HLL

↓

Preprocessor

↓ pure HLL

Compiler

↓

Assembler

↓ relocatable

loader / linker

↓

executable code
(absolute machine code)

Preprocessor: is responsible for.

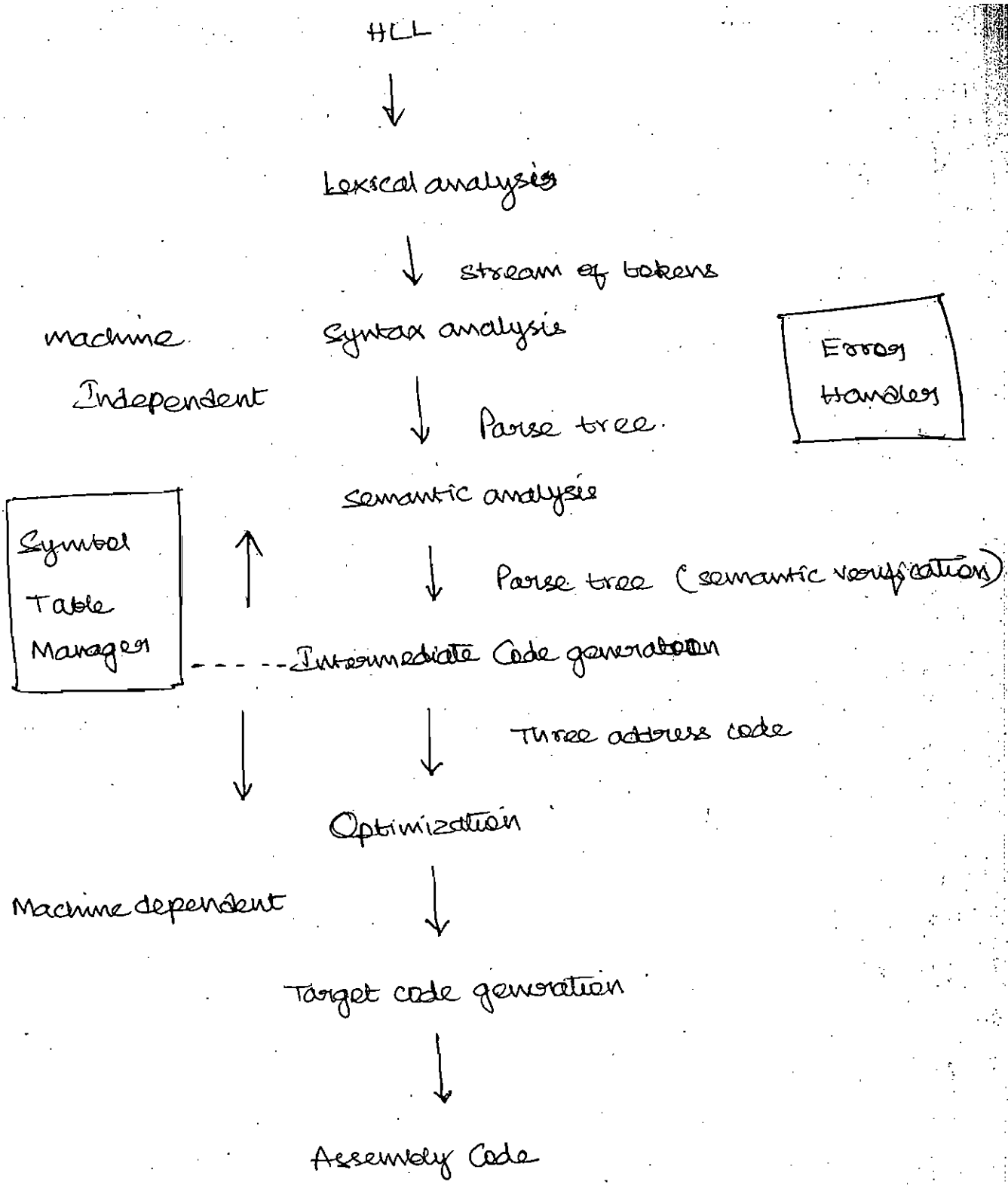
1. Macro expansion.
2. file inclusion

3. Loader: is responsible for.

1. Allocation
2. Re-allocation
3. Linking
4. Loading

Lex and yacc: are tools used in Unix operating system for compiler design. First compiler is FORTRAN. It took 18 man years to build it.

Compiler Phases.



Lexical analysis:

HLL text are source text is broken into tokens.

ex: if (A > B) 10 tokens.

a = 10 ;

Syntax analysis (Complex layer)

uses PDA to implement CFG with slight modifications

Page No.

Date: / /

eg: $x = a + b * c$.

LA

↓

id = id + id * id.

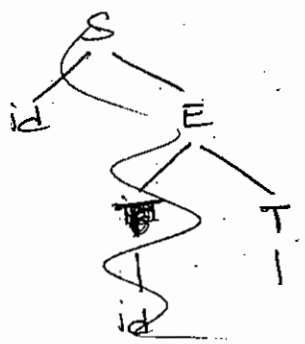
syntax analysis :

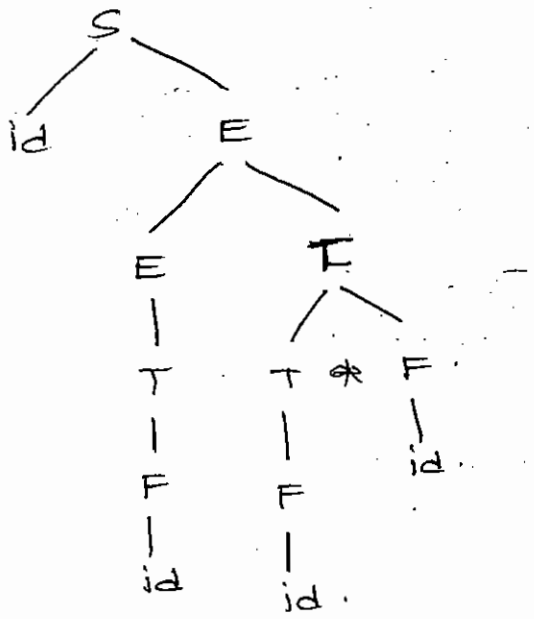
$S \rightarrow id = E$

$E \rightarrow E + T$

$T \rightarrow T * F$

$F \rightarrow id$.





