Leftmost

write all terminal first

abbde

~~abbcbd~~ (A→b)

aAbcde (A→Abc)

aAde (B→d)

aABe (S→aABe)

S.

Rightmost.

write first production rule of grammar

S→ aABe

→ aAde

→ aAbcde

→ abbcde.

Handle pruning:-

E→E+E | E→id.

I/p string : id+id+id.

$E \Rightarrow E+E$

$E \underset{rm}{\Rightarrow} E+E+E$

$E \underset{rm}{\Rightarrow} E+E+id$ (E→id).

$E \underset{rm}{\Rightarrow} E+E+id$ (E→id)

$E \underset{rm}{\Rightarrow} E+id+id$ (E→id)

$E \underset{rm}{\Rightarrow}$ id+id+id.

$E \Rightarrow E+E$ (E→E+E)

$E \underset{lm}{\Rightarrow} E+E+E$ (E→id)

$E \underset{lm}{\Rightarrow}$ id+E+E (E→id)

$E \underset{lm}{\Rightarrow}$ id+id+E (E→id)

$E \underset{lm}{\Rightarrow}$ id+id+id.

Operator precedence:- parsing.

→ An efficient & easy way of constructing a shift-reduce parser is called operator-precedence parsing.

→ It can be constructed from a grammar called Operator precedence.

Eg: consider the grammar.

< → shift
> → pop.

E → EAE | (E) | - E | id.

A → + | - | * | / | ↑ | id.

I/p string id+id*id

E → E + E | E * E | E - E | E / E | E ↑ E | (E) | - E | id.

Rules to find operator precedence:

1) ↑ is of highest precedence & right associative (<)
2) * and / are next highest & left associative (>)
3) + and - are lowest & left associative. (>).
4) equal precedence (>).
5) id (<) (b) $ (>), (∃) ((<) (8)) (>).

+, -, *, / → Left associative.

* and / higher precedence than +, -

	+	-	*	/	↑	id	(	)	$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
↑	>	>	>	>	<	<	<	>	>
id	>	>	>	>	>	-	-	>	>
(	<	<	<	<	<	<	<	=	-
)	>	>	>	>	>	-	-	>	>
$	<	<	<	<	<	<	<	-	-

Stack		Input string	Comment
$	<·	id + id * id $	shift id.
$id	·>	+ id * id $	pop the top of the stack id.
$	<·	+ id * id $	shift +
$+	<·	id * id $	shift id.
$+id	·>	* id $	pop id.
$+	<·	* id $	~~pop id~~. shift *
$+ *	<·	id $	~~shift id~~. ~~*~~ id
$+ * id	·>	$	pop id
$+ *	·>	$	pop *
$+	·>	$	pop +
$		$	Accepted.

Precedence Relation matrix

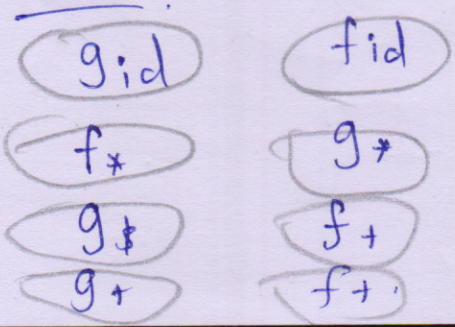
	id	+	*	$
id	-	·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	-

Create Symbol

f:id , f+ f* f$

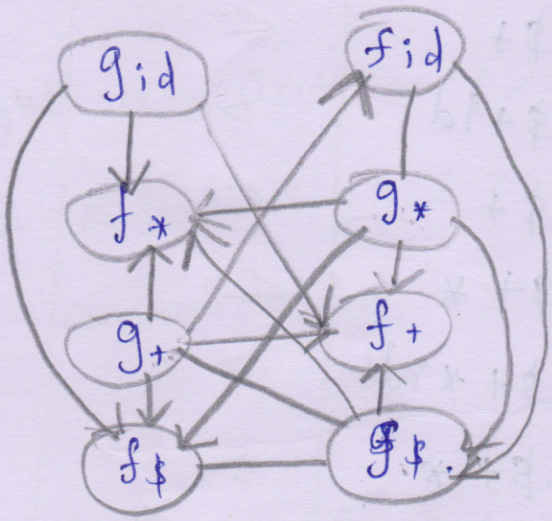
g:id g+ f* g$

create node:



place the following edges:-

+ ⋖ id , edges from g id to f +.
+ ⋖ * , edges from g * to f +
+ ⋗ $, edges from g $ to f +
* ⋖ id , edges from g id to f *
* ⋗ + , edges from g + to f *
* ⋗ $, edges from g $ to f *
$ ⋖ id , edges from g id to f $
$ ⋖ + , edges from g + to f $
$ ⋖ * , edges from g * to f $
id ⋗ $, edges from g $ to f id
id ⋗ + , edges from g + to f id.
id ⋗ * , edges from g * to f id
+ ⋗ + , edges from g + to f +
* ⋗ * . edges from g * to f *.



Eg2:

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

LR Parsing:- (Left to Right).

→ Bottom up syntax analysis technique that can be used to parse a large class of CFG is called LR(k) parsing.

