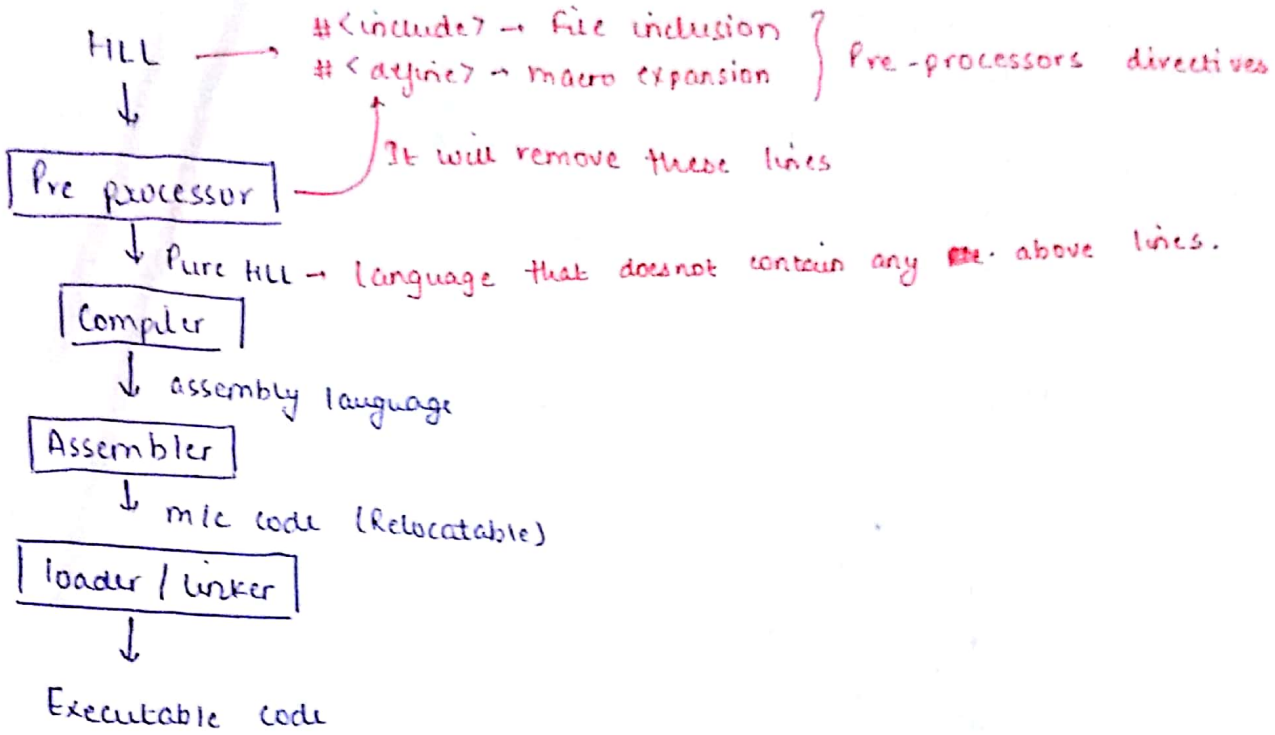
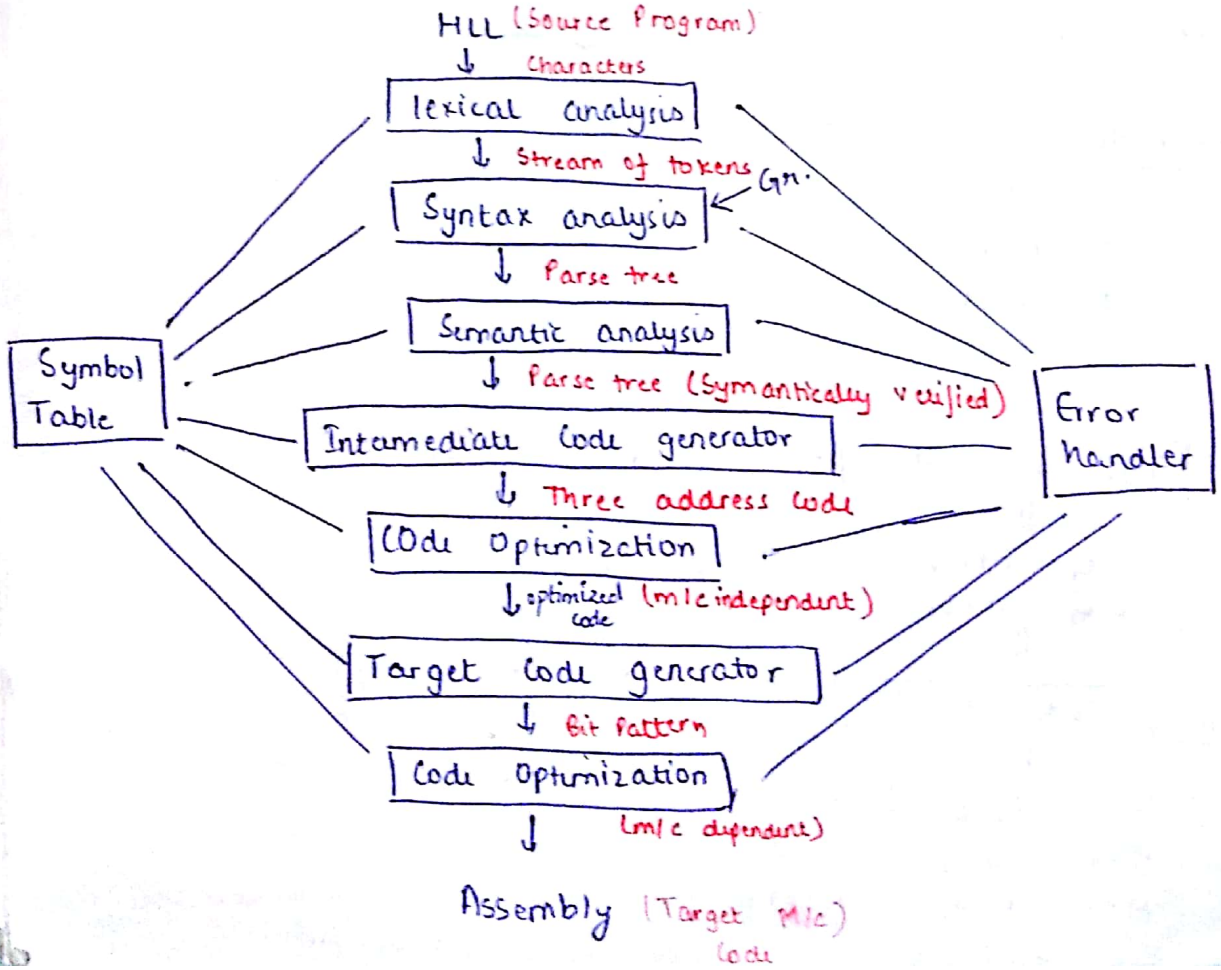
Compiler Construction

oduction and various phases of compiler

converts HLL to LLL



→ Phases of compiler



I/P : $x = a + b * c;$ [character or lexemes]

⇓

lexical analyser

⇓

id = id + id * id [Stream of tokens).

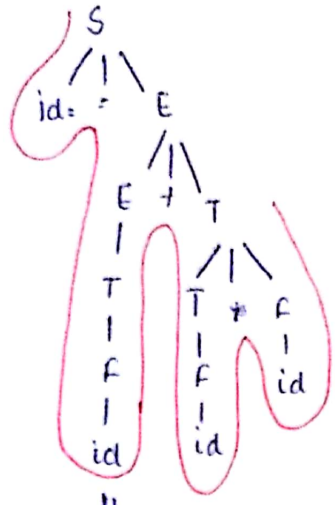
⇓

Syntax Analyser ⇐

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

E = Expr
T = Ter
F = Fact

Parse tree



⇓

Semantic Analyser → Type checking

⇓

(Parse tree semantically)

⇓

Intermediate code generator

⇓

$t_1 := b * c;$
$t_2 := a + t_1;$
$x := t_2;$

⇓

Code Optimization → To reduce no. of lines

⇓

$t_1 := b * c;$
$x := a + t_1;$

⇓

Target Code Generator

⇓

Multiplying R_1 & R_2 & storing it into R_2

Mul	R_1, R_2	$a \rightarrow R_0$	That assembler
Add	R_0, R_2	$b \rightarrow R_1$	understands.
Mov	R_2, x	$c \rightarrow R_2$	

E → E+F
/F*E
/F

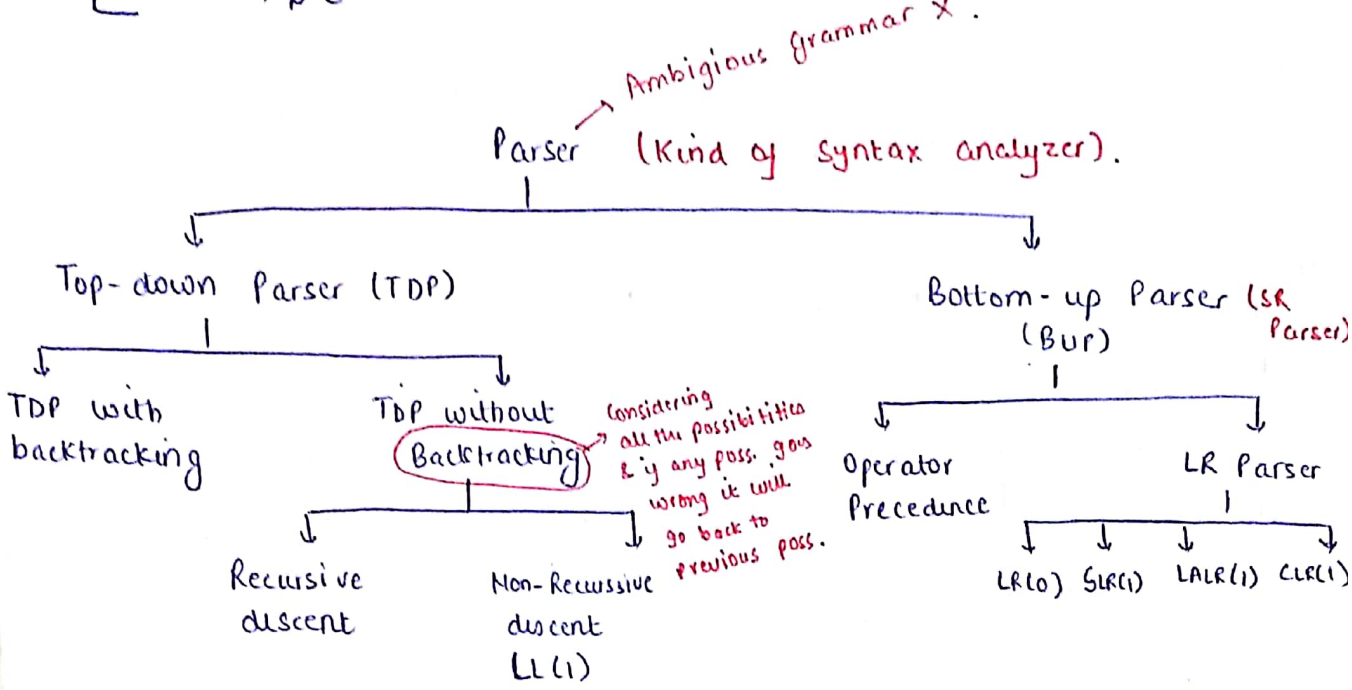
* -> * (Defined left associative)
+ & + (Defined right associative)