Semantic Analysis: verifies whether the statements are meaningful or not. Performs static type checking.
Interpreters can perform dynamic type checking.

Intermediate code generator - Parse tree converted to three address code.

Ex $x = a + b * c$.

$t_1 = b * c$

$t_2 = a + t_1$ $x = t_2$

Optimizer: main goal is to decrease no of instructions

$x = a + b * c$

```
t1 = b * c
t2 = a + t1
x = t2
```

↓

```
t1 = b * c
x = a + t1
```

Target code generation: Intermediate code will be converted to assembly code according to the architecture.

Phases:

Frontend
depends on source - till ICG.

Backend
depends on target machine

LANCE tool takes HLL produces ICG.

Symbol table
used to store information about various tokens.

```
int x, y, z;
```

token	lexeme	type
id	x	int
id	y	int
id	z	int

Interpreter:
No object code is produced. it reads the text line by line and executes each line. Interpreter is more portable than compiler. Dynamic type specification is possible only with interpreter. eg SNOBOL $x = 10 \ldots x = 10.5 \ldots x = \text{"string"}$

Lexical Analyser:

secondary process of lexical analyser:

1. Removing the comment lines & wide space characters
2. Correlating error messages
3. only at the lexical analyser the code or text is read character by character. This helps in giving the line number at which error has occured.

Design tool: lex, hand code,

Generally token rules are described by regular expressions.

eg: identifier. $l(l/d)^*$.

Convert regular expression to finite automata.

For lex tool, give the regular expression as input. It gives DFA as output. ex: int max(x, y)

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

25

printf("%d Hai", x); . 8 .

A set of finite rules which defines infinite sentences.

LHS of production are variables and remaining are terminals

ex: $E \longrightarrow E+E \mid E*E \mid i$

i + i * i

$E \longrightarrow E+E$

$E + E*E$

$E + E*i$

$E + i*i$

$i + i*i$

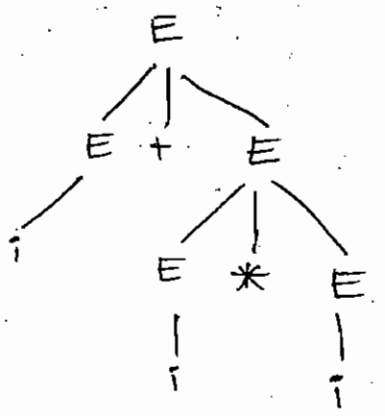
Left most derivation

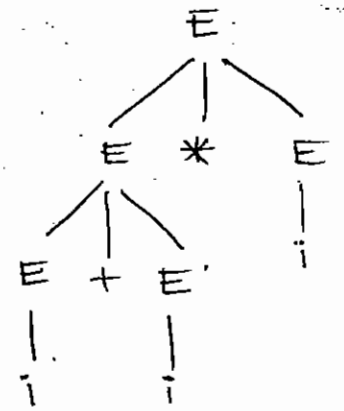
$E \longrightarrow E*E$

$E+E*E$

$i + i*i$

Right Most Derivation





Ambiguous.

More than one left most derivation or right most derivation for a given string.

LMD / RMD represent different parse trees.

More than one parse tree for a given string.

Unambiguous.

There exists one leftmost and one right most derivation exactly.

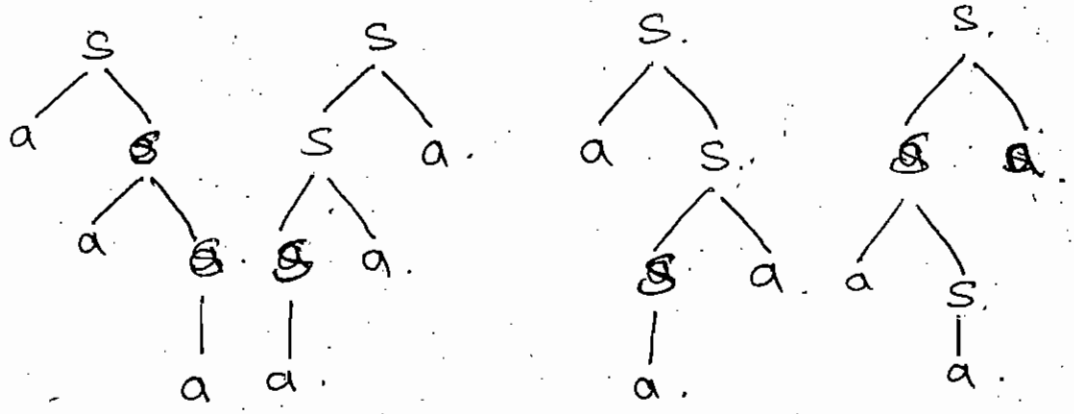
LMD / RMD represent same parse tree.

Unique parse tree for a given string.

