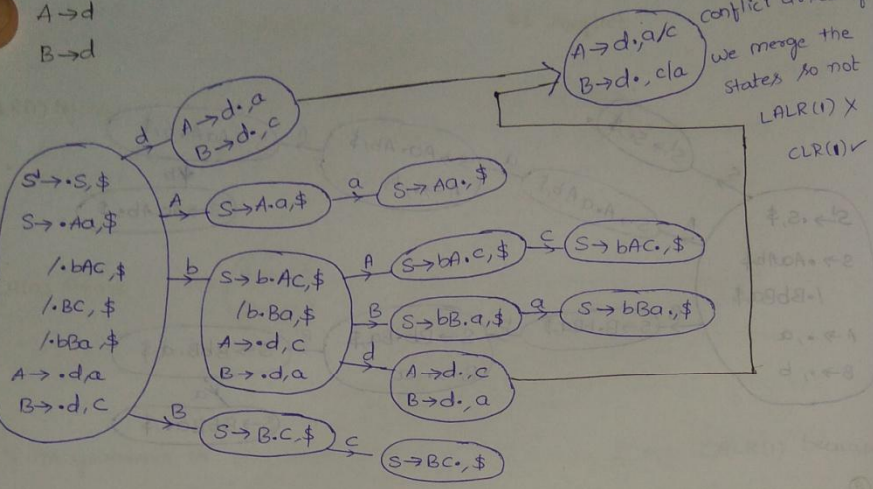
LL(1) X
LR(0) X
SLR(1) X
CLR(1) ✓
LALR (1)



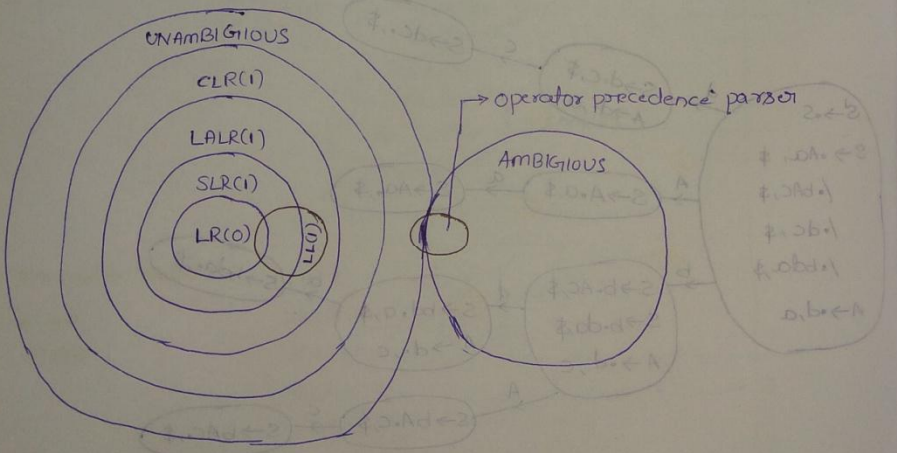
(3) S → Aa/bAc/Bc/bBa

A → d

B → d



COMPARISION OF THE PARSERS (L-16)



- Every Grammar which is LL(1) is definitely LALR(1)

SYNTAX

⇒ Gramm

1) E → E+
/T
T → T*F
/F
F → nu

2) E → E
/
T → T
/
F →

⇒ Grammar + Semantic Rules = SDT

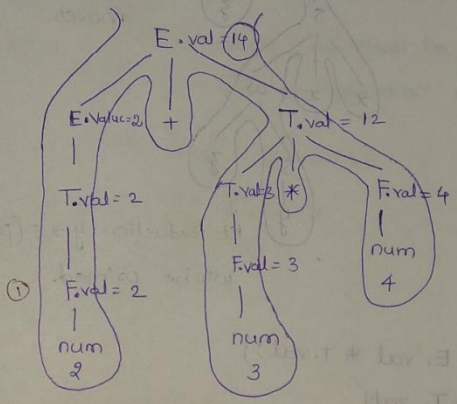
1) $E \to E+T$ { E.value = E.value + T.value } Bottom-up parsing.