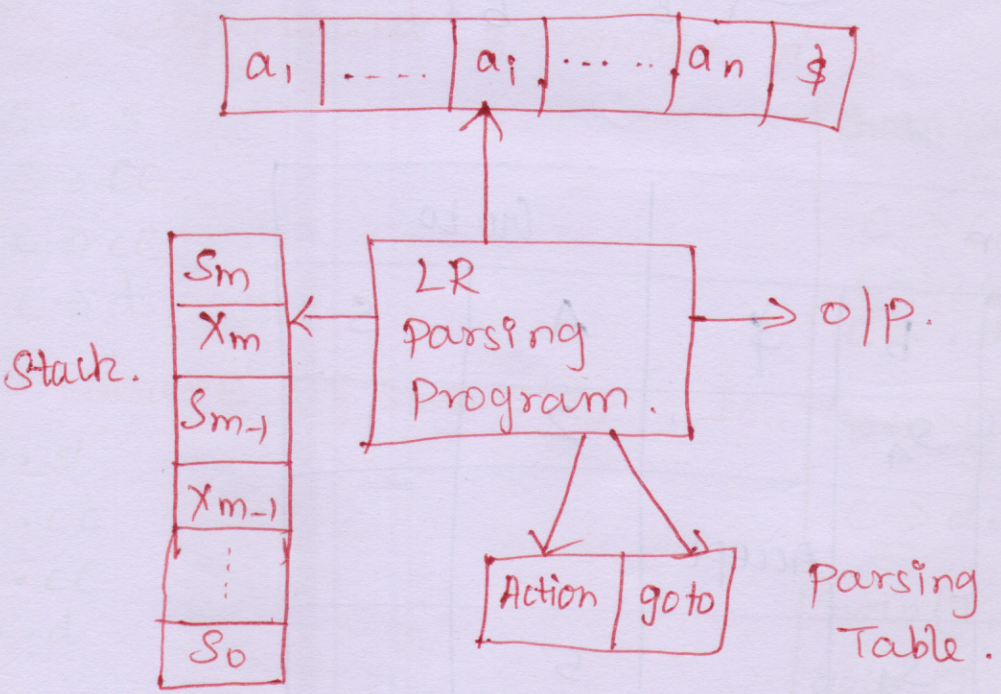
Types of LR parsing methods:-

① SLR → simple LR : Easiest to implement. Lower powerful.

② CLR → Canonical LR : Most powerful. Most Expensive.

③ LALR → Look ahead LR : Intermediate in power and cost b/w the above two method.

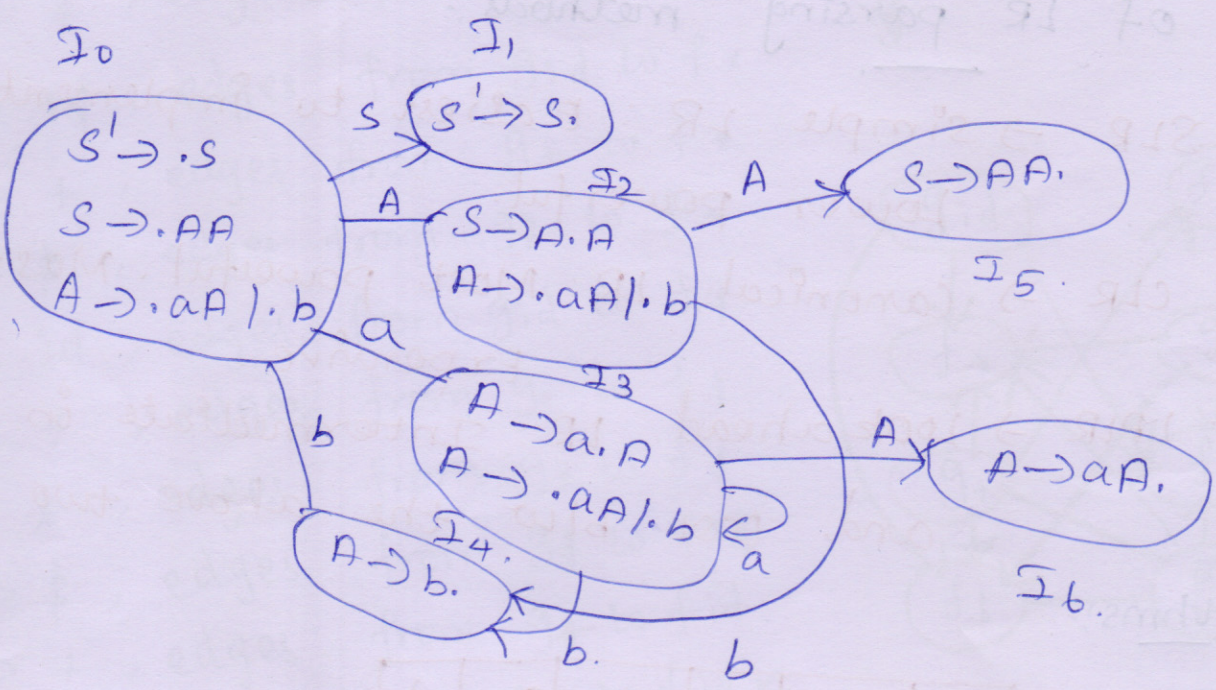
Algorithms:-



Eg:

$S \to AA$

$A \to aA \mid b$.