F → F-F
/id

left Recursion

$A \rightarrow A\alpha / \beta$

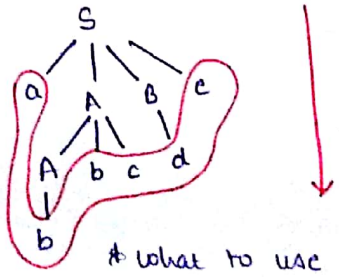
$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' / \beta \in$



* Without Backtracking → generate the string exactly what is required.
↳ Left Recursion X
↳ ND X (Non- deterministic).

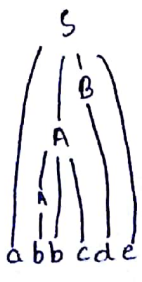
Eg,

S → aABe
A → Abc / b
B → d
ω: abbcde



* what to use

Eg,



* whom to redu

* First & follow

	First	Follow
① S → ABCDE	{a,ϵ}-ϵ ∪ {b,ϵ}-ϵ ∪ {c} = {a,b,c}	{$}
A → a/ϵ	{a,ϵ}	{b,ϵ}-ϵ ∪ {$} = {b,
B → b/ϵ	{b,ϵ}	{c}
C → c	{c}	{d,ϵ}-ϵ ∪ {$} = {d,$
D → d/ϵ	{d,ϵ}	{e,ϵ}-ϵ ∪ {$} = {e,$
E → e/ϵ	{e,ϵ}	{$}

②

S → Bb/Cd	{a,b} ∪ {c,d} = {a,b,c,d}	{$}
B → aB/ϵ	{a,ϵ}	{b}
C → cC/ϵ	{c,ϵ}	{d}

③

E → TE'	{id,(}	{$,)}
E' → +TE'/ϵ	{+,ϵ}	{$,)}
T → FT'	{id,(}	{+,ϵ}-ϵ ∪ {$,)} = {+,$,)}
T' → *FT'/ϵ	{*,ϵ}	{+,$,)}
F → id / (E)	{id,(}	{*,ϵ}-ϵ ∪ {+,$,)} = {*,+,$,)}

④

S → ACB | CbB | Ba	{d,g,h,ϵ,b,a}	{$}
A → da / BC	{d,g,h,ϵ}	{h,ϵ}-ϵ ∪ {g,$} , {h,g,$
B → g/ϵ	{g,ϵ}	{$,a,g,h}
C → h/ϵ	{h,ϵ}	{b,$,g,h}

truction of LL(1) Parsing Table

(3),

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

→ Recursive Descent Parser

E → iE'

E' → +iE' / ϵ

```
E()
{  if (l=='i')
   {
     match ('i');
      E'();
   }
}

E'()
{  if (l=='+')
   {  match ('+');
      match ('i');
       E'();
   }
   else
     return;
}
```