/T { E.value = T.value }

$T \to T*F$ { T.value = T.value * F.value }

/F { T.value = F.value }

$F \to num$ { F.val = num.lvalue } { lvalue = lexim value }



2+3*4
= 2+(3*4)
= 2+12 = 14

2) $E \to E+T$ { printf("+"); } ①

/T { } ②

$T \to T*F$ { printf("*"); } ③

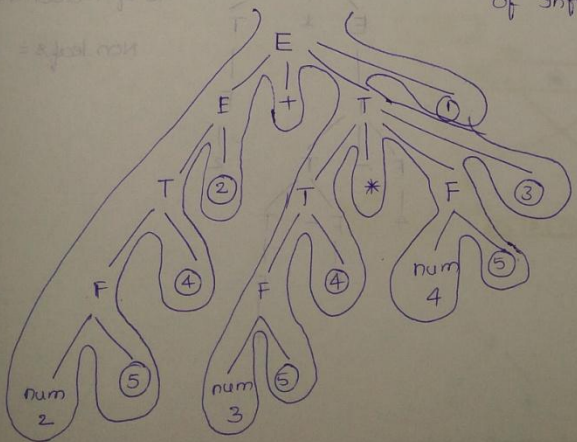
/F { } ④

$F \to num$ { printf(num.lval); } ⑤

2+3*4 (Top-down parser)

234*+

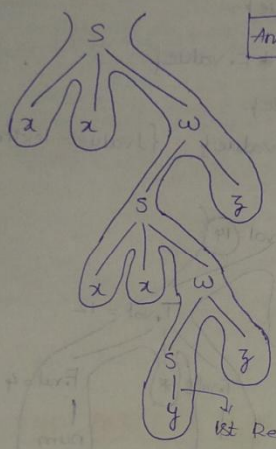
⇒ This is the SDT for convertion of Infix to postfix Expression



Top down parsing

③ $S \to xxw$ { printf(1); }
/ y { printf(2); }
$w \to Sz$ { printf(3); }

String xxxyzz



Ans: 2 3 1 3 1

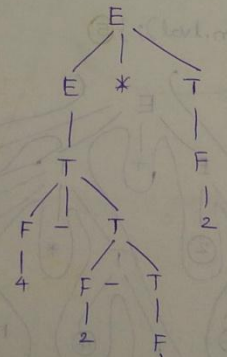
⇒ Carry the action on Reduce moves

1st Reduction $y \to S$ (printf(2)); '2' will be printed.

④

$E \to E * T$ { E.val = E.val * T.val; }
/ T { E.val = T.val; }
$T \to F - T$ { T.val = F.val - T.val; }
/ F { T.val = F.val; }
$F \to 2$ { F.val = 2; }
/ 4 { F.val = 4; }

D = 4 - 2 - (4 * 2)
w = ((4 - (2 - 4)) * 2)
= (4 - (-2)) * 2
= (4 + 2) * 2
= 12



No. of Reductions = No. of Non leafs = 10

/T {T.val = T.val}

T → T&F {T.val = T.val + F.val}

/F {T.val = F.val}

F → num {F.val = num.lvalue;}

ω = 2#3&5#6&4 what is the output

= 2*3+5*6+4

= ((2*3)+(5*6)+4)

= (6+30+4)

= 40

⇒ wrong because '+' is defined at highest level (Bottom level) and must be evaluated first and then multiplication must be Evaluated.