$S \longrightarrow aS \mid Sa \mid a$

$w = aaa.$

$S \longrightarrow$	aS	$Sa.$	$aS.$	$Sa.$
	$aaS.$	$Saa.$	$aSa.$	$aSa.$
	$aaa.$	$aaa.$	$aaa.$	$aaa.$



$S \rightarrow aSbS \mid bSaS \mid \epsilon$.

Page No.
Date: / /

$W = abab$.

$S \rightarrow aSbS$ ~~$aSbS$~~
~~$aSb\epsilon$~~ ~~$a\epsilon bS$~~
~~$a\epsilon b\epsilon$~~ ~~$a\epsilon b\epsilon$~~

$S \rightarrow aSbS$ $aSbS$
$abSaSbS$ $aSbSbSbS$
$ab\epsilon aSbS$ $a\epsilon baSbS$
$ab\epsilon a\epsilon bS$ $a\epsilon ba\epsilon bS$
$ab\epsilon a\epsilon b\epsilon$ $a\epsilon ba\epsilon b\epsilon$

$A \rightarrow R+R \mid RR \mid R^* \mid a \mid b \mid \epsilon$ regular expression.

Left recursive, right recursive

Note Ambiguity is not allowed by many parsers.

Left recursive

$A \rightarrow A\alpha \mid \beta$

Head recursion

Right recursive

$A \rightarrow \alpha A \mid \beta$

Tail recursion.

Eliminate left recursion.

$\beta\alpha^*$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

left recursion leads to infinite loop so we eliminate left recursion by rewriting as above.

$E \longrightarrow E+T \mid T.$

$T \longrightarrow T*F \mid F.$

$F \longrightarrow id.$

$E \longrightarrow TE'.$

$E' \longrightarrow +TE' \mid \epsilon.$

$T \longrightarrow FT'$

$T' \longrightarrow *FT' \mid \epsilon.$

$F \longrightarrow id.$

$A \longrightarrow A\alpha \mid \beta.$

$\Downarrow$

$A \longrightarrow \beta A'$

$A' \longrightarrow \alpha A' \mid \epsilon$

$S \longrightarrow (L) \mid x.$

$L \longrightarrow L,S \mid S$

$L \longrightarrow .SL'$ eliminate left recursion.

$L' \longrightarrow ,SL' \mid \epsilon$

$S \longrightarrow SOS1S \mid 01.$

$\alpha = OS1S.$

$\beta = 01$

$S \longrightarrow 01S'$

$S' \longrightarrow OS1SS' \mid \epsilon$

$A \longrightarrow A\alpha \mid \beta.$
$A \longrightarrow \beta A'$
$A' \longrightarrow \alpha A' \mid \epsilon$

$A \longrightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots$
$\beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$

$A \longrightarrow \beta_1 A' \quad A' \longrightarrow \alpha_1 A' \mid \epsilon.$
$\beta_2 A' \qquad \alpha_2 A'$
$\beta_3 A' \qquad \alpha_3 A'$
$\vdots \qquad \vdots$
$\beta_n A' \qquad \alpha_n A'$

Grammars can be deterministic non deterministic

$A \longrightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$ non deterministic

$A \longrightarrow \alpha A'$

$A' \longrightarrow \beta_1 \mid \beta_2 \mid \beta_3$. deterministic

Using deterministic version we can avoid backtracking which is caused if we use non deterministic grammar

ex: $S \longrightarrow iEtS \mid iEtSeS \mid a$.

$E \longrightarrow b$.

$S \longrightarrow S' \mid S'eS \mid a$.

$S' \longrightarrow ibtS \mid$ [illegible].

$S \longrightarrow iEtS \mid iEtSeS \mid a$.

$S \rightarrow iEtSS'$.

$S' \longrightarrow eS \mid \in$.

$E \longrightarrow b$.

$S \rightarrow iEtS \mid iEtSeS \mid a$

$E \longrightarrow b$.

$S \longrightarrow B \mid BeS \mid a$.

$B \longrightarrow iEtS$.

$S \longrightarrow b$.

iEtiEtSeS generate using G_1 & G_2

Removing non determinism does not eliminate ambiguity of the grammar.

S → m | u

m → i E t m e m | a

u → i E t S | i E t m e u.

E → b.

S → i E t S.

i E t i E t S e S.

i E t i E t S e S.

i E t i E t m e m.

S → i E t m e m

i b t i b t a e a.

i E t i E t a e m

↓ ↓

a. a

i E t

~~S → i E t m e m~~

= i b t i b t a e a.

~~Convert left factor~~ the following grammar.

S → aSSbS | aSaSb | abb | b.

B → aS S → BSbS | aSb | abb

S → aSSbS | aSaSb | abb | b

S → aSS' | abb | b

S' → SbS | aSb | ∈

S → aS''

S'' → SS' | bb

$S \rightarrow bSSaaS \mid bSSaSb \mid bSb \mid a$

$S \rightarrow bSS' \mid a$

$S' \rightarrow SaaS \mid SaSb \mid b$

$S' \rightarrow SaS'' \mid b$

$S'' \rightarrow aS \mid Sb$

Parsers: syntax analysers.

Top Down Parser

$S \rightarrow aABe$

$aAbcBe$

$abbcBe$

$abbcde$

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

$w = abbcde$

Bottom Up Parser

constructs parse tree starting with root (start symbol) and proceeds to children.

It uses left most derivation

constructs parse tree starting with children and proceeds to root

uses reverse of right most derivation.

Design of topdown parser

TDP

TDP with backtracking

(Brute Force method)

TDP without backtracking

grammar should be free of left recursion & left factoring

(Predictive parsing)

Recursive descent

non recursive descent LL(1)

Brute force:

G: $S \rightarrow aAd \mid aB$.

$A \rightarrow b \mid c$

$B \rightarrow ccd \mid ddc$

$w = addc$.