	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	r_1	r_1		
6	r_2	r_2	r_2		

SLR Parser (Simple Left to Right).

Steps for SLR Parser Problem

1. write an augumented grammar.
2. write the closure of starting NT
3. Apply goto() functions of for all symbols.
4. Computation of FIRST and FOLLOW.
5. Construct Parsing Table.
6. check the input string with stack implementation.

Grammar :- eg

① $S \to CC$
② $C \to cC$
③ $C \to d$ & check the I/p string "cdcd" and "ccd".

Step1: The augumented grammar G' is.

$S \to S'$
(1) $S \to CC$
(2) $C \to cC$
(3) $C \to d$.

Step 2: closure ({ $S \to .S'$ }) = I_0.

$S \to .S'$
$S \to .CC$
$C \to .cC$
$C \to .d$

Step 3: Apply goto function.

Goto (I_0, S) = I_1

$S \to S.'$

Goto (I_0, C) = I_2

$S \to C.C$
$C \to .CC$
$C \to .d$

Goto (I_0, C) = I_3.

$C \to C.C$
$C \to .CC$
$C \to .d$

Goto (I_0, d) = I_4

$C \to d.$

Goto (I_1, S) = NIL
Goto (I_1, C) = NIL
Goto (I_1, C) = NIL
Goto (I_1, d) = NIL

Goto (I_2, S) = NIL

Goto (I_2, C) = I_5

$S \to CC.$

Goto (I_2, C) = I_3

$C \to C.C$ $C \to .CC$ $C \to .d$

Goto $(I_2, d) = I_4$

$C \to d.$

Goto $(I_3, S) = NIL$

Goto $(I_3, C) = I_6$

$C \to CC.$

Goto $(I_3, C) = I_3.$

$C \to C.C$

$C \to .CC$

$C \to .d$

Goto $(I_3, d) = I_4.$

$C \to d.$

Goto $(I_4, S) = NIL$

Goto $(I_5) = NIL$

Goto $(I_6) = NIL.$

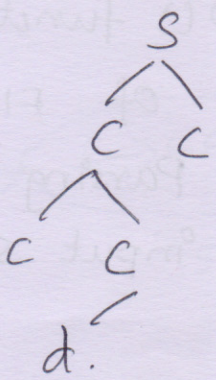
Step 4: Computation First

Grammar

$S \to CC$

$C \to cC$

$C \to d.$



first(S) = { c, d }

first(C) = { c, d }.

Step 4: Computation Follow:

follow (S) = { $ }.

follow (C) = { c, d, $ }.

Step 5: Construct pasing Table:-

S → Shift.

R → Reduce

	c	d	$	S	C
0	S3	S4		1	2
1	S3	S4	Accept.		
2	S3	S4			5
3	S3	S4			6
4	R3	R3	R3		
5			R1		
6	R2	R2	R2		

Step 6 : Stack Implementation :-

Stack	Input string	Action.
0	cdcd $	shift 3.
0c3	dcd $	shift 4.
0c3d4	cd $	Reduce 3. C→d.
0c3C6	cd$	Reduce 2 C→CC
0C2	cd$.	shift 3.
0C2c3	d$	shift 4.
0C2c3d4	$	Reduce 3 C→d.
0C2C6	$	Reduce 2 C→CC
0C2C5	$	Reduce 1 S→CC
0S1	$	Accept.

Stack Implementation "ccd."

Stack	Input string	Action.
0	ccd $	shift 3.
0c3	cd$	shift 3.
0c3c3	d$	shift 4.
0c3c3d4	$.	Reduce 3. C→d.
0c3c3C	$	~~NIL~~ Reduce 2 C→CC
0c3C6	$	~~NIL~~ Reduce 2 C→CC
0C5	$	~~NIL~~
0C2	$	Error.

Eg2:

E → E+T | T
T → T*F | F
F → (E) | id

Eg3

S → (L) | a
L → L, S | S

Canonical Left to Right parser: (CLR).

LALR(1) CLR(1)

LALR

LR(1) Item = LR(0) items.

Grammar:

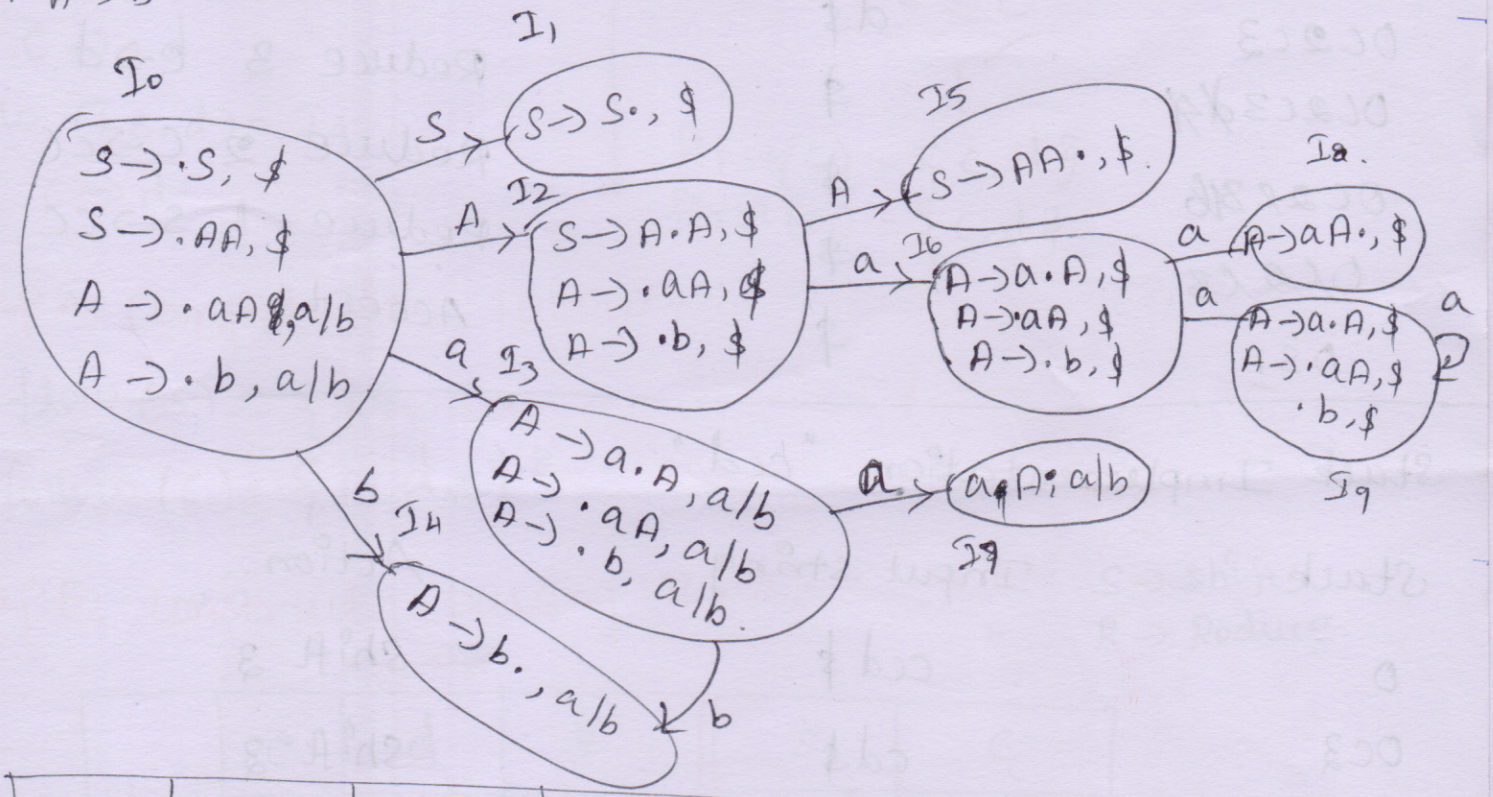
① $S \rightarrow AA$.

② $A \rightarrow aA/b$,

③ $A \rightarrow b$

First (S) = {a, b}
(A) = {a, b}

Follow (S) = {$,}
(A) = {$, a, b}



	a	b	$	S	A
0	S3	S4		1	2
1			Accept		
2	S6				5
3	S8	S4			
4					
5	R5	R5	R5		
6	S9				
7	R2	R2	R2		
8	R2	R2	R2		
9	S9	S6			

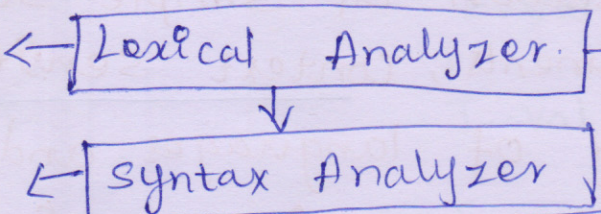
YACC → Yet Another Compiler Compiler.

lex → Lexical Analyzer generator.

YACC → parser generator.

Its a tool which generate LALR Parser.

lex → RE specification.



YACC working:-

Step1: YACC specification

parser.y → YACC Compiler → y.tab.c.

Step 2:

y.tab.c → C compiler → a.out {S.A}.

Step 3:

i/p {tokens LA} → a.out → o/p {parse}.

Parse.

Grammar.y → YACC → y.tab.c {source pgm by YACC}

{file containing desired grammar in YACC format.} {YACC pgm}

Complier

i/p ← a.out → S.A → parse tree.

token gen by lex.