⇒ 2*(3+5)*(6+4)

= 2*(8)*(10)

= 160 ✓

L-18

SDT TO BUILD SYNTAX TREE

E → E1+T {E.nptr = mknode (E1.nptr, '+', T.nptr)}

/T {E.nptr = T.nptr;}

T → T1*F {T.nptr = mknode (T1.nptr, *, F.nptr);}

/F {T.nptr = F.nptr}

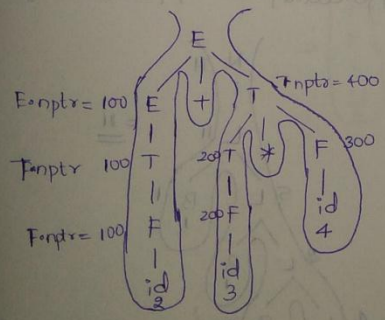
F → id {F.nptr = mknode (null, id.name, null);}

Returns Address

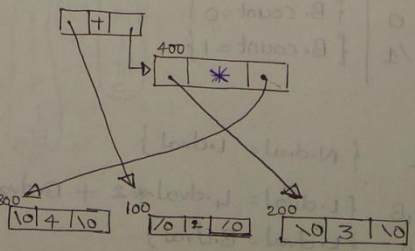
mknode = makenode

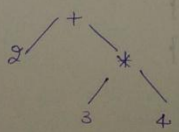
nptr = node pointer

W = 2+3*4



Concrete Syntax tree





Abstract Syntax tree

$E \rightarrow E_1 + E_2$ {if ((E1.type == E2.type) && (E1.type = int)) then E.type = int else error}

/ $E_1 == E_2$ {if ((E1.type == E2.type) && (E1.type = int / boolean)) then E.type = boolean else error}

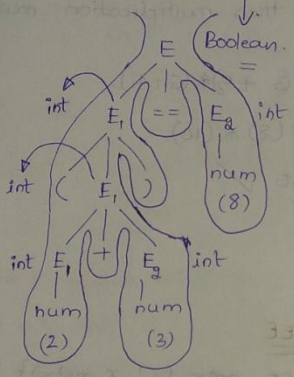
/ (E1) { E.type = E1.type }

/ num { E.type = int; }

/ True { E.type = boolean }

/ False { E.type = boolean }

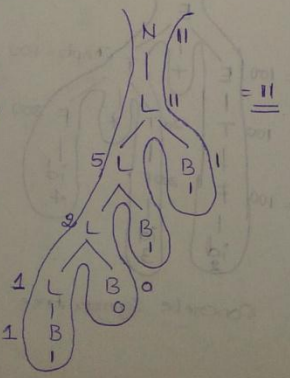
w = {(2+3) == 8} = Boolean Expression



	count all 1's	count all 0's	No. of Bits
N → L	{ N.count = L.count }	⇒	⇒
L → L1 B	{ L.count = L1.count + B.count }	⇒	⇒
/ B	{ L.count = B.count }	⇒	⇒
B → 0	{ B.count = 0 }	{ B.count } = 1	{ B.count = 1 }
/ 1	{ B.count = 1 }	{ B.count } = 0	{ B.count = 1 }

N → L	{ N.dval = L.dval }
→ L1 B	{ L.dval = L1.dval * 2 + B.dval }
/ B	{ L.dval = B.dval }
B → 0	{ B.dvalue = 0 }
/ 1	{ B.dvalue = 1 }