$S \rightarrow aAd$.

acd.

Backtrack 1.

$S \rightarrow aB$.

accd.

Backtrack 2.

$S \rightarrow aB$

addc.

Disadvantage is too many backtracks and process is deathslow

Without backtracking:

In these methods Grammar should be ~~left~~ factored to avoid backtracking and free from left recursion to avoid infinite loops

Recursive descent:

Writing recursive procedure for each non terminal is recursive descent parser.

ex: $E \rightarrow E+T \mid T$.

$T \rightarrow i$

$A \rightarrow A\alpha \mid \beta$

$\alpha = +T$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

$E \rightarrow$

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow i$

$E \rightarrow iE'$

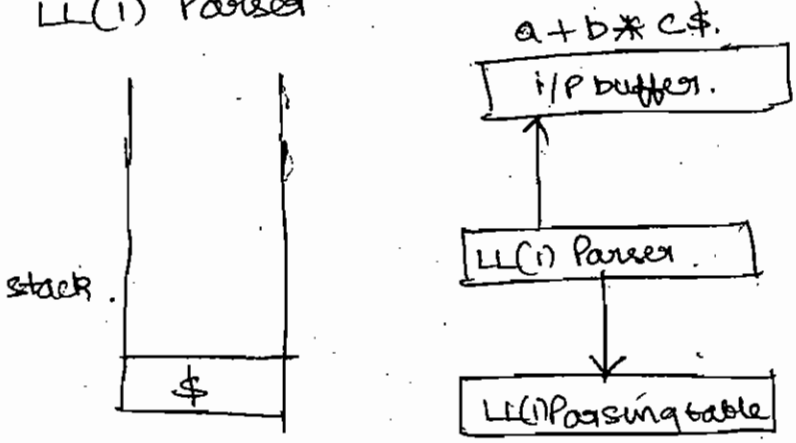
$E' \rightarrow +iE' \mid \epsilon$

l= lookahead.