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

main()
{  E();
   if (l=='$')
     printf ("Parsing successful");
   }
```

→ Operator Precedence Parser

* Operator grammar

$E \rightarrow E+E / E*E / id$

No two operands should be adjacent.

Eg, $E \rightarrow EAE / id$
$A \rightarrow + / *$] X

	id	+	*	$
id	-	·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	-

④ '$' & '$' → Not compared.

⑤ '+' and '+' always ·> because it is left associative.

⑥ '*' & '*' always ·> ___

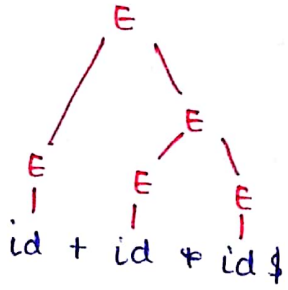
⑦ '↑' & '↑' always <· because right associative.

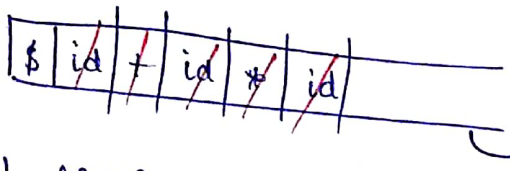
⑧ 'id' will not be compared with 'id'.

② "id" will have high precedence than any other operator. & vice-versa.

③ '$' will always have less precedence than any other operator.

* less → Push
More → Pop.





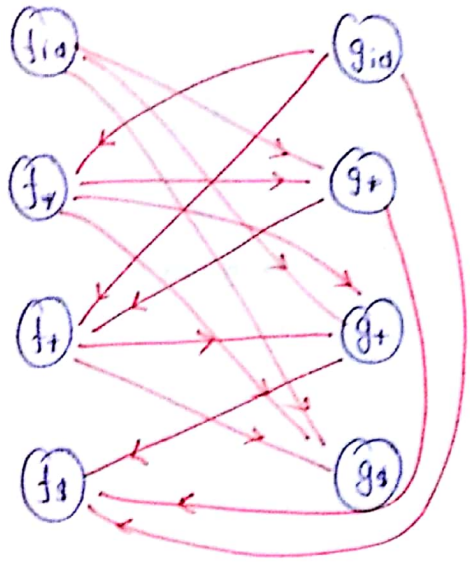
→ Stack

* we will compare stack with string always. and if the precedence is low we will push otherwise pop.

...dion precedence Parser

	id	+	*	$	]g
id	-	⋗	⋗	⋗	
+	⋖	⋗	⋖	⋗	
*	⋖	⋗	⋗	⋗	
$	⋖	⋖	⋖	-	

→ This basically shows the arrow's direction.



$f_{id} \to g_+ \to f_+ \to g_* \to f_\$$

$g_{id} \to f_+ \to g_\$ \to f_+ \to g_+$

f_+

$f_\$$

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

- function Table.

* If there is a loop we will stop finding the path.

(2)

P → SR / S → (1)
R → bSR / bS → (2)
S → wbS / w
w → L * w / L
L → id

The above grammar is not an operator grammar as 'SR' is adjacent. So, now we will make it operator grammar.

P → SbSR / SbS / S [From (2)]

Again adjacent then,

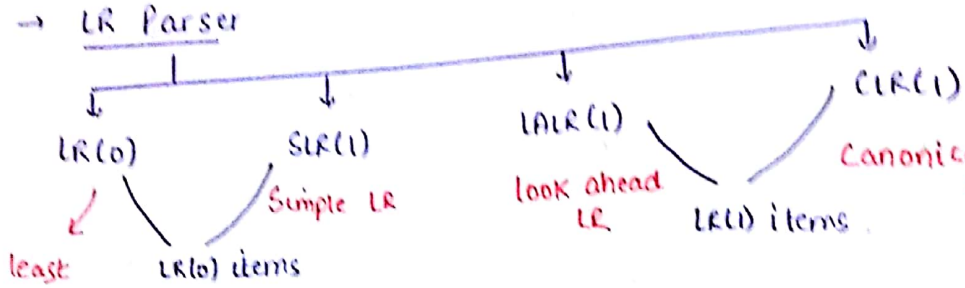
P → SbP / SbS / S
S → wbS / w
w → L * w / L
L → id

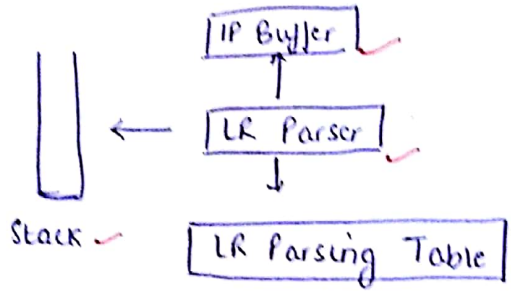
} Now, it is an operator grammar.

Now, constructing operator precedence table.

	id	*	b	$
id	-	⋗	⋗	⋗
*	⋖	⋖	⋗	⋗
b	⋖	⋖	⋖	⋗
$	⋖	⋖	⋖	-

→ LR Parser





* LR Parser is basically an algorithm where it will take I/P buffer and there will be Stack for all 4. But the difference will be only in the construction of "LR Parsing Table".

* Construction of LR(0) construction table

The given grammar is,

S → AA

A → aA/b.

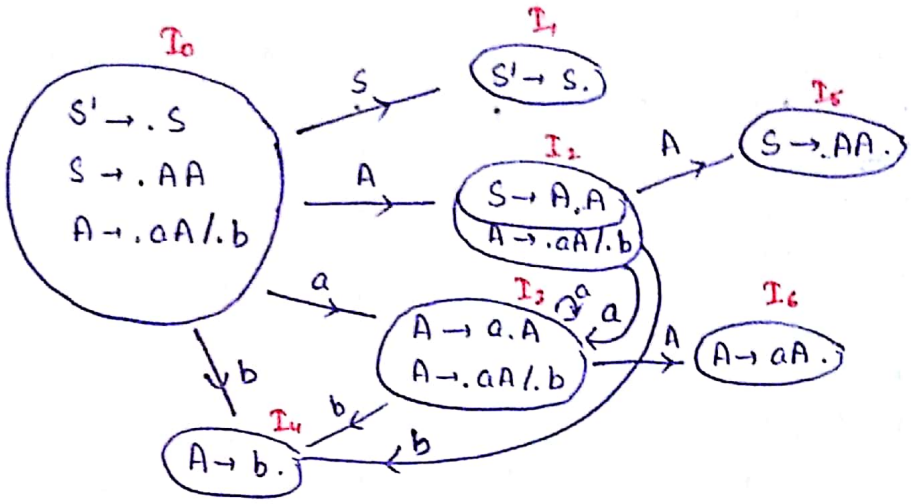
Now, we will add augmented State,

S' → S

S → AA

A → aA/b.

finding the goto of productions,



goto (I_0, S)
{ S' → S. } I_1

goto (I_0, A)
{ S → A.A
A → .aA / .b } I_2

goto (I_0, a)
{ A → a.A
A → . aA / .b } I_3

goto (I_0, b)
{ A → b. } I_4

goto (I_2, a)
{ A → a.A
A → . aA / .b } I_3

goto (I_2, b)
{ A → b. } I_4

goto (I_2, A)
{ S → AA. } I_5

goto (I_3, A)
{ A → aA. } I_6

S → AA ①
A → aA / b
② ③

* LR(0) Parsing Table

Action Table | Goto Table

	a	b	$	A	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		

Shift.
Terminal going to which state will be the $S_=$

Reduce
We will find this when '.' (dot) is at the end of production (A → b.) then we will notice that in which state it is and what is the no. of that production. $R_=$

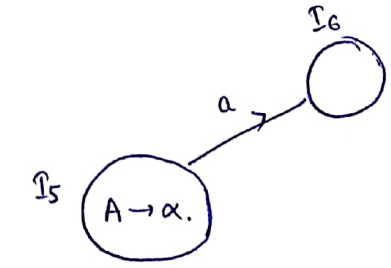
* SLR(1) Passing Table for the previous one.

	Action			Goto	
	a	b	$	A	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		
6	r_2	r_2	r_2		

* Shift is fou... Same procedu...
* For Reduce in... we will find ... where the "." (do... at the end then ... will check the no. ... production which ... for R_. But we ... fill R_ in that block ... only where that production's left side non-terminal follows. (1)

* SLR(1) is more powerful than LR(0) because this has less no. of Reduce nodes so we this ~~will~~ to find the error quickly and at earlier stage

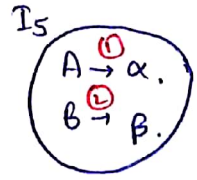
* LR(0) is invalid when there is SR or RR conflict. (Shift- Reduce or Reduce- Reduce conflict).



	a	b
5	S_6/R_1	R_1

SR Conflict

This happens when a state is going to another state i.e. goto (I_5, a) and I_5 contains the final state $(A \to \alpha.)$ ↙ dot at the end.

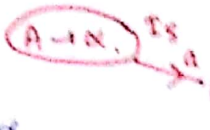


	a	b	c
5	r_1/r_2	r_1/r_2	r_1/r_2

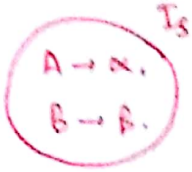
RR Conflict

This happens when there is two final production in one State.

is invalid when the follow(A) ∩ [a] ⇒ SR conflict.

↳ incase of 

follow(A) ∩ follow(B) ≠ ∅.

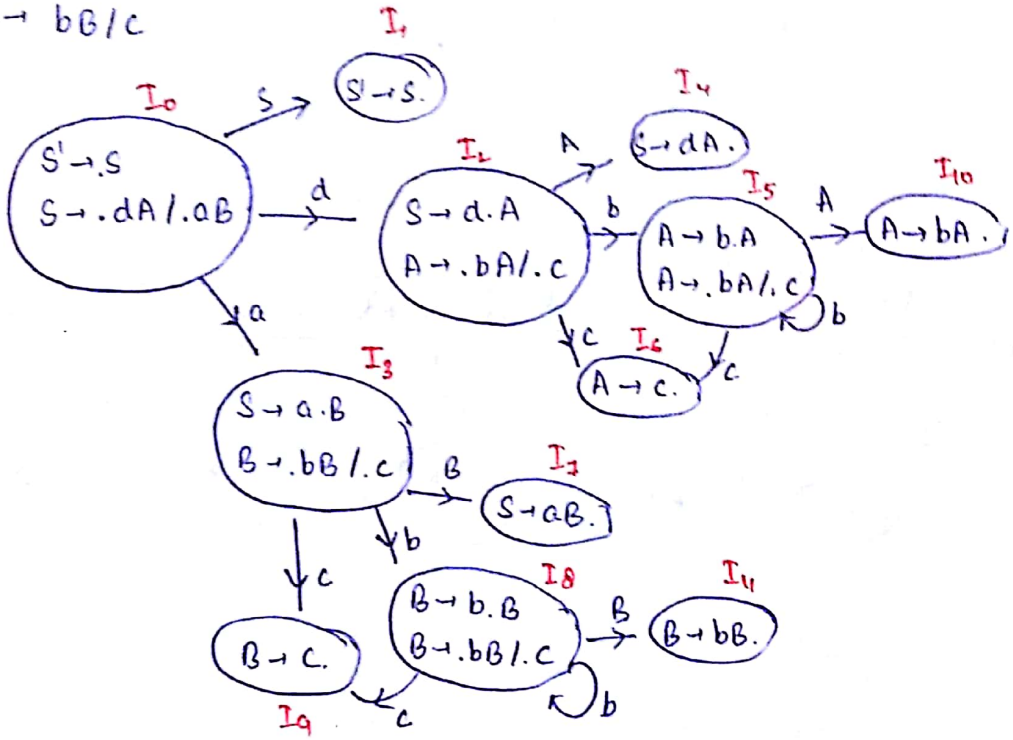
↳ in case of 

Eg,

① $S \rightarrow dA / aB$

$A \rightarrow bA / c$

$B \rightarrow bB / c$



② * Whether this grammar is LL(1) or not.

↳ Yes, it is because first(S) = [d, a]
first(A) = [b, c]
first(B) = [b, c]
} There no repitition of first's.

① * LR(0) or not

↳ It's a LR(0) because there is no SR or RR conflict in all the final states.

* SLR(1) or not

↳ It is SLR(1) because it is LR(0) & if a grammar is LR(0) so it will not any conflicts by itself.

R(1) and CLR(1)

LR(1) item = LR(0) items + look ahead

Eg, S → .aA , a/b

NOTE

* For look ahead,

  A → α.Bβ , a/b

  B → .γ , ↓ First of the remaining after B.

  ① By which production it comes from.

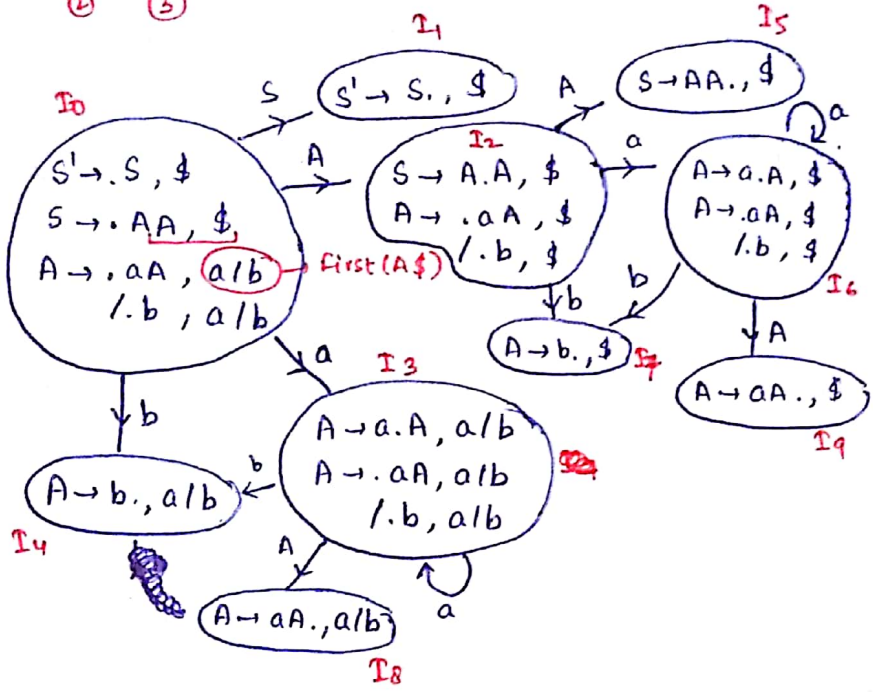
  ② After that, the first of the remaining will be the look ahead.

* While applying the transition look ahead will not change
* While applying closure LA may change

→ Shifting the dot

① S → AA ①

A → aA / b ② ③



→ CLR(1) Parsing Table

	a	b	$	S	A
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		

① I_3, I_6 → I_{36}

② I_4, I_7 → I_{47}

③ I_8, I_9 → I_{89}

Same productions, but diff. LA.