Syntax:

definitions

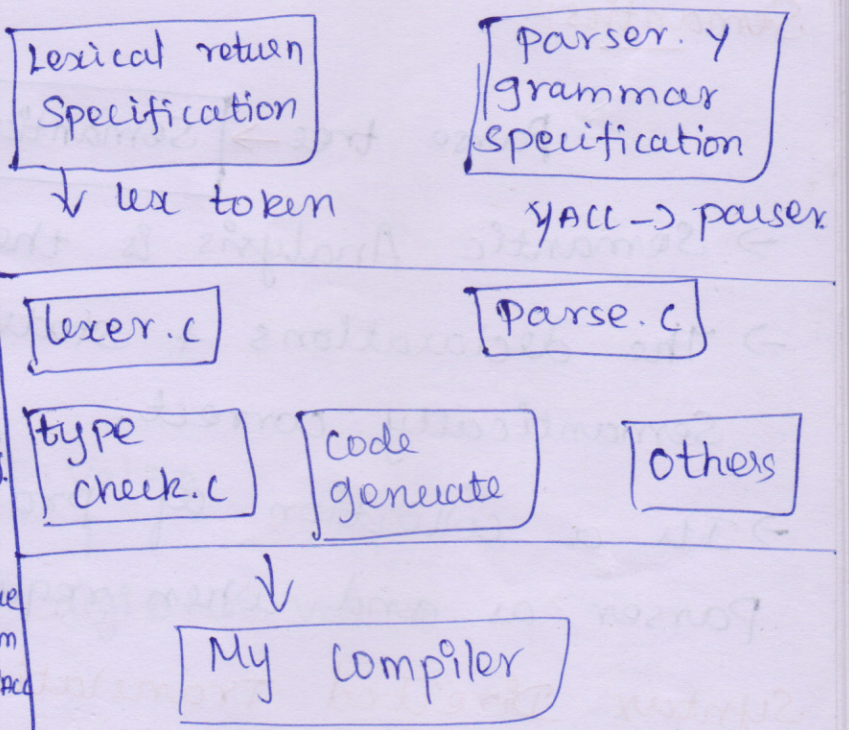
%%

Rules

%%

supplementary code.

Big picture of lex/YACC:-



Definition:-

Declaration of tokens & types of values used on a parse.

Rules:

List of grammar rules with semantic return.

Unit-III

Semantics And Context Sensitive Features.

Semantics: Syntax directed translations, S-attributed L-attributed grammars, Intermediate code-abstract syntax tree, translation of simple statements and control flow statements, context sensitive features. Chomsky hierarchy of languages and recognizers, Type checking type conversions, equivalence of type expressions, overloading of functions and operations.

Semantics:-

Parse tree → [Semantics] → Annoted Grammar.

→ Semantic Analysis is the third phase of compiler.

→ The declarations & statements of program are semantically correct.

→ Its a collection of procedures which is called by Parser as and when required by grammar.

Syntax Directed Translation:-

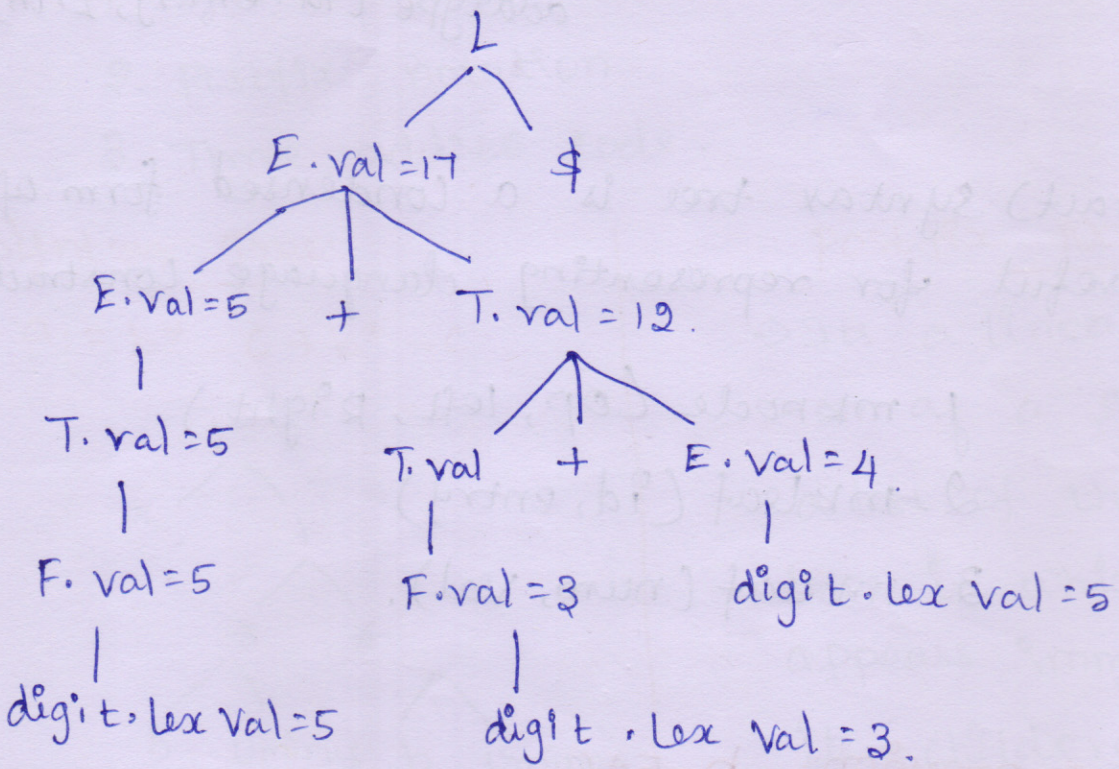
→ An arrociate if the input with the programming languages constructs by attaching attributes to grammar Symbols.