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

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

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

main()
{
  E();
  if(l==$)
     pf("parsing success");
}
```

LL(1) scan input from left & lookahead 1 and do left most derivation.

LL(1) Parser



Algorithm:

If X is the Grammar symbol on top of the stack and 'a' is the lookahead symbol then,

1. If $X = a = \$$ then parsing a successful.
2. If $X = a \neq \$$ then pop the stack and increment the input pointer
3. If X is non terminal then, consult the parsing table M. Take the production from $M[X, a]$, if $M[X, a]$ is $X \rightarrow uvw$, replace X by uvw by pushing them such that u appears on top

Constructing Parsing Table M

First(α): it is the set of all terminals that begins in any string derived from α.
where α is string of grammar symbols.

$S \rightarrow ABCDE$
$A \rightarrow a/\epsilon$ $D \rightarrow d/\epsilon$
$B \rightarrow b/\epsilon$ $E \rightarrow e/\epsilon$
$C \rightarrow c$

First(A) = {a, ε} First(E) = {e, ε}
First(B) = {b, ε} First(S) = {a, b, c, d, e}
First(C) = {c} {abcde, bcde, cde, de, e, ...}
First(D) = {d, ε}.

1. If α is a terminal First(α) = terminal itself
   First(α) = α.

2. If α is a non terminal
   a) and has ϵ production ($\alpha \longrightarrow \epsilon$)
      then $A \longrightarrow \epsilon$
      First(α) = ϵ

   b) and has non null production ($\alpha \longrightarrow \beta$)
      ($\alpha \longrightarrow X_1 X_2 X_3$).
      then First(α) = First($X_1, X_2 X_3$)
      = First(X_1), $X_1 \not\Rightarrow \epsilon$

      If $X_1 \longrightarrow a|b$.
      First(X_1) = {a, b}.

      If $X_1 \longrightarrow a|b|\epsilon$
      First(α) = First(X_1) {ϵ}
      $\cup$ First($X_2 X_3$)

It will continue until production does not contain ϵ.
Finally if last X_i contains ϵ, then only we add ϵ to final list.

$S \longrightarrow ABCDE$
$A \longrightarrow a|\epsilon$
$B \longrightarrow b|\epsilon$
$C \longrightarrow c$
$D \longrightarrow d|\epsilon$
$E \longrightarrow e|\epsilon$

	First(α)
First(a) = a.	S a, b, c
First(b) = b.	A a, ϵ
First(c) = c	B b, ϵ
First(d) = d.	C c
First(e) = e.	D d, ϵ
	E e, ϵ

$S \rightarrow Bb \mid cd$
$B \rightarrow aB \mid \epsilon$
$C \rightarrow cC \mid \epsilon$

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

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

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

$S \rightarrow ACB \mid CbB \mid Ba$
$A \rightarrow da \mid BC$
$B \rightarrow g \mid \epsilon$
$C \rightarrow h \mid \epsilon$

First(C) = h, ε
First(B) = g, ε
First(A) = d, g, h, ε
First(S) = d, g, h, b, a, ε

Follow(A): Argument for follow is non terminal.
output of follow is set of all terminals that may follow immediately to the right of A in any sentential form.

Rules: 1. If A is start symbol, follow(A) = $.
2. If A is available as $X \rightarrow \alpha A \beta$ Follow(A) = First terminal in β
in G where α & β are strings of Grammar symbols then follow(A) = First(β)
3. If First(β) contains ε then Follow(A) = First(β) − ε and apply 4th rule.
4. If $X \rightarrow \alpha A$ or $X \rightarrow \gamma A \beta$ and $\beta \Rightarrow^{*} \epsilon$
Follow(A) = Follow(X)

$S \longrightarrow aABb$. First $(B) = d, \epsilon$.

$A \longrightarrow c \mid \epsilon$. First $(A) = c, \epsilon$.

$B \longrightarrow d \mid \epsilon$. First $(S) = a$

Follow $(B) = b$

Follow $(A) = d, b$

Follow $(S) = \$$.

$S \longrightarrow aBDh$ First $(F) = f, \epsilon$. Follow $(F) = h$.

$B \longrightarrow cC$ $E = g, \epsilon$. Follow $(E) = h$

$C \longrightarrow bC \mid \epsilon$. $D = g, f, \epsilon$ Follow $(D) = h, b$

$D \longrightarrow EF$. $C = b, \epsilon$. Follow $(C) = h$

$E \longrightarrow g \mid \epsilon$. $B = c$ Follow $(B) = h$.

$F \longrightarrow f \mid \epsilon$. $S = a$. Follow $(S) = \$$.

Follow $(F) = \$$. Follow $(F) =$ Follow (D)

Follow $(E) = g, f$ Follow $(E) =$ Follow (F)

Follow $(D) = h$. Follow $(D) = h$.

Follow . Follow $(C) =$ Follow (B)

Follow $(B) =$ First (Dh)

$= g, f, h$.

Constructing LL(1) parsing table.

For each production $A \longrightarrow \alpha$, repeat the following steps.

1. add $A \longrightarrow \alpha$ under $M[A, a]$ for each terminal a in First (α)
2. If First (α) contains ϵ, add $A \longrightarrow \alpha$ to $M[A, b]$ where b is in Follow (A)