CLR(1) & LR(0) = SLR(1)
↳ = LALR(1)
} No. of states.

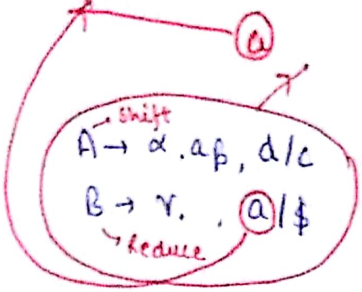
→ For LALR(1) Parsing Table,

	a	b	$	S	A
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		
~~36~~	~~S_{36}~~	~~S_{47}~~			89 (Merged)
~~47~~			~~R_3~~		(Merged)
89	R_2	R_2	R_2		
~~89~~			~~R_2~~		(Merged).

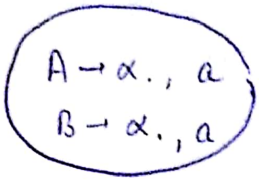
LALR(1) Parsing Table,

	a	b	$	S	A
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		

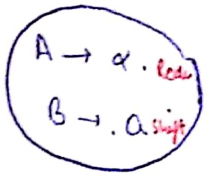
SR conflict

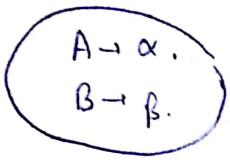


RR conflict



erns



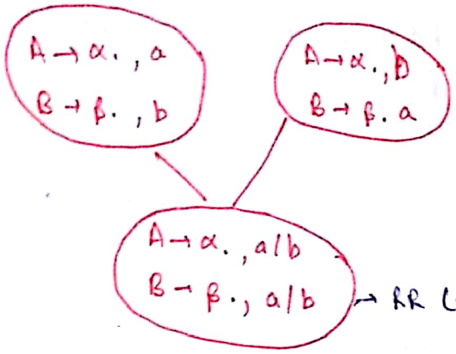


CLR(1) ✓ then LALR(1) may or may not contain

* If CLR(1) X then LALR(1) X for sure

lict,

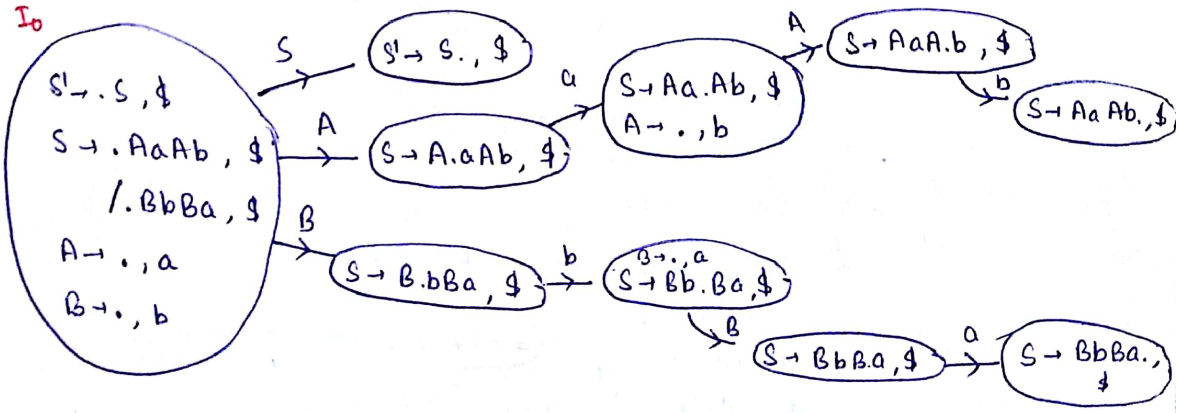
Eg,



→ RR conflict ⇒ LALR(1) X.

S → AaAb / BbBa
A → ϵ
B → ϵ

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



→ Syntax Directed Translation (SDT)

Grammar + Semantic action / Rules = SDT

↳ Transforming the tokens into structure. / tree.

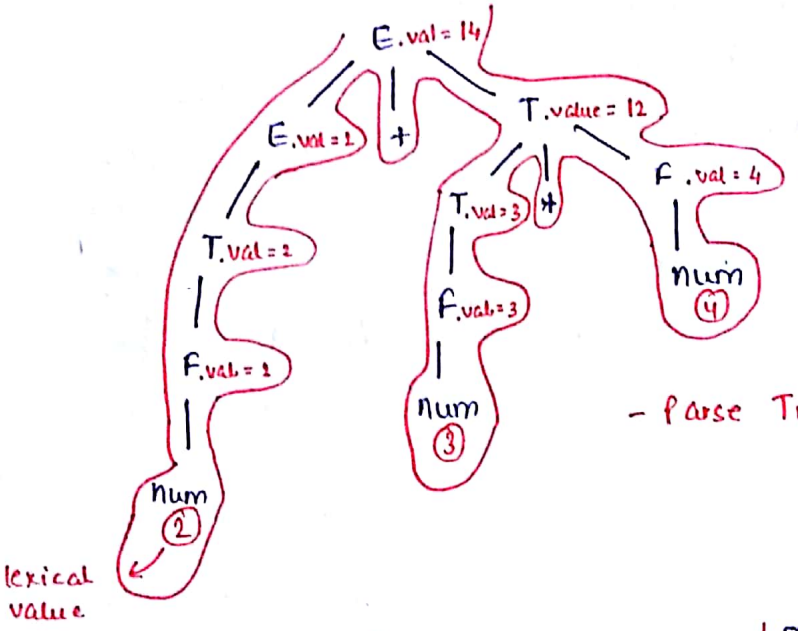
SDT for evaluation of exp.

① $E \to E+T$ { E.value = E.value + T.value }
/T { E.value = T.value }

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

$F \to num$ { F.value = num.lvalue }
↳ lexical value.

w = 2 + 3 * 4



- Parse Tree.

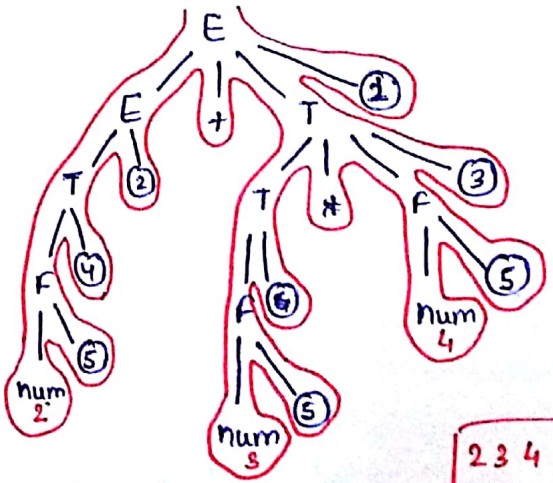
Top - bottom - left - Right.

② SDT, ① → Semantic action.

$E \to E+T$ { Printf ("+"); }
/T { } ②

$T \to T*F$ { Printf ("*"); } ③
/F { } ④

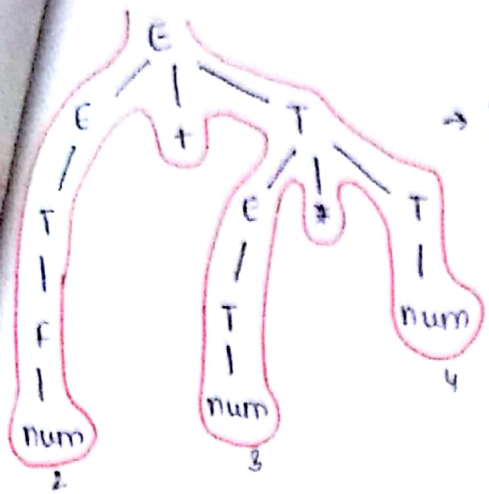
$F \to num$ { Printf (num.val); } ⑤



Top - down

2 3 4 * +
↳ Postfix exp. from infix.

2 + 3 * 4



→ This is how bottom-up works.

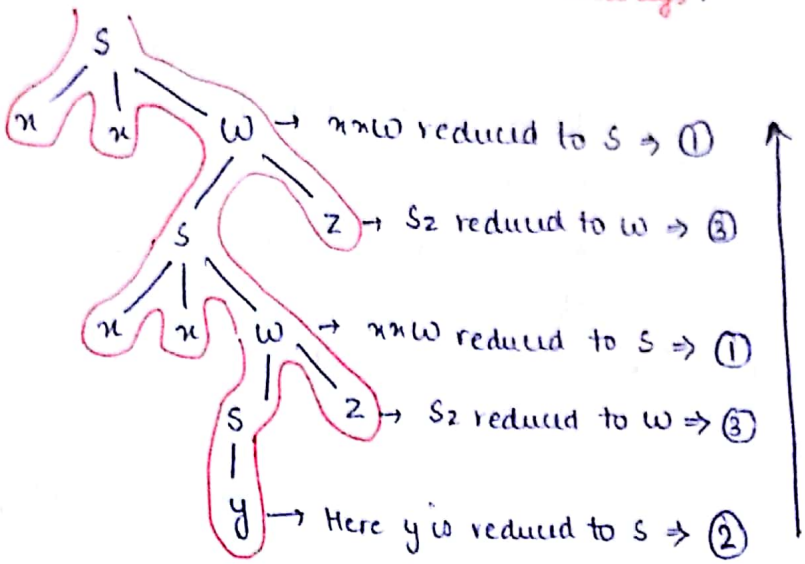
(2 3 4 * +) Postfix.

③

S → x x w { Printf (1); }
/ y { Printf (2); }
w → S z { Printf (3); }

w: x x x x y z z

* SDT's semantic actions performed on reduction always.



Output,

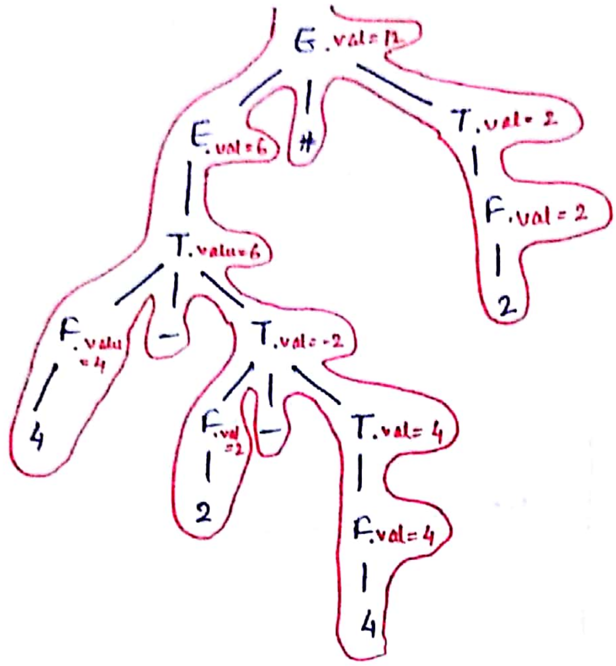
2 3 1 3 1

② ④ E → E * T { E.val = E.val * T.val }
/ T { E.val = T.val }

① T → F − T { T.val = F.val − T.val }
/ F { T.val = F.val }

F → 2 { F.val = 2 }
/ 4 { F.val = 4 }

$w = 4 - 2 - 4 * 2$ $\Rightarrow$ $((4 - (2 - 4)) * 2)$



④

E → E # T { E.val = E.val * T.val ; }
/T { E.val = T.val }
T → T & F { T.val = T.val + F.val }
/F { T.val = F.val }
F → num { F.val = num.lval }

| w = 2 # 3 & 5 # 6 & 4 |

⇓

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

⇓

((2 * 8) * 10)

16 * 10

= 160

to build Syntax Tree

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

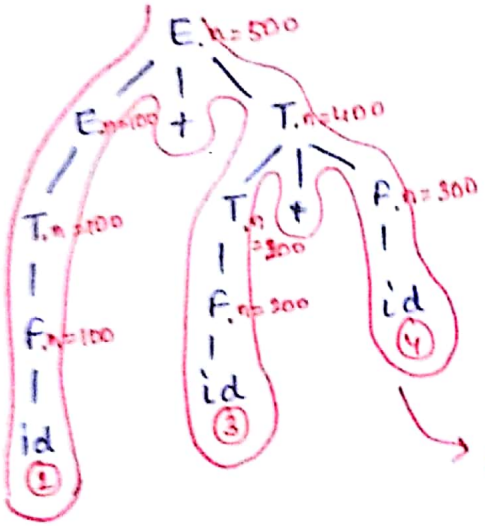
/ T { E.nptr = T.nptr ; }

T → T * F { T.nptr = Tmknode (T.nptr , '*' , F.nptr); }

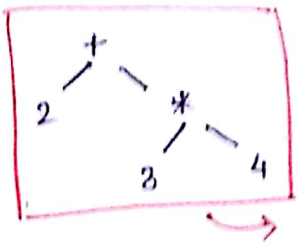
/ F { T.nptr = F.nptr ; }

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

w = 2 + 3 * 4



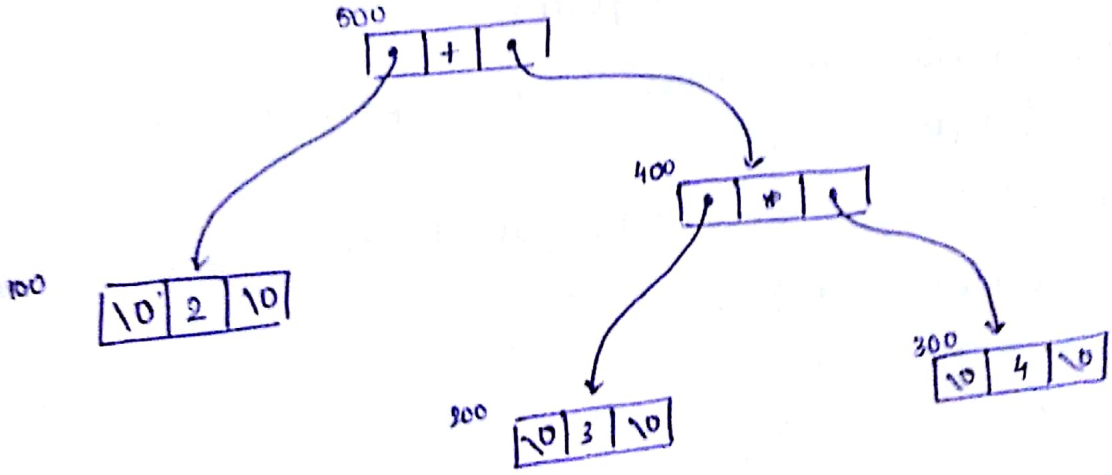
Concrete parse tree.
(Syntax tree)



Abstract parse tree
(Syntax tree).

⇓

Parse tree where variables are not shown.



(2) Here both are E but to distinguish we are writing E_1 & E_2

Type Checking

$E \to E_1 + E_2$ { if ((E_1.type == E_2.type) && (E_1.type = int)) then E.type = int

/ $E_1 == E_2$ { if ((E_1.type == E_2.type) && (E_1.type = int / bool)) then E_1.type = boolean

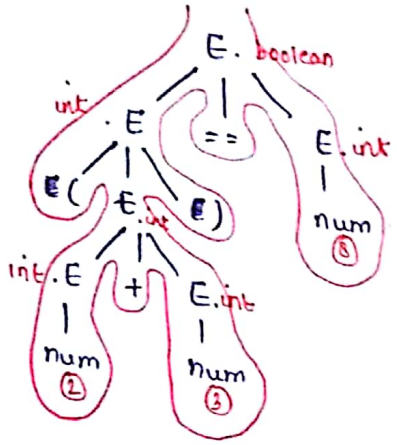
/ (E) { E.type = E_1.type }

/ num { E.type = int; }

/ True { E.type = boolean; }

/ False { E.type = boolean; }

w = (2+3) == 8



→ SDT to generate three address code

$S \to id = E$ { gen (id.name = E.place); }

$E \to E_1 + T$ { E.place = new Temp(); gen (E.place = E_1.place + T.place); }

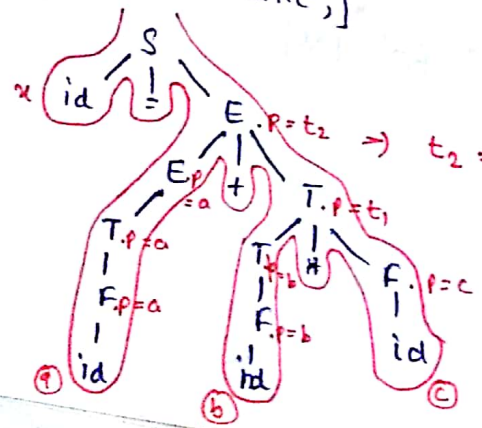
/ T { E.place = T.place; }

$T \to T_1 * F$ { T.place = new temp(); gen (T.place = T.place * F.place); }

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

$F \to id$ { F.place = id.name; }

w: x = a+b*c



⇒ $t_2 = a + t_1$

$t_1 = b * c$

$t_2 = a + t_1$

$x = t_2$

Three address code for $x = a + b * c$.

Synthesized attributes

When a node takes the value from it's children then it is known as synthesized.

Inherited attributes

When a node takes the value from its parent or from its siblings then it is inherited attribute.

Eg, $A \rightarrow BCD$

$A.s = f(B.s, C.s, D.s)$ → Synthesized

$C.i = A.i$ → from parent.
$C.i = B.i$ → Inherited.
$C.i = D.i$ → from siblings

S-attributed SDT

① Uses only synthesized att.

② Semantic actions are placed at right end of production

$A \rightarrow BCC\{\}$

* Postfix SDT

③ Attributed are evaluated during Bottom-up parsing.

* Post order traversing.

L-attributed SDT

① Uses both synthesized as well as inherited att. But each inherited att. is restricted to inherit from either parent or from left sibling only.

Eg, $A \rightarrow XYZ$

$Y.S = A.S$ ✓ ; $Y.S = X.S$ ✓ , $Y.S = Z.S$ ✗

② Semantic actions are placed anywhere on RHS.

$A \rightarrow L\{\}BC$
$/ D L\{\} B$
$/ FG\{\}$

③ Attributes are evaluated by traversing parse tree depth first, left, to right.

Pre order traversing

→ Syntax Directed Translation (SDT)

As we know that after syntax analysis, the semantic of input program is needed.

"Syntax Directed Translation Scheme" is the process of the semantic actions to be attached to the context free grammar productions.

Two notations are used to associate the semantic rules to the production.

① Syntax Directed Definition (SDD)

② Translation Scheme.