w = 1011



d value

• (01) = $1/2^2$ = 0.25

decimal value

• (11) ⇒ $3/4$ = 0.75

③ N → $L_1 . L_2$ | {N.dval = L_1.dval + (L2.dval / $2^{L2.count}$)}

L → L B / B | {L.dvalue = {L_1.dval + B.dval}} {L.count = L_1.count + B.count}

B → 0 | {L.count = B.count}{L.dval = B.dvalue}

/1 | {B.count = 1, B.dvalue = 0}

{B.count = 1, B.dvalue = 1}

SDT TO GENERATE THREE ADDRESS CODE

⑩ S → id = E {gen (id.name = E.place);}

E → E_1 + T {E.place = newTemp(); gen (E.place = E_1.place + T.place);}

/T {E.place = T.place;}

T → T_1 * F {T.place = newtemp(); gen (T.place = T_1.place * F.place);}

/F {T.place = F.place}

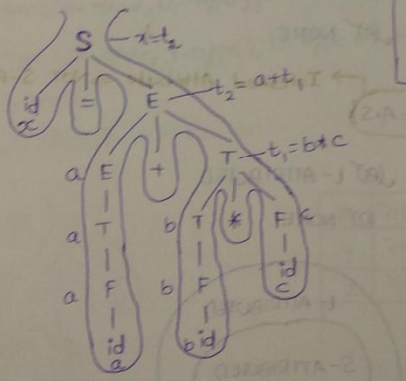
F → id {F.place = id.name}

w ⇒ x = a + b * c

t_1 = b * c

t_2 = a + t_1

x = t_2



DIFFERENCE BETWEEN S-ATTRIBUTED AND L-ATTRIBUTED SDT

S- ATTRIBUTED SDT

1) Uses only synthesized attributes

2) Semantic actions are placed at Right end of production

$A \rightarrow BCC\{\ \}$

3) Attributes are evaluated during Bottom up parsing

L- ATTRIBUTED SDT

1) Uses Both inherited and Synthesized attributes. Each inherited attribute is restricted to inherit either from parent or Left sibilings only.

Ex: $A \rightarrow XYZ$ $\{Y.S = A.S,\ Y.S = X.S,\ Y.S = Z.S\}$ (✓, ✓, ✗)

2) Semantic Actions are placed anywhere on RHS

$A \rightarrow \{\ \}BC$

$/D\{\ \}E$

$/FG\{\ \}$

3) Attributes are evaluated by traversing parse tree depth first left to Right

Inherited Attribute

Not S-ATTRIBUTED

① $A \rightarrow LM$ $\{L.i = f(A,i);\ M.i = f(L,S);\ A.S = f(M.S);\}$

$A \rightarrow QR$ $\{R.i = f(A,i),\ Q.i = f(R,i);\ A.S = f(Q,S);\}$ (✗)

→ NOT L-ATTRIBUTED

(A) S- ATTRIBUTED (B) L-ATTRIBUTED

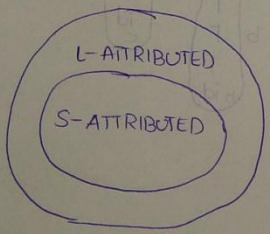
(C) BOTH (D) NONE

② $A \rightarrow BC$ $\{B.S = A.S\}$

→ Inherited Attribute ⇒ NOT S-ATTRIBUTED

a) S-ATTRIBUTED (B) L-ATTRIBUTED

(C) BOTH (D) NONE



INFORMATION INTO SYMBOL TABLE

① $D \rightarrow TL$ {L.in = T.type} ⇒ Inherited Attribute

$T \rightarrow$ int {T.type = int;} ⇒ Synthesized Attribute

/char {T.type = char;}

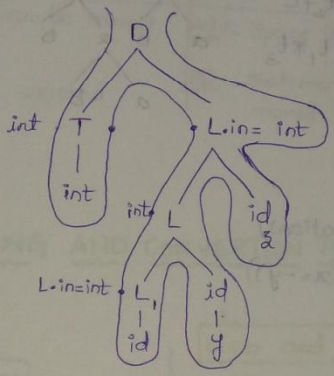
L-ATTRIBUTED

int x, y, z;

$L \rightarrow L_1$, id {L_1.in = L.in, add type (id.name, L_1.in);}

/id {add type (id.name, L.in)} ↳ Inherited

x	int
y	int
z	int



⇒ Evaluate the synthesized attribute when you last visit it

⇒ Evaluate the Inherited attribute when you first visit it.