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow id \mid (E)$

First(F) = id, (

First(T') = *, ϵ

First(T) = id, (

First(E') = +, ϵ

First(E) = id, (

Follow(E) =), $

Follow(E') = Follow(E) =), $

Follow(T) = +,), $

Follow(T') = +,), $

Follow(F) = *, +,), $

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

	+	*	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			$F \to id$	$F \to (E)$		

Construct LL(1) parsing table

$S \to (L) \mid a$

$L \to L, S \mid S$

$L \to SL'$

$L' \to L, L'$

$A \to A\alpha \mid \beta$

$A \to \beta A'$

$A' \to \alpha A' \mid \epsilon$

$S \to (L) \mid a$

$L \to SL'$

$L' \to , SL' \mid \epsilon$

	(	)	a	,	$
S	$S \to (L)$		$S \to a$		
L	$L \to SL'$		$L \to SL'$		
L'		$L' \to \epsilon$		$L' \to , SL'$	

$S \longrightarrow A$ $\quad A \longrightarrow Ad \mid aB$.

$A \longrightarrow aB \mid Ad \longrightarrow A \longrightarrow aBA'$

$B \longrightarrow bBC \mid f \quad A' \longrightarrow dA' \mid \epsilon$

$C \longrightarrow g$

	a	b	c	d	f	g	$
S	$S \longrightarrow A$						
A	$A \longrightarrow aBA'$						
A'				$A' \longrightarrow dA'$			$A' \longrightarrow \epsilon$
B		$B \longrightarrow bBC$			$B \longrightarrow f$		
C						$C \longrightarrow g$	

$S \longrightarrow AaAb \mid BbBa$

$A \longrightarrow \epsilon$

$B \longrightarrow \epsilon$

	a	b	$
S	$S \longrightarrow AaAb$.	$S \longrightarrow BbBb$.	
A	$A \longrightarrow \epsilon$	$A \longrightarrow \epsilon$.	
B.	$B \longrightarrow \epsilon$	$B \longrightarrow \epsilon$	

No two entries in LL(1) table

When we have more than one production in one entry grammar is not LL(1)

Possibilities of multiple entries:

A Grammar without ϵ rules is LL(1) if each production is of form $A \rightarrow \alpha_1 | \alpha_2 | \alpha_3 \ldots | \alpha_n$ and $First(\alpha_1) \cap First(\alpha_2) \cap First(\alpha_3) \ldots \cap First(\alpha_n) \neq \phi$

$First(\alpha_i) \cap First(\alpha_j) \neq \phi$. then it's not LL(1)

A Grammar with ϵ rules is LL(1) if each production is of form $A \rightarrow \alpha | \epsilon$ and $First(\alpha) \cap Follow(A) \neq \phi$ then LL(1) is not the Grammar type.

Find if following Grammar is LL(1) or not.

$S \rightarrow aSbS | bSaS | \epsilon$

not LL(1)

	a	b	$
S	$S \rightarrow aSbS$	$S \rightarrow bSaS$	
	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$ (clash)

Note: Any ambiguous grammar is not LL(1).

Any left recursive grammar is not LL(1)

$S \rightarrow aABb$

$A \rightarrow c | \epsilon$

$B \rightarrow d | \epsilon$

	a	b	c	d	$
S	$S \rightarrow aABb$				
A			$A \rightarrow c$		
B		$B \rightarrow \epsilon$		$B \rightarrow d$	
				$A \rightarrow \epsilon$	

It is LL(1).

$S \rightarrow A | a$ Ambiguous Grammar.

$A \rightarrow a$.

$S \rightarrow aB \mid \epsilon$ · a,
$B \rightarrow bC \mid \epsilon$ ·
$C \rightarrow cS \mid \epsilon$ ·

	a	b	c	$
S	$S \rightarrow aB$ ·			$S \rightarrow \epsilon$ ·

B $B \rightarrow b$ ·

C.

$S \rightarrow AB$ ·
$A \rightarrow a \mid \epsilon$ · a b, $
$B \rightarrow b \mid \epsilon$ · b $

$S \rightarrow aSA \mid \epsilon$ · a, c, $ ·
$A \rightarrow c \mid \epsilon$ · c, c, $

Not LL(1).

$S \rightarrow A$ ·
$A \rightarrow Bb \mid Cd$ · {a, b} ∩ {c, d}
$B \rightarrow aB \mid \epsilon$ · {a} ∩ {b}
$C \rightarrow cC \mid \epsilon$ · {c} ∩ {d}

It is LL(1)

$S \rightarrow aAa \mid \epsilon$ ⊗ Not LL(1)

$A \rightarrow abS \mid c$ $\{a\} \cap \{a\}$

$\{a\} \cap \{c\}$

Page No.
Date: / /

Program → begin d semi X end.

$X \rightarrow d \text{ semi } X \mid S\ Y$.

$Y \rightarrow \text{semi } S\ Y \mid \epsilon$.

begin d semi end. $. $

Program. Program→
begin d semiX end.

X $X \rightarrow d \text{ semi } X$. $X \rightarrow SY$.

Y. $Y \rightarrow \text{semi } SY$.

$Y \rightarrow \epsilon$.

$E \rightarrow aA \mid (E)$

$A \rightarrow +E \mid *E \mid \epsilon$.

a ( ) + * $.

E $E \rightarrow aA$ $E \rightarrow (E)$.

A $A \rightarrow +E$ $A \rightarrow *E$

$A \rightarrow \epsilon$ $A \rightarrow \epsilon$

$S \rightarrow iEtSS' \mid a$. i a.

$S' \rightarrow eS \mid \epsilon$ e.

$E \rightarrow b$. b.

NOT LL(1).

$A \longrightarrow AA \mid (A) \mid \epsilon$ Not suitable for predictive parsing because Grammar is ambiguous.

$A \longrightarrow A\alpha \mid \beta$

$A \longrightarrow \beta A'$

$A' \longrightarrow \alpha A' \mid \epsilon$

Consider the following C code