Synthesized attributes

Eg:-

Production	Semantic Rules.
L → E S	Print (E.val)
E → E1 + T	E.val → E1.val + T.val.
E → T	E.val → T.val.
T → T1 * F	T.val → T1.val * F.val.
F → (E)	F.val → E.val.
F → digit.	F.val → digit.lexval.

Eg2: Input String. : 5+3*4 = 17. (Annotated parse Tree).



STD:

1. Synthesized Attributes
2. Inherited attributes.

2. Inherited attributes:

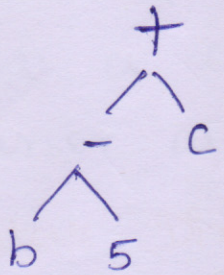
→ An inherited attributes is one whose value at a node in a parse tree is defined in terms of attributes at the parent and siblings of that node.

Production	Semantic Actions.
D → TL	L.in := T.type.
T → int	T.type := integer.
T → real	T.type := real.
L → L1, id	L1.in := L.in addtype (id.entry, L.in)
L → id	addtype (id.entry, L.in).

Syntax Trees:

→ An (abstract) syntax tree is a condensed form of parse tree. useful for representing language constructs.

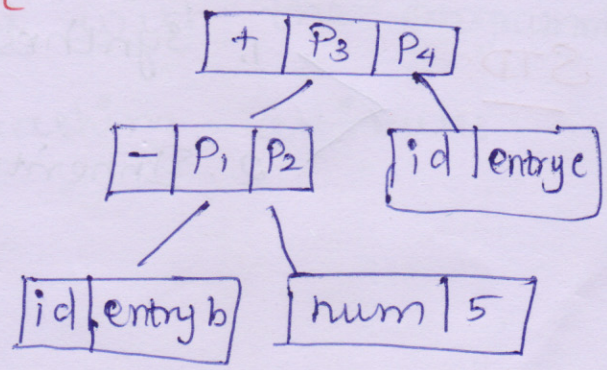
Eg: b-5+c



1. mknode (op, left, Right)
2. mkleaf (id, entry)
3. mkleaf (num, val).

Syntax Tree expression:- b-5+c

1. P_1 := mkleaf (id, entry b);
2. P_2 := mknum (num, 5);
3. P_3 := mknode ('-', P_1, P_2)
4. P_4 := mkleafe (id, entry c)
5. P_5 := mknode ('+', P_3, P_4).



Production	Semantic Rules.
$E \rightarrow E_1 + T$	E.nPtr := mknode('+', E_1.nPtr, T.nPtr)
$E \rightarrow E_1 - T$	E.nptr := mknode('-', E_1.nptr, T.nPtr)
$E \rightarrow T$	E.nPtr := T.nPtr.
$T \rightarrow (E)$	T.nPtr := E.nPtr
T → id	T.nPtr := mkleaf (id, id.entry)
T → num	T.nPtr := mkleaf (num, num.val)

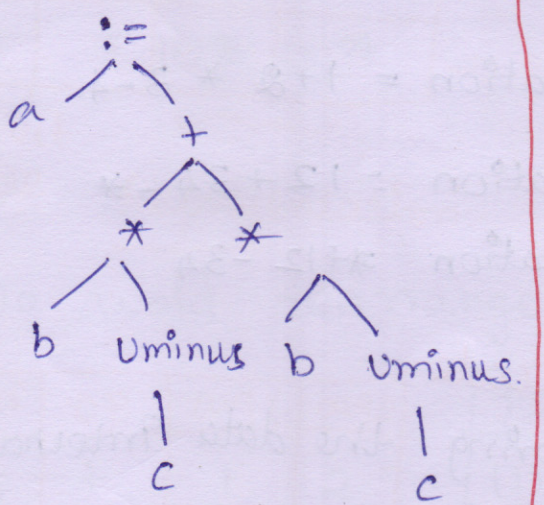
Intermediate Languages:- (or) code

→ There are 3 kinds of intermediate representations

1. syntax trees
2. postfix notation
3. Three addres code.

Syntaax Tree:

a := b* - c + b* - c



3. Three address code.

~~x = y Op z~~

x + y * z is,

t_1 := y * z

t_2 := t_1 + x

Postfix Notation

→ Its a linearized representation of a syntax tree, its a list of the nodes of the tree in which a node appears immedately after its childers.

→ The postfix notation for above syntax tree is

abc uminus *bc uminus * + assign

Three address code:

→ It contains maximum 3 addresses.

→ Complex expressions can be converted into simple address format.

eg:

$a = (b+c) / d;$ →

$T1 = b+c;$ → MOV b, R0
MOV c, R1
$T2 = T1 / d;$ ADD R0, R1
$a = T2;$ MOV R1, T1

→ Its easy to generate and can be easily converted to machine code.

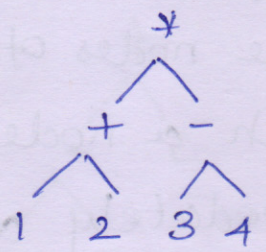
→ It makes use of maximum three addresses, with temporary variables.

→ The compiler decides the order of operation given by three address code.