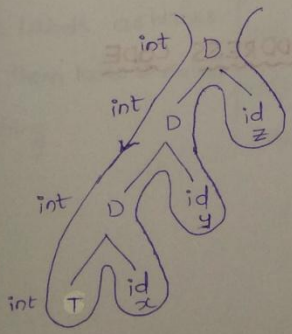
S-ATTRIBUTED SDT FOR THE SAME QUESTION

② $D \rightarrow D_1$, id {add-type (id.name, D_1.type)}

/T id {add-type (id.name, T.type), D.type = T.type}

$T \rightarrow$ int {T.type = int;}

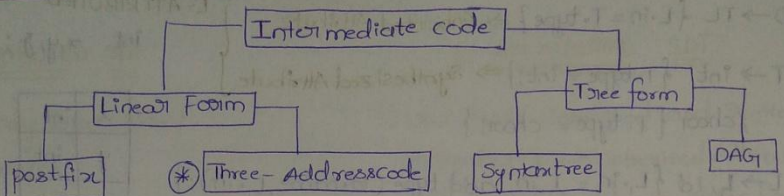
/char {T.type = char;}



x	int
y	int
z	int

INTRODUCTION TO INTERMEDIATE CODE

Ex: (a+b) * (a+b+c)



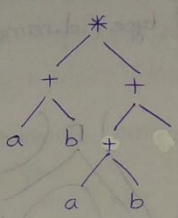
ab+ab+c+*

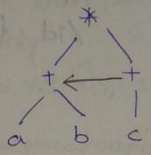
$t_1 = a+b$

$t_2 = a+b$

$t_3 = t_2 + c$

$t_4 = t_1 * t_3$





TYPES OF 3-ADDRESS CODE

1> x = y op z (x=a+b (Binary operation))

2) x = op z (unary operation (x = -y))

3) x = y (Assignment)

4> if x (rel op) y goto L (If 'x' (relational operator) y goto L)

5> goto L ⇒ (unconditional)

6> A[i] = x (Array indexing)

y = A[i]

7> x = *p ⇒ pointer

y = &y ⇒ Address of variable assigned to another variable

VARIOUS REPRESENTATIONS OF 3-ADDRESS CODE

⇒ (a+b) * (c+d) + (a+b+c)

1) $t_1 = a+b$

2) $t_2 = -t_1$

3) $t_3 = c+d$

4) $t_4 = t_2 * t_3$

5) $t_5 = a+b$

6) $t_6 = t_5 + c$

7) $t_7 = t_4 + t_6$

	Opr	Op1	Op2	Result
1)	+	a	b	t_1
2)	−	t_2	NULL	t_2
3)	+	c	d	t_3
4)	*	t_2	t_3	t_4
5)	+	a	b	t_5
6)	+	t_5	c	t_6
7)	+	t_4	t_6	t_7

Adv: Statements can be moved Around

Dis: More Space wasted

	Opr	Op1	Op2
1)	+	a	b
2)	−	(1)	
3)	+	c	d
4)	*	(2)	(3)
5)	+	a	b
6)	+	(5)	c
7)	+	(4)	(6)

Adv: Space is not wasted

Dis: Statements cannot be moved

INDIRECT

1)	(1)
ii)	(2)
iii)	(3)
iv)	(4)
v)	(5)
vi)	(6)
vii)	(7)

Adv: State ments can be moved

Dis: Two Access of Memory.

BACK PATCHING AND CONVERTION TO 3-ADDRESS CODE

Back Patching

```
If (a<b) then t=1
else t=0
```

(i): if a<b goto (i+3)

(i+1): t=0

(i+2): goto (i+4)

(i+3): t=1

(i+4): Return.

Leaving the Labels as blanks and filling them later is called Back patching

t_1 t_2 t_3

a<b and c<d or e<f

100) if (a<b) goto 103

101) $t_1 = 0$

102) goto 104

103) $t_1 = 1$

104) if (c<d) goto 107

105) $t_2 = 0$

106) goto 108

107) $t_2 = 1$

108) if (e<f) goto 111

109) $t_3 = 0$

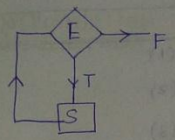
110) goto 112

111) $t_3 = 1$

112) $t_4 = [t_1$ and $t_2]$

113) $t_5 = [t_4$ or $t_3]$

① While E do S



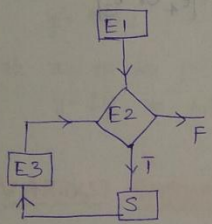
L: if (E==0) goto L1
S
Goto L
L1:

(or) if (E) goto L1
goto last
L1: S
goto L

② while (a<b) do
x = y+z

L: if a<b goto L1
goto last
L1: t = y+z
x = t
goto L
last:

③ for (E1; E2; E3)



for (i=0; i<10; i++)
a=b+c;

i = 0
L: if (i<10) goto L1
goto last
L1: t_1 = b+c
a = t_1
t = i+1
i = t
goto L
last:

(8) switch (i+j)
{
case (1): a=b+c;
break;
case (i): p=q+R;
break;
default: x=y+z;
break;
}

t = i+j
goto test
L1: t_1 = b+c
a = t_1
goto last
L2: t_2 = q+R
p = t_2
goto last
L3: t_3 = y+z
x = t_3
goto last
test: if (t==1) goto L1
if (t==2) goto L2
goto L3
last :

TWO DIMENSIONAL ARRAY TO 3-ADDRESS CODE

x = A[y z]

t_1 = y * 20
t_2 = t_1 + z
t_3 = t_2 * 4

(y*20+z)×4

t_4 = Base Address of A

x = $t_4[t_3]$

↳ Base offset Addressing
(Base Address + offset)

A: 10×20 –

A [4] [4]

00	01	02	03
10	11	12	13
20	21	22	23
30	31	32	33

Cross '2' rows ↑ ↱ cross '3' columns
A [2 3]
↳ No of elements in row
= 2*4+3
= 11

Row Major order: 00 01 02 03 10 11 12 13 20 21 22 23 30 31 32 33

column major order: 00 10 20 30 01 11 21 31 02 12 22 32 03 13 23 33

⇒ Run time Environment means when you run the program what is the support that you need from the operating system.

STORAGE ALLOCATION STRATEGIES

1) Static

1) Allocation is done at compile time
2) Bindings do not change at Runtime
3) One Activation Record per procedure

Disadvantages

1) Recursion is not supported.
2) Size of data objects must be known at compile time
3) Data Structures cannot be created Dynamically

2) Stack

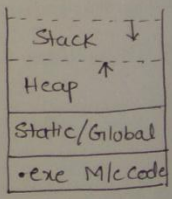
whenever a new activation begins, Activation record is pushed onto the Stack and whenever Activation ends, Activation record is popped off

⇒Local variables are bound to fresh storage

Disadvantages

1> Local variables cannot be retained once activation end

3) Heaps



⇒ Allocation and de allocation can be in any order

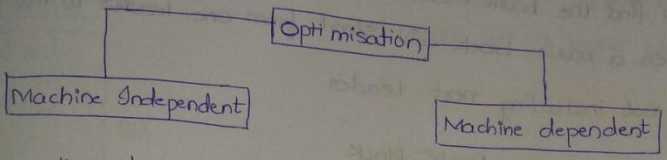
⇒ Disadv : Heap management is overhead

SUMMARY

Activations can have

1) permanent lifetime in case of static allocation
2) Nested lifetime in case of stack Allocation
3) Arbitrary lifetime incase of Heap Allocation.

INTRODUCTION TO CODE OPTIMISATION



1) Loop optimisations
   (a) code motion (or)
   Frequency reduction
   (b) Loop unrolling
   (c) Loop Jamming
2) folding
   - constant propogation
3) Redundancy Elimination
4) Strength Reduction

1) Register Allocation
2) use of Addressing modes
3) peephole optimisation
   (a) Redundant - load/store
   (b) Strength Reduction
   (c) flow of control options
   (d) use of M/c idioms

LOOP OPTIMISATION AND BASIC BLOCKS

→ To apply optimisations, we must first detect loops

→ For detecting loops we can use control flow Analysis (CFA) using program Flow Graph (PFG)

→ To find PFG, we need to find Basic blocks

A basic block is a sequence of 3-Address statements where controls enters at the begining and leaves at the end without any jumps or halts.