① SDD is generalization of the context free grammar in which each grammar symbol either terminal or non-terminal has associated set of attributes. / or is
↳ string, number, type, etc.

associated with the set of semantic rules of the form. $A \to \alpha$

$$x := f(a_1, a_2 \ldots a_k)$$

attributes are associated with the grammar symbol i.e. A

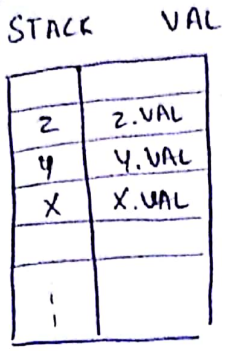
* Rules are associated with the production i.e α.

* Annotated Parse Tree

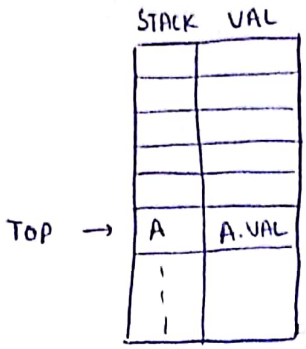
A parse tree containing the values of attributes at each node of the tree.

* SDT can be implemented, by using extra fields in the parser stack entries corresponding to grammar symbol. So, Stack is implemented by two pair of arrays.

STACK & VAL.



Before reduction

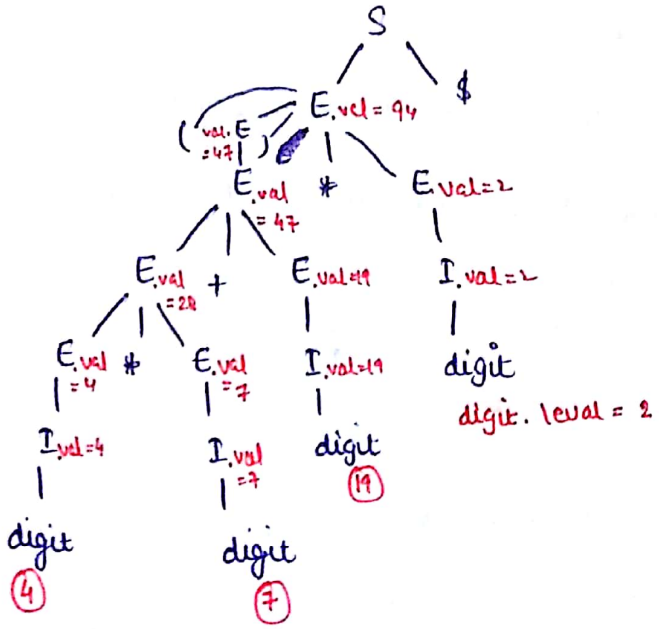


After reduction

Eg,

$S \to E\$$	{ Printf E.val }
$E \to E^{(1)} + E^{(2)}$	{ E.val = $E^{(1)}$.val + $E^{(2)}$.val }
$E \to E^{(1)} * E^{(2)}$	{ E.val = $E^{(1)}$.val * $E^{(2)}$.val }
$E \to (E^{(1)})$	{ E.val = $E^{(1)}$.val }
$E \to I$	{ E.val = I.val }
$I \to I^{(1)}$ digit	{ I.val = 10 × $I^{(1)}$.val + lex val }
$I \to$ digit	{ I.val = lex val }

(4*7+19)*2$



→ Intermediate Code Generation

Intermediate code → In most of the compilers the source code is translated to a language which is in complexity b/w HLL & machine code such code is called intermediate code.

Most common representation of source code are

1. Postfix notation
2. Syntax trees
3. Quad triples
4. Triples
5. Indirect triple

① Postfix Notation

a) $e_1 \oplus e_2 \Rightarrow e_1 e_2 \oplus$

b) $(a+b) * c \Rightarrow ab+c*$

c) If 'e' then 'x' else 'y' by using ? ⇒ exy?

d) If a then if c-d then a+c else a*c else a+b

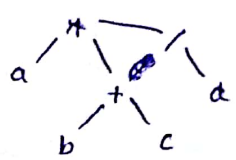
acd-ac+ac*?ab+?

→ This ternary operator will come after every else statement.

② Syntax Tree

It is a variant to parse tree in which each leaf represent an operand and each interior node represents an operator.

Eg, a*(b+c)/d ,

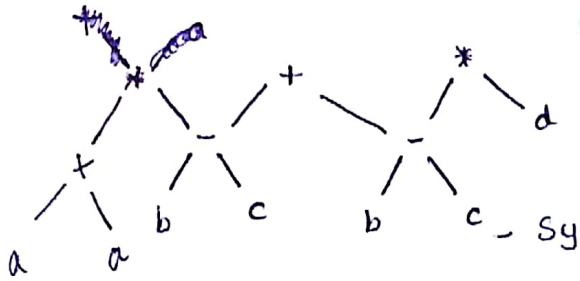


ed Acyclic Graph (DAG)

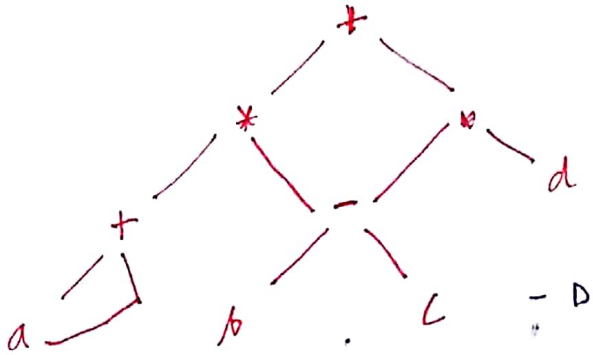
similar to Syntax tree having leaves corresponding to ds and interior node representing the operators.

ence b/w DAG & Syntax tree is that a node N in a DAG more than one parent if N represents a common uberpression in a Syntax tree. means DAG represents he generation of efficient code to evaluate the expression.

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



– Syntax Tree



– DAG.

→ Three address code (at most 3 operand per instruction and one operator in RHS)

It is a linearized representation of a syntax tree or a DAG in which explicit names corresponding to the interior nodes of the graph are used.

It is a sequence of statments of the form,

A = B op C

↓ 2 operands

Result

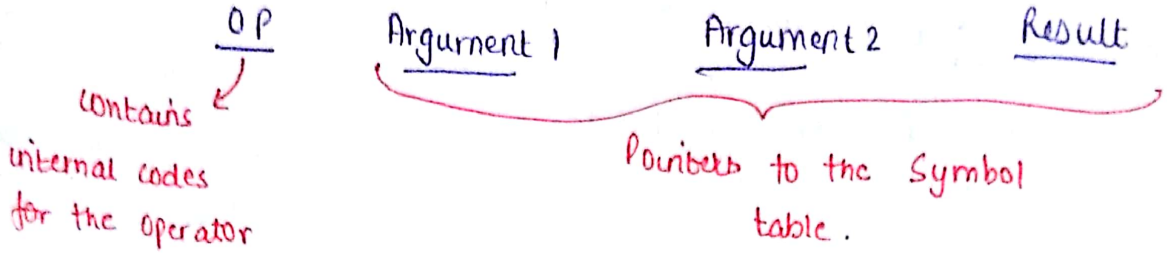
Three address code.

→ Implementation of three-address code

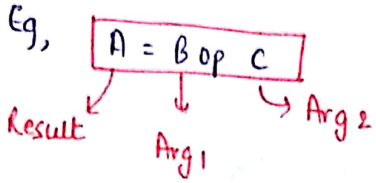
3 address code is an abstract form of intermediate. These statements can be implemented as record for the operator & operands.

① Quadruples

It is a record structure using 4 fields.



Eg,



Eg,

① $A := -B * (C+D)$

$T_1 := -B$
$T_2 := C+D$
$T_3 := T_1 * T_2$
$A := T_3$

} Three-address code

Now, Quadruple table,

	OP	Arg 1	Arg 2	Result
(0)	unary minus	B	-	T_1
(1)	+	C	D	T_2
(2)	*	T_1	T_2	T_3
(3)	:=	T_3		A

Quadruples

avoid entering temporary names into the symbol table, temporary value can be referred by the position of the statement that computes it.

So, in this, the three-address statements can be represented by record structure with only three fields.

OP	Arg1	Arg2

These are either pointers to the symbol table or into the structure itself.

Eg, A := - B * (C+D)

$T_1 := -B$

$T_2 := C+D$

$T_3 = T_1 * T_2$

$A = T_3$

	OP	Arg 1	Arg 2
(0)	U minus	T_1	-
(1)	+	C	D
(2)	*	(0)	(1)
(3)	:=	A	(2)

→ Indirect Triples

Consist of listing of pointers to triples rather than a listing of triples themselves in desired order.

Eg, A := - B * (C+D)

	Statement
(0)	(9)
(1)	(10)
(2)	(11)
(3)	(12)

	OP	Arg 1	Arg 2
(9)	Un minus	B	-
(10)	+	C	D
(11)	*	(9)	(10)
(12)	:=	A	(11)

→ Some examples of generating three-address codes

① Assignment statement of integer type

Eg, $a = b+c+d$

⇒ $t_1 := b+c$

$t_2 := t_1+d$

$a := t_2$

② Assignment statement with mixed type

Eg, ① $X = Y + I \# J$

Where,

X and Y → Real

I and J → integer

⇒ $T_1 := I \# J$

$T_2 := \text{int to Real}(T_1)$

$T_3 := Y + T_2$

$X := T_3$

b) $X := Y * Z + Y * r$

Where,

r and y are integers

x and z are float

$T_1 := \text{int to float } Y$

$T_2 := \text{int to float } r$

$T_3 := T_1 * T_2$

$T_4 := T_1 * Z$

$T_5 := T_3 + T_4$

$X := T_5$

③ Boolean expression

For generating the three address code for boolean exp., Some branching statements of the following form are used

(Unconditional Jump statement)

① → goto L → means three address code statement with label L is next to be executed.

② → if A goto L

↳ Conditional Jump Statement

(means if A is non-zero / true then statement with label L is executed otherwise next statement)

③ → if A relop B goto L

↳ means if A relop B is true statement with label L is executed else the next statement in the normal flow sequence is executed.

3,

① If A < B then 1 else 0

Three address code is given as,