Three address code -generation:

→ Parse tree into Linear representation.

eg: $((1+2) * (3-4))$



→ Infix notation = 1+2 * 3-4.

→ Postfix notation = 12+34-*

→ Prefix notation *+12-34

Representation of 3 address code:

→ It means storing the data internally.

1). Quadruple.

2). Tuples

3). Indirect Tuples.

Quadruple representation:

→ Its a Structure with consist of 4 fields namely Operator (op), Operand (arg 1), arg 2 and Result.

oper	op	arg1	arg2	res.
(1)	+	1	2	3
(2)	−	3	4	−1
(3)	*	3	−1	−3

Eg: (1+2) * (3-4)

T1 = 1+2

T2 = 3-4

T3 = T1 * T2

Tuple Representation:-

→ To avoid extra temporary variable.

→ Remove Result column from the Quadruple Representation.

Oper	OP	arg1	arg2
(1)	+	1	2
(2)	−	3	4
(3)	*	(1)	(2)

→ Representation Stored here.

Eg: (1+2) * (3-4)

T1 = 1+2

T2 = 3-4

T3 = T1 * T2

Indirect Tuple Representation:-

→ To avoid rearrangement problem during optimization.

Oper	OP	arg1	arg2
(1)	+	1	2
(2)	−	3	4
(3)	*	(51)	(52)

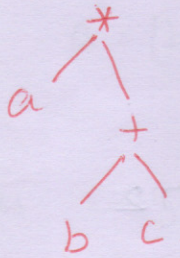
PSN	ptr.
(1)	(51)
(2)	(52)
(3)	(53)

Abstract Syntax Tree:

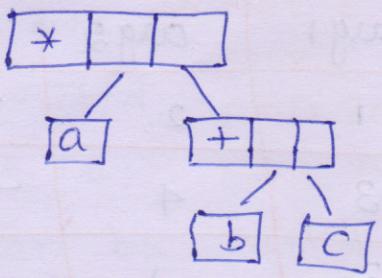
→ Its more compact than a parese tree and can be easily used by compiler.

eg: a * b + c.

Parse tree.



Abstract Syntax tree.



Translation of Simple Statement into control flow Statement:

In the Statement basically two types.

① Numerical Representation.

② Flow of control.

① Numerical Representation	Flow of control
→ To encode true / False value numerically.	→ position reached.
→ Evaluate Boolean expression like airthmetic expression.	→ convenient for if-then and loop Statement.

eg:

In the statement if (E) S

Boolean expression.

① if-then

if-then-else

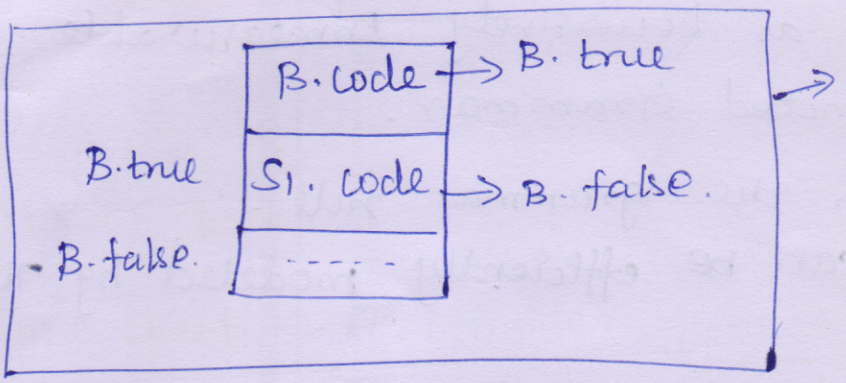
while-do

S → if (B) S1

| if (B) S1 else S2 → programming

| while (B) do S1 → Statement

Simple Statement:



→ code for the control flow statements.

S → if (B) S_1	B.true = newlable() B.false = S_1.next = S.next S.code = B.code || label(B.true) || S_1.code.

→ SDD.