ALGORITHM TO FIND LEADERS

Finding the BASIC BLOCKS

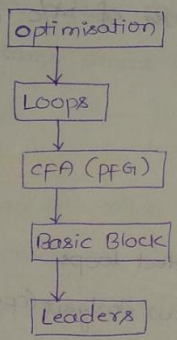
→ In order to find the basic blocks, we need to find the leaders in the program then a basic block will start from one leader to the next Leader but not including next Leader.

Identifying Leaders in the Basic Block

1) I Statement is a leader

2) Statement that is target of conditional or unconditional statement is a Leader → if() goto 200 (or) goto 300 → leader
   ↳ Leader

3) Statement that follows immediately a conditional or unconditional Statement is a Leader.



Example

```
fact(x)
{
    int f=1
    for(i=2;i<=x;i++)
    f=f*i;
    return f;
}
```

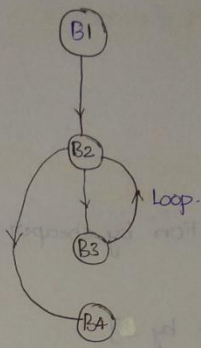
1) i = 1
2) i = 2 — B1

3) If (i > x) goto 9 — B2

4) t_1 = f * i ;
5) t_2 = i + 1 ;
7) i = t_2 ;
8) goto (3) — B3

9) Goto calling program — B4

⇒ Incase you have 4 leaders we will get 4 Basic Blocks ⇒ If you have 'n' basic Leaders will get 'n' basic blocks



TYPES OF LOOP OPTIMIZATION

Frequency Reduction

Moving the code from high frequency region to low frequency Region is called code motion.

Ex: while (i<5000)
{
A = Sin(x)/cos(x) * i;
i++;
}

⇓

t = Sinx/cos x
while (i<5000)
A = t * i;

Loop unrolling

while (i<10)
{
x[i] = 0
i++
}

⇓

while (i<10)
{
x[i] = 0;
i++;
x[i] = 0;
i++;
}

Loop jamming

combines the bodies of two loops

for(i=0; i<10; i++)
for(j=0; j<10; j++)
x[i,j] = 0;
for(i=0; i<10; i++)
x[i,i] = 0

⇓

for(i=0, i<10; i++)
{
for(j=0; j<10; j++)
{
x[i,j] = 0;
} x[i,j] = 0;
}

Folding:

Replacing an expression that can be computed at compile time by its value.

Ex: $2+3+C+B = 5+C+B$

Redundancy Elimination (DAG)

$A = B+C$

$D = 2+B+3+C$

$D = 2+3+A$

Strength Reduction

Replacing a costly operation by cheaper one

Ex: $B = A * 2$

$B = A \ll 1$ [Left shift by 1]

Algebraic Simplification

$A = A+0$
$x = x*1$
} eliminate such statements

MACHINE DEPENDENT OPTIMISATION

1) Register Allocation
   - Local Allocation
   - Global Allocation

2) Use of Addressing modes

3) peephole optimisation

a) Redundant Load and store elimination

$x = y+z$

```
Mov y,R0
Add z,R0
Mov R0,x
```

```
a = b+c | Mov b,R0
d = a+e | Add c,R0
        | Mov R0,a   X
        | Mov a,R0   X
        | Add e,R0
        | Mov R0,d
```

(b) Flow control optimisation

Avoid jumps on jumps

```
L1: Jump L2        (L4)
L2: Jump L3
L3: Jump L4
```

Eliminate dead code

```
#define x 0
if (x)
{
    dead code ×
}
```

d) Use of M/c idioms

```
i = i+1     MOV  R0, i
            add  R0, 1      } increment 'i'  [inc i]
            mov  i, R0
```

⇒ Handle of the string is a substring that matches with RHS of production

⇒ RR conflicts occur in LALR(1) parser when merging of the states

⇒ SR conflicts doesnot occur in LALR(1) parser

⇒ If the attribute can be evaluated in Depth-first-order then the attribute is L-attributed.