```
If A<B then L1
   T=0
   goto L2
L1: T=1
L2:
```

OR

```
(1) if A<B goto (4)
(2) T=0
(3) goto (5)
(4) T=1.
```

② If A<B or C then 1 else 0.

```
(1) if A<B goto (4)
(2) T1 = 0
(3) goto (5)
(4) T1 = 1
(5) T2 := T1 or C
```

③ If a>b then x = a+b.

```
(1) if a>b goto (3)
(2) goto (5)
(3) T1 := a+b
(4) X := T1
(5)
```

OR

```
if a>b goto L1
goto L2
L1: T1 := a+b
    x := T1
L2:
```

→ Code Optimization

This refers to techniques which is used to improve the of the generated object code by compiler.

It includes various transformations on intermediate representation of the program where the these transformations must preserve the semantics of the program.

* Features

① Attempt from compiler to generate better object code.
② Code improvement
③ Replacing pattern by equivalent & more efficient constraints.
   ↳ may be local or global.
   ↳ may be machine dependent / independent.
④ Inner loop optimization (90% - 10% Rule)
⑤ Constant folding
⑥ Elimination of common sub-expression
⑦ Elimination of loop-invariant
⑧ Elimination of induction variables.

There are mainly 3 interrelated areas of code optimization,

① local optimization
② loop optimization
③ Data flow analysis.

① which is performed in a straight line (basic block of code means a block with no jump except at the beginning & no jumps except at the end.

③ Transmission of useful from all parts of the program to the places where the information can be of use.

Blocks and flow graph

raph representation of three address statement is called
graph where the nodes in flow graph represents the
mputation and the edges represents the flow of control.
low graph can be used to collect information about
the intermediate code. and to find the inner loop where
a program is expected to spend most of its time.

Basic Block

It is a sequence of consecutive statements in which

a) flow of control can only enters the block in the beginning.
No jumps are allowed in the middle of the block.

b) flow of control can leave the block at the end without
halt on possibilities of branching ex

Steps to construct basic block

① We first determine the set of leaders. Rules are

a) The first statement is a leader.

b) Any statement which is a target statement of conditional
or unconditional goto statement is a leader.

c) Any statement which immediately follows the conditional
goto is a leader.

② From one leader statement to the just prior statement
of the next leader (if no such leader statement is present then
upto the last statement) will form a basic block.

* If there is any statement which has not been included to any
basic block can be removed & discarded.

Eg,
①

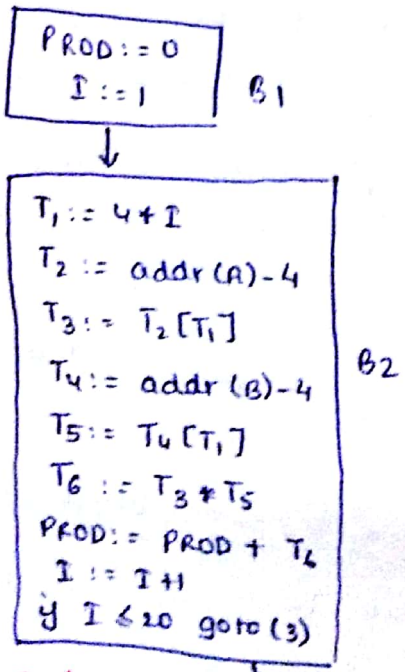
```
Begin
    PROD := 0;
    I := 1;
    do
    begin
      PROD := PROD + A[I] * B[I];
      I := I+1;
    end
    while I ≤ 20.
end
```

- Program Segment