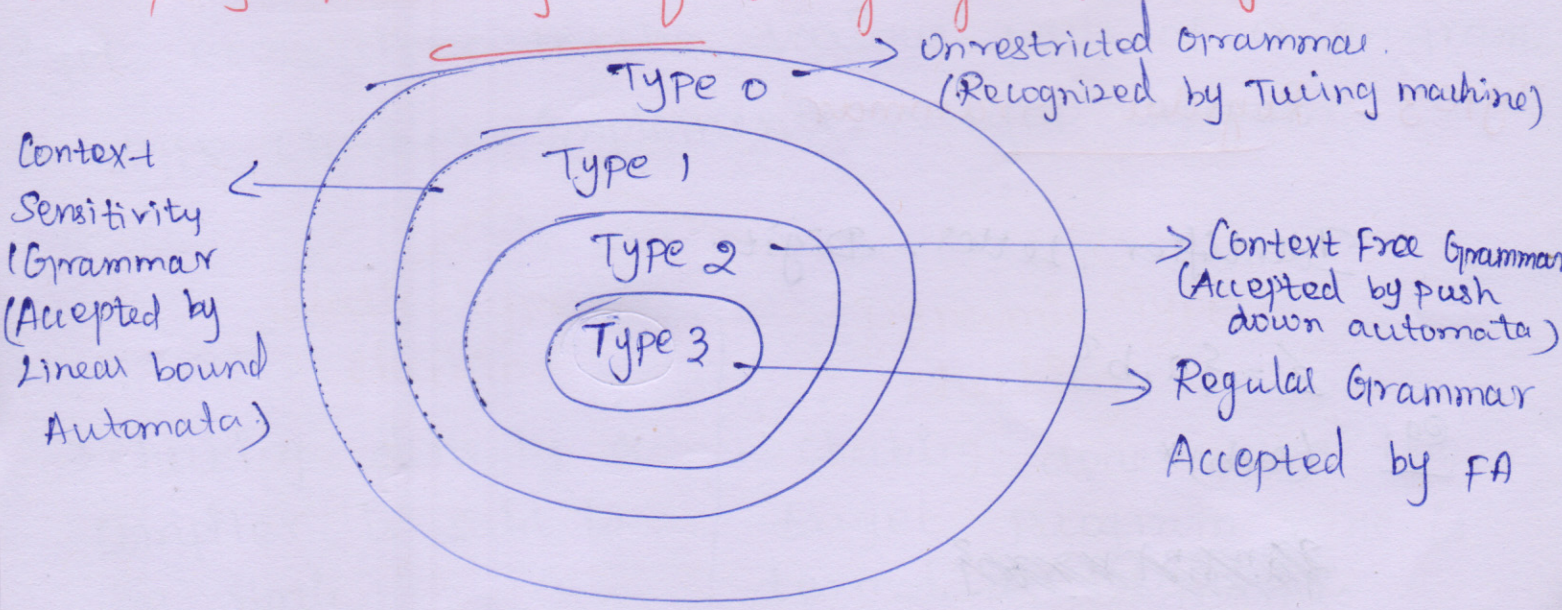
Eg: 1. if (a<b) S_1.

→ Assume that the attributes true & false exist for the entire expression as labels Ltrue & Lfalse res.

```
if a<b goto Ltrue
goto Lfalse
Ltrue: S1.code
Lfalse.
```

Context Sensitivity Features

Chomsky Hierarchy of language & Recognizers.



Type 0 - Unrestricted Grammar

→ Its also known as Recursively Enmeasurable language / Unrestricted Grammar.

→ No restriction as the grammar rule.

→ There language can be efficiently modeled by Turing machine.

$\alpha \rightarrow \beta$

α is $(V+T)^* . V (V+T)^*$, β is $(V+T)^*$

Eg:

$Sab \rightarrow ba$	S, A → Variable.
$A \rightarrow S$	ab → Terminal.

Type 1 - Context Sensitive Grammar

1) Eg:

$V \rightarrow T$

$S \rightarrow AT$

$T \rightarrow xy$

$A \rightarrow z$.

Type 2 - Context Free Grammar

$A \rightarrow aBb$

$A \rightarrow b$

$B \rightarrow a$.

Type 3 - Regular Grammar

Identifier, Letter, Digit.

$\Sigma = \{a, b\}$.

Eg: $(a+b)^*$

~~$\{aa, bb, ab, ba\}$~~.

Type checking:

→ A compiler must check the source program after syntactic convention is called static checking.

Eg: static check:

(1) Type check → A compiler generate error if an operand is applied to an incompatible operand.

(2) Flow of control check → statement that cause flow of contrd to leave a construct must have some place to which to transfer the flow of control.
eg: break statement.

(3) Uniqueness check → An object must be defined exactly only once.
eg: Labels in case statement.

(4) Name - Related check → Some times, the same name must appear two or more times.

Type system is a collection of rules of assigning type expressions to the various parts of a program.
A type checker implements a type system.

Static Type checking

→ checking done by a compiler is said to be static.

Dynamic Type checking.

→ checking done when the target program runs is termed as dynamic.

Type Expressions:-

→ A type expression is either a basic type or is formed by applying an operator called a type constructer to other type expressions.

1. Arrays.
2. Products
3. Records.
4. pointers.

Type checking expressions:-

→ The synthesized attributes type for E gives the type expression assigned by the type system to the expression generated by E.

E → literal { E.type := char }.

E → num { E.type := integer }

E → id { E.type := lookup (id.entry) }.

When an identifier appears in an expression, declared type is fetched and assigned to the attribute type.

$E_1 \rightarrow E_1$ mod E_2 { E.type := if E_1.type := integer and E_2.type = integer then integer

else type-error }.

The expression formed by applying the mode operator to two subexpressions of type integer has type integer, otherwise, its type is type-error.

$E \rightarrow E_1[E_2]$ { E.type := if E_2.type = int and E_1.type = array (s, t) then t

else type error }.

In any array reference $E_1[E_2]$, the index expression E_2 must have type integer, in which case the result is element type t obtained from the type array(s, t) of E_1.

$E \rightarrow E_1 \uparrow$ { E.type := if E_1.type = pointer(t)
then t
else type_error }.

Type postfix operator ↑ yields the object pointed to by its operand. The type of E↑ is the type t of the objected pointed to by the pointer E.

Overloading of functions and operations:-

→ Function overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading.

→ The function is redefined by using their different types of arguments or a different number of arguments.