```
int main()
{
   int i, n;
   foo(i=0; i<n; i++);
}
```

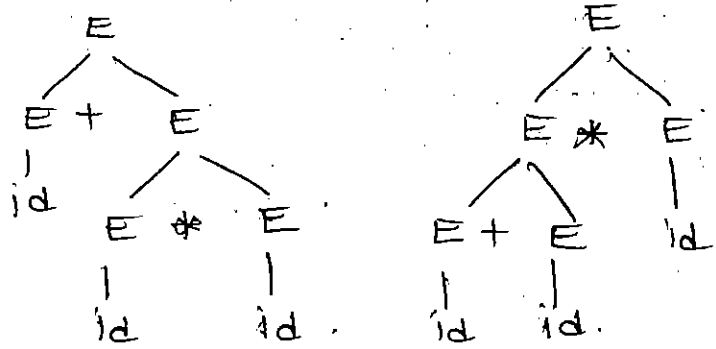
a. No compilation error
b. Compilation – only lexical error
c. Only syntax error ✓
d. Both (b) and (c)

Ambiguous to unambiguous:

$E \longrightarrow E+E \mid E*E \mid id$. G1

id + id * id.

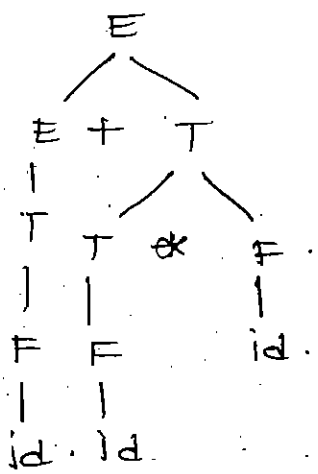
ambiguous grammar.



$E \longrightarrow E+T \mid T$.

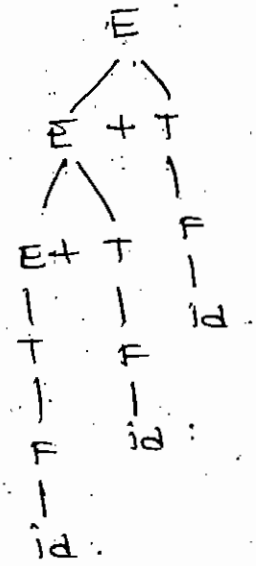
$T \longrightarrow T*F \mid F$.

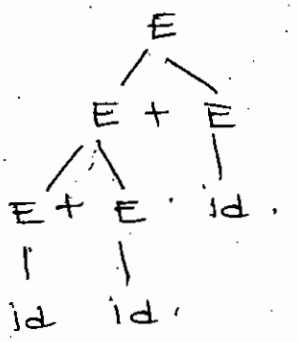
$F \longrightarrow id \mid (E)$.

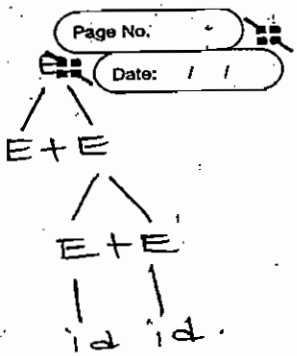


unambiguous grammar.

id + id + id.







If operator is left associative - define left recursive grammar.
If operator is right associative - use right recursive grammar
For precedence use least precedence at top level & highest precedence at lower level

Precedence and associativity is not taken care of in the grammar G, To impose precedence define productions starting with operator with lowest precedence to highest precedence.
For associativity, if operator is left associative then, define the production as left recursive. or if the grammar is right associative define the production as right recursive.

bExp → bExp and bExp. | bExp or bExp. | not bExp.
| true | false.

bExp → ~~not~~ X and X | X ~~or~~ X.
X → y and y | y or y.
y → true | false.

bExp → ~~bExp or~~ y | y
y → y and X | X.
X → not V | True | False
~~V → true | false~~

$R \rightarrow R+R \mid RR \mid R^* \mid a \mid b$.

③ ② ①

$+ < \cdot < *$.

$E \rightarrow E+T \mid T$.

$T \rightarrow TF \mid F$.

$F \rightarrow F^* \mid a \mid b$.

$A \rightarrow A\$B \mid B$.

$B \rightarrow B\#C \mid C$.

$D \rightarrow c@D \mid P$.

$D \rightarrow d$.

$\$ < \# < @ < d$.

$E \rightarrow E*F \mid F+E \mid F$.

$F \rightarrow F-F \mid id$.

* L to R.
+ R to L.

a. * has highest precedence than +.

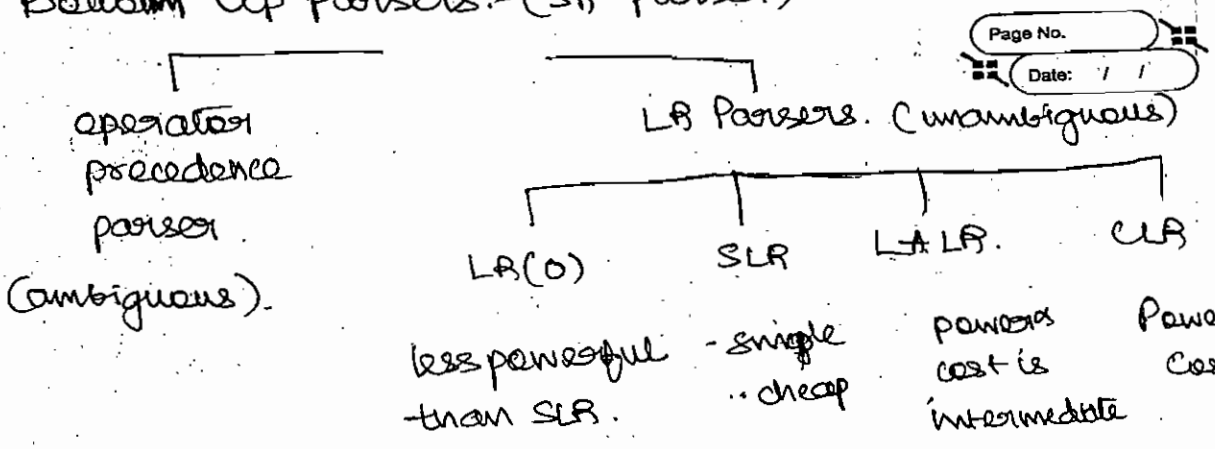
b. + has higher precedence than * ✓

c. * and - have same precedence.

d. + has higher precedence than +

if + > + then + is left associative.

Bottom Up Parsers:- (SR parser)



SR - shift reduce parsing

Operator precedence parser:

operator grammar: A Grammar that does not contain.

1. $\in$ rules. i.e no variable is nullable.
2. 2 adjacent non terminals on RHS of any production

ex: $E \longrightarrow EAE \mid id.$
$A \longrightarrow + \mid * \mid - \mid /$

not operator grammar
by expanding A we get op

$E \longrightarrow E+E \mid E*E \mid E-E \mid E/E \mid id.$

$S \longrightarrow SAS \mid a$
$A \longrightarrow bSb \mid b.$
$\Rightarrow$ $S \longrightarrow SbSbS \mid a. \mid SbS.$

Though the grammar is ambiguous, operator precedence parser uses, precedence relation table to construct the parse tree.