```
(1)   PROD := 0
(2)   I := 1
```
B_1

```
(3)   T1 := 4 * I
(4)   T2 := addr(A) - 4
(5)   T3 := T2[T1]
(6)   T4 := addr(B) - 4
(7)   T5 := T4[T1]
(8)   T6 := T3 * T5
(9)   PROD := PROD + T6
(10)  I := I+1
(11)  if I ≤ 20 goto (3)
```
B_2

- Three address code for a machine with 4 bytes / word. (BASIC BLOCK).



- FLOW GRAPH

* The relationship among basic block is represented

→ To block begining with the statement following statement (11)

```
Sum = 0;
i = 1;
while  i<=10  do
{
   Sum =  Sum + a[2*i];
   i = i+1;
}
 average = Sum/i;
```

TAC is ⇒

```
(1)  Sum := 0
(2)  i := 1
(3)  if i > 10 then goto (11)
(4)  T1 := 2 * i
(5)  T2 := a[T1]
(6)  T3 := Sum + T2
(7)  Sum := T3
(8)  T4 := i+1
(9)  i := T4
(10) goto (3)
(11) T5 := Sum/i
(12) average := T5
```

B1: (1), (2)
B2: (3)
B3: (4) to (10)
B4: (11), (12)

→ local Optimization

A basic block computes of a set of expressions. Two basic blocks are said to be equivalent if they computes the same set of expressions.

Optimization can be achieved by applying no. of transformation with in each blocks. Two classes of local transformation can be applied to basic blocks.

① Structure preserving transformation (common subexpression elimination)
② Algebric transformation.

① Structure Preserving Optimization

A) Common sub-expression elimination

Eg, consider the following basic blocks.

1) $a = b+c$		1) $a = b+c$	Transformed block as 2 statements with the same exp.
2) $b = a+d$	$\Rightarrow$	2) $b = a+d$	
3) $c = b+c$		3) $c = b+c$	
4) $d = a+d$		4) $d = b$	

* But 1st and 3rd statements are same with the same exp. takes the different value of b.

B) Dead Code Elimination

A variable is said to dead if its value is not used anywhere in the program. Subsequently an assignment statement made to the dead variable can be safely removed.

$x = t_3$ ⇒ dead code.
$a[t_2] = t_5$
$a[t_4] = t_2$
print $a[t_4]$

C) Copy Propagation

let we have following expression,

$a = d+e$ $\quad$ $b = d+e$

new variable ← $C = d+e$

Here the common subexpression is $C = d+e$ is eliminated using new variable to hold the value of $d+e$

Constant folding

during at compile time that the value of an expression a constant and using the constant instead is known to constant folding.

Eg, a = 36 * 2
print a

Here, a = 72 will be computed at compile time.

(2) Algebric Transformation

Many algebric transformations can be used to change the set of expression computed by the basic block into simplified expressions or to replace the expensive operation into cheaper ex operations.

Eg, a) $x = x + 0$ and $x = x * 1$
$\Downarrow$ $\Downarrow$
$x = x$ $x = x$

b) $x = y^{power} 2$ can be replaced by $x = y * y$.

c) $2 * x$ ↳ $x + x$

d) $x/2$ ↳ $x * 0.5$

e) x^2 ↳ $x * x$.

→ Loop Optimization

(1) loop invariant code Elimination
(2) Induction variable elimination.

} Structure preserved transformation

Reduction in Strength → Algebric transformation

B) Induction Variable elimination

Induction variables are those variables whose ... in the loop are of the form $I = I + C$ where C is ... Elimination of induction variables also performs ... reduction.

Eg,

```
L : T1 := 4 * i
    T3 := T2[T1]
    I := I + 1
    if I <= 20 goto L
```

⇒

```
L : T1 = 0
    T1 := T1 + 4
    T3 := T2[T1]
    if T1 <= 76 goto L
```

Here,

T_1 and I are in the lock stage. When we increase the value of I like 1, 2 ---- 20 then the value of T_1 also increases from 4, 8 ---- 80. So, I and T_1 are the induction variables.

* Reduction in Strength

Replacement of an expensive operation by a cheaper one

Eg,

```
x = 1
for (i=1; i<=10; i++)
{
    x = i * 2;
}
```

⇒

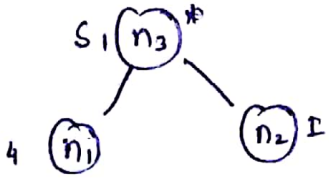
Can be optimized as,

```
x = 0
for (i=1; i<=10; i++)
{
    x = i + i;
}
//
```

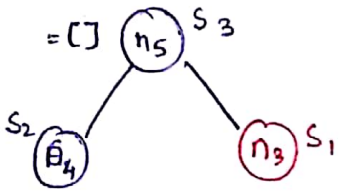
ed Acyclic Graph

- useful data structure for analysing basic block.
- It is a directed graph with no cycle.
- Determines common sub-expressions in a basic block.
- Determines which names are used inside but evaluated outside of the block.
- (5) Determines which statement of the block could have their values outside the block.

Eg, $S_1 := 4 * I$

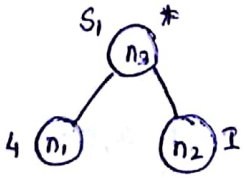


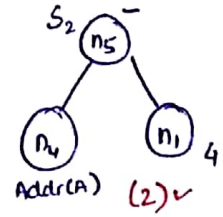
$S_3 := S_2[S_1]$

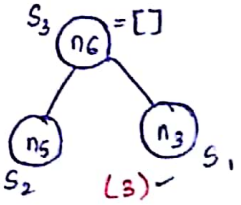


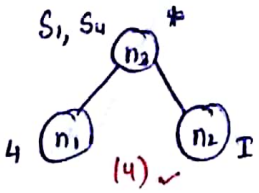
→ Constructing a DAG example

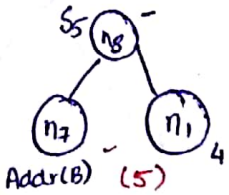
(1) $S_1 := 4 * I$
(2) $S_2 := Addr(A) - 4$
(3) $S_3 := S_2[S_1]$
(4) $S_4 := 4 * I$
(5) $S_5 := Addr(B) - 4$
(6) $S_6 := S_5[S_4]$
(7) $S_7 := S_3 * S_6$
(8) $S_8 := PROD + S_7$
(9) $PROD := S_8$
(10) $S_9 := I + 1$
(11) $I := S_9$
(12) if $I \leq 20$ goto (1)

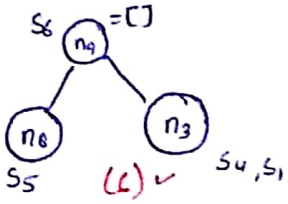


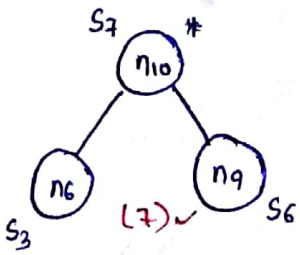


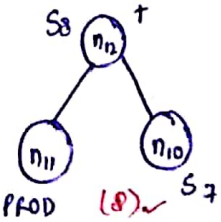


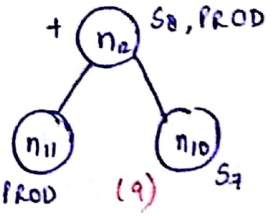


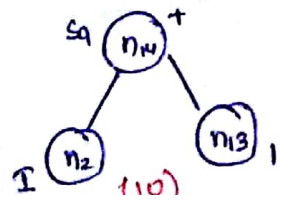


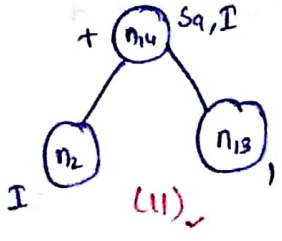


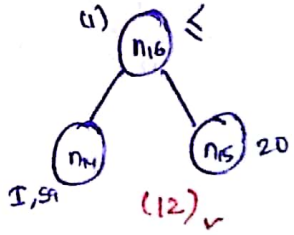




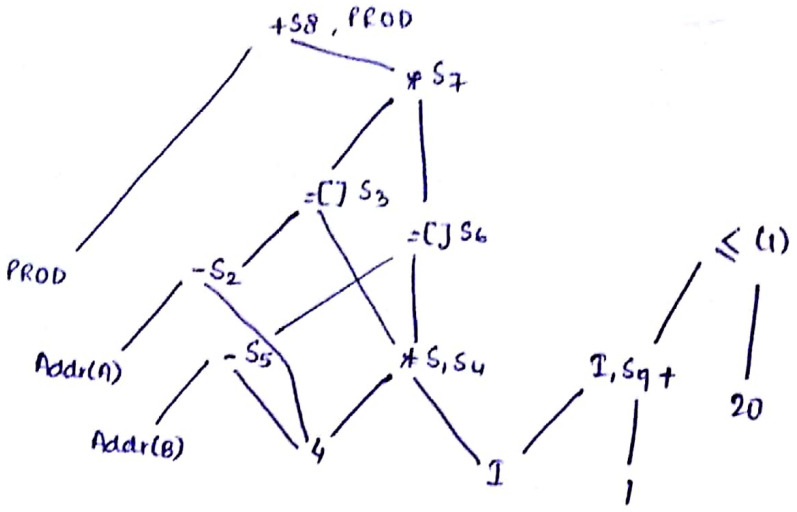








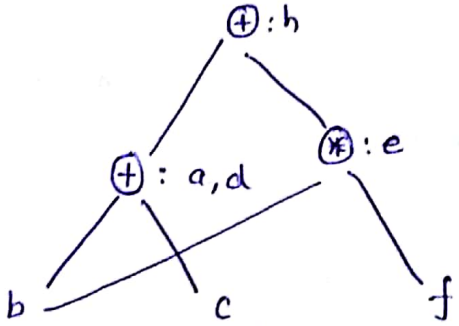
Now, Constructing DAG,



→ Basics

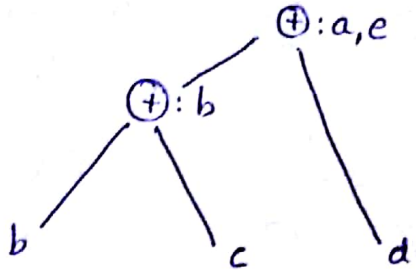
(1)

a = b + c
d = b + c
e = b * f
h = a + e



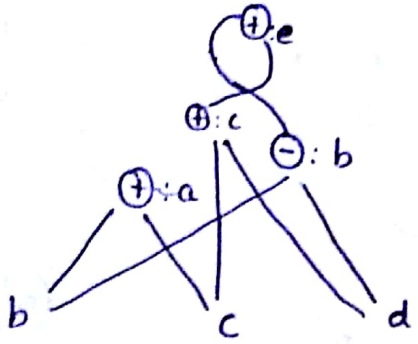
(2)

b = b + c
a = b + d
e = a



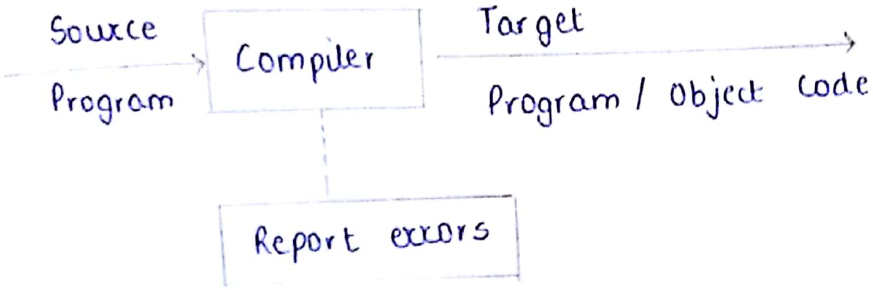
(3)

a = b + c
b = b - d
c = c + d
e = b + c



Q. What do you mean by Compiler? Write different types of compilers.

A compiler is a program that reads a program written in one language known as source language and translates into an equivalent program into another language called target language compiler also reports errors to the user.



→ Types of Compilers.

① Native code compiler : The compiler used to compile a source code for the same type of platform only is known as native code compiler.
Eg, ocamlopt

② Cross compiler : A compiler that runs on one machine and produces code for another machine is called a cross compiler. It compiles for multiple platforms from one machine.
Eg, GCC collections of compiler can be set up to cross compile.

ce to source compiler: The compilers that take a high-level language as input and output source code of another high-level language only. It is also known as a transpiler.

Eg, Script Sharp, Cfront.

④ One pass Compiler: It is a type of compiler that compiles the whole process in only one pass.

Eg, early PASCAL compilers.

⑤ Threaded Code Compilers: The compilers which simply replace a string by an appropriate binary code.

Eg, BASIC compiler.

⑥ Incremental Compiler: The compiler which compiles only the changed lines from the source code and update the object code.

Eg, IBM Visual Age C++ compiler 4.0.

⑦ Source Compiler: The compiler which converts the source code high-level language code into assemby language only.

Explain analysis Synthesis model of a compiler. Also explain analysis tools.

In analysis synthesis model of a compiler, compilation process is divided into two parts:

1. Analysis
2. Synthesis

① Analysis

It breaks up the source program into constituent pieces and creates an intermediate representation of the source program. During analysis, the operations implied by the source program are determined and recorded in a syntax tree. The analysis part also collects the important information about the source program and stores it in a data structure called symbol table, which is passed to synthesis part.

② Synthesis

It constructs desired target program by using intermediate code and information stored in the symbol table provided by the analysis part.

Various analysis tools are:

a) lexical ~~analysis~~ : analyzer,

It reads the streams of characters from the source program and groups it into meaningful sequences called lexemes which produces tokens of the form,

< token name, attribute value >

Analyzer,

characters or tokens are grouped hierarchially noted collections. The parser uses the tokens produced by lexical analyzer to create syntax tree.

Semantic Analyzer,

It uses the syntax tree and information in the symbol table to check the source program for semantic consistency. It also gathers type information, performs type checking and some implicit conversions called coersions.

the following,

Symbol table,

A symbol table is a data structure containing a record for each identifier with attribute details of the identifier. When an identifier in a source program is detected by lexical analyzer, the identifier is entered into the symbol table. The remaining phase enter the information about identifier into the ST and then use their information in various ways.

② Error detection and Reporting,

Each phase of the compiler encounters errors. The lexical analyzer phase can detect errors only where the characters remaining in the input do not form any token of the language. Syntax error detects errors when token streams violates the structure rules. Semantic analyzer tries to detect the errors for the programmatical constructs that have correct syntactic structure but no meaning.

③ Text formatter

A text formatter takes input that is a stream of characters, most of which is text to be typeset but some of which includes commands to indicates paragraphs, figures or mathematical structures like subscripts.

g a compiler,

ation to a compiler several others program may be ed to create an executable target program,

preprocessor,

It is a program that processes its input data to produce output that is used as input to another program. They may peform the following functions.

1. Macro processing
2. File inclusion
3. Rational preprocessor
4. Language extensions.

b) Assembler,

It creates object code by translating assembly instruction mnemonics into machine code. They may be one-pass assembler or two-pass compiler.

c) Linkers,

A linker is a program that takes one or more objects generated by a compiler and combines them into a single executable program.

d) Loader

A loader is a part of the OS that is responsible for loading programs in memory.

Construction Tools,

development tools that are available for implementation of one or more compiler phases are known compiler construction tools. These are as follows;

Parser Generator,

They produce syntax analysis normally from context free grammar input.

Eg, PIC, EQM.

b) Scanner Generator,

These generators automatically generate lexical analyzer, normally based on a regular expression.

c) Syntax - directed Translation Engines,

They generate intermediate code with three add format based on the parse tree.

d) Automatic Code Generator,

It takes a collection of rules that defines the translation of each operation of the intermediate language for the target mach.

e) Data flow engines,

It gathers informations about how values are transmitted from one part of the program to each other part.

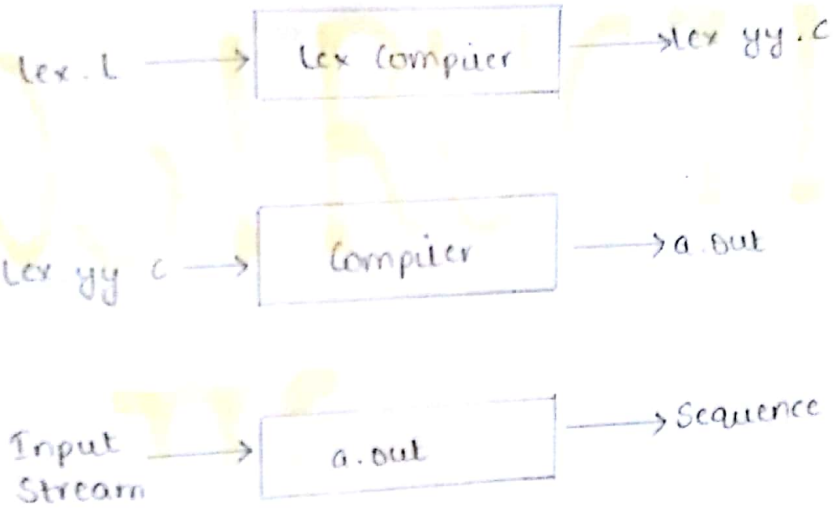
Case study on LEX.

LEX:

LEX is a tool in lexical analysis phase to recognise tokens using a regular expression.

① LEX tool itself is a lex compiler.

② A LEX source program is a specification of lexical Analyser consisting of set of regular expressions together with an action for each regular expression.



* lex.l : is an input file written in a language which describes the generation of lexical analyser. lex compiler transforms lex.l to C program known as lex.yy.c

lex.yy.c :- is compiled by the c-compiler to file called a.out.

The attribute value can be numeric code, pointer to Symbol table or nothing.

of lex programs :

gram will be in the following form

arations

%

anslation Rules

%

auxiliary functions

a) Declarations : This section includes declaration of variables, constants and regular definitions.

b) Translation Rules : It contains regular expression and code segments.

form : Pattern { Action }

c) Auxiliary Functions : holds additional functions which are used in actions. These are compiled separately and loaded with lexical Analyser.

* Conflict Resolution in lex :

Conflict arises when several prefixes of input matches one or more patterns. This can be resolved by the following.

① Always prefer a longer prefix than a shorter prefix.

② If two or more patterns are matched for the longer prefix, then the first pattern listed in lex program is preferred.

lexical Analyser

analyser can either be generated by NFA or by

is preferable in the implementation of Lex.

...entiate between top - down and bottom - up ...owing.

Top - Down	Bottom - Up
① Starts at the root and goes to the leaves.	① Starts at the leaves and goes to the root.
② It performs a left most derivation.	② It performs Right most derivation in reverse.
③ It ends when the stack is empty.	③ It starts when the stack is empty.
④ It expands non-terminals.	④ It reduces non-terminals.
⑤ It reads terminals when popped off Stack.	⑤ It reads terminals when they're pushed on stack.
⑥ Pre Order traversal of Parse tree.	⑥ Post Order traversal of Parse tree.
⑦ Pops the non-terminals and pushes it's RHS onto Stack.	⑦ It tries to recognise RHS. Pops it and pushes corresponding non-terminal.
⑧ The Stack is used for what is expected.	⑧ The stack is used for what is already seen.
⑨ It may requires back tracking.	⑨ It doesn't require back tracking.
⑩ It requires grammar free of left recursion & left factoring.	⑩ It doesn't require such a grammar.

advantages and disadvantages of top-down bottom-up parsing.

Top-down Parsing:

Advantages,

1. Highly flexible
2. Efficient error recovery.

Disadvantages,

1. Semantic actions cannot be performed while making a prediction. The actions must be delayed until the prediction is known to be a successful parse.
2. Precise error reporting is not possible. A mismatch merely triggers back tracking. A source string is known to be erroneous only after all predictions have failed.

Bottom-Up Parsing;

Advantages,

1. Attribute computation is easy.
2. Parser can be generated automatically.

Disadvantages,

1. It doesn't support left / right information flows.