Precedence relation table contains 3 relations.

1. $<\cdot$ $\quad \cdot>$ $\quad \doteq$

	id	+	*	$
id		$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$

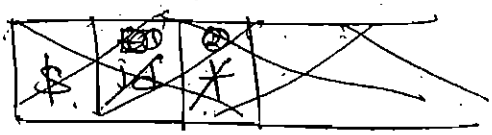
for operator precedence parser.

If 'a' is on the top of the stack and 'b' is lookahead symbol then.

1. if $a \lessdot b$ or $a \doteq b$ then shift b and increment input pointer.
2. if $a \gtrdot b$ repeat pop off stack until top of stack is related by $\lessdot$ to symbol recently popped
3. if $a \doteq b = \$$ successful completion.

If there is a blank entry & algorithm refers it then it is an error.

id + id + id $.



Construction of Precedence Table

1. If θ_1 has higher precedence than θ_2 then $\theta_1 \gtrdot \theta_2$
   $\theta_2 \lessdot \theta_1$ eg $* \gtrdot +$ $+ \lessdot *$.
2. If θ_1 and θ_2 are of equal precedence,
   a. and are left associative $\theta_1 \gtrdot \theta_2$ and $\theta_2 \gtrdot \theta_1$ ($+ \gtrdot +$ $- \gtrdot +$)
   b. and are right associative $\theta_1 \lessdot \theta_2$ and $\theta_2 \lessdot \theta_1$ ($\uparrow \gtrdot \uparrow$ $\uparrow \lessdot \uparrow$).
3. θ has less precedence than id. (any θ)
   $\theta \lessdot id$
   $id \gtrdot \theta$.

   θ has higher precedence than $ (any θ)
   $\$ \lessdot \theta$
   $\theta \gtrdot \$$

θ and '(' are right associative.

θ <· (

(<· θ

If θ and ')' are left associative.

P → SR | S.

R → bSR | bS.

S → WbS | W.

W → L*W | L.

L → id.

P → SbS | SbSbS.

S → WbS | L*W | id.

W → L*W | id.

SR = P

P → SbS | SbSbS.

~~S → WbS | L*W | id~~

P → SbS | SbSbS | W.

W → L*W | L.

L → id.

P → SbP | S.

	b	*	id	$
b	≐	<·	<·	·>
*	·>	<·	<·	·>
id	·>	·>		·>
$	<·	<·	<·	

	id	*	b	$
id		>	>	>
*	<	<	>	>
b	<	<	>	>
$	<	<	<	

since precedence relation table takes much space, parsers store precedence function table instead.

Converting relation table to function table.

1. create two symbols, f_a and g_a for each terminal
   a. partition these symbols into groups by using $\doteq$ relation i.e

eg: $a = b$

(f_a, g_b)

2. construct the digraph with groups as nodes and edges given by $\lessdot$ or $\gtrdot$

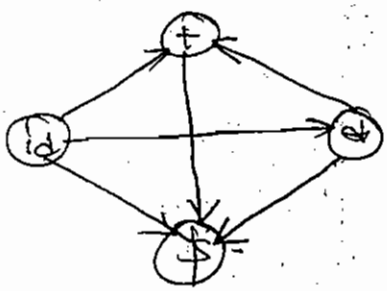
eg:

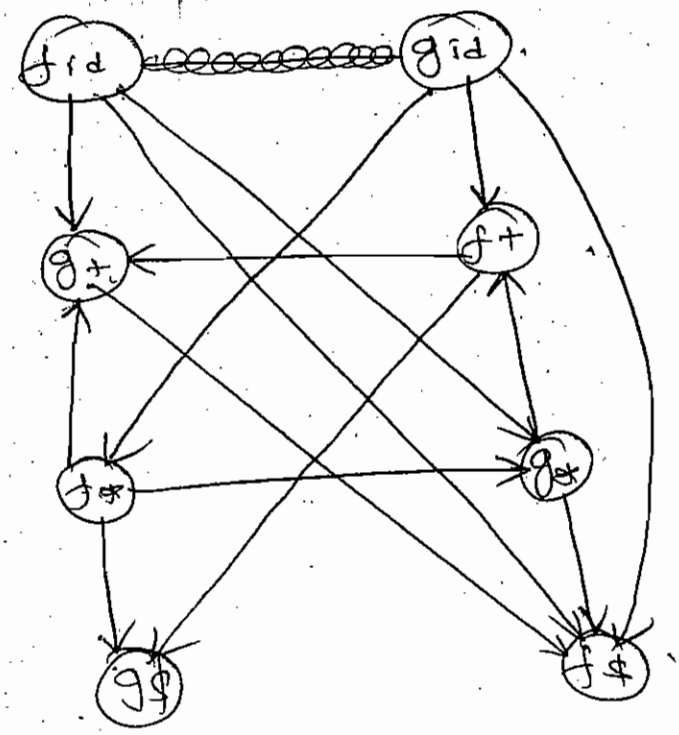
$a \lessdot b$ $\qquad$ $a \gtrdot b$.

Check for cycles in the digraph, if there is a cycle, stop the procedure and compute such a relation table cannot be reduced to function table.

If digraph is acyclic, length of the longest path from node (f_a), gives the value $f(a)$.

	id	*	+	$
id	—	⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	—





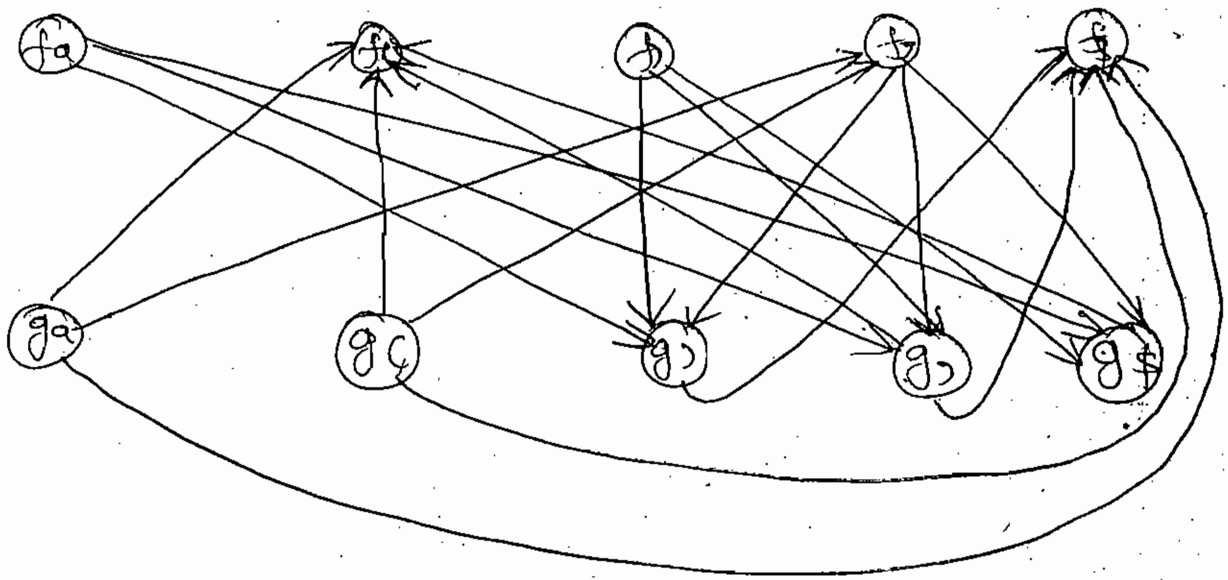
	id	+	*	$
f	4	2	4	0
g	5	1	3	0

If $f(a) > f(b)$ then $a \gtrdot b$.
$f(a) < f(b)$ then $a \lessdot b$.
$f(a) = f(b)$ then $a \doteq b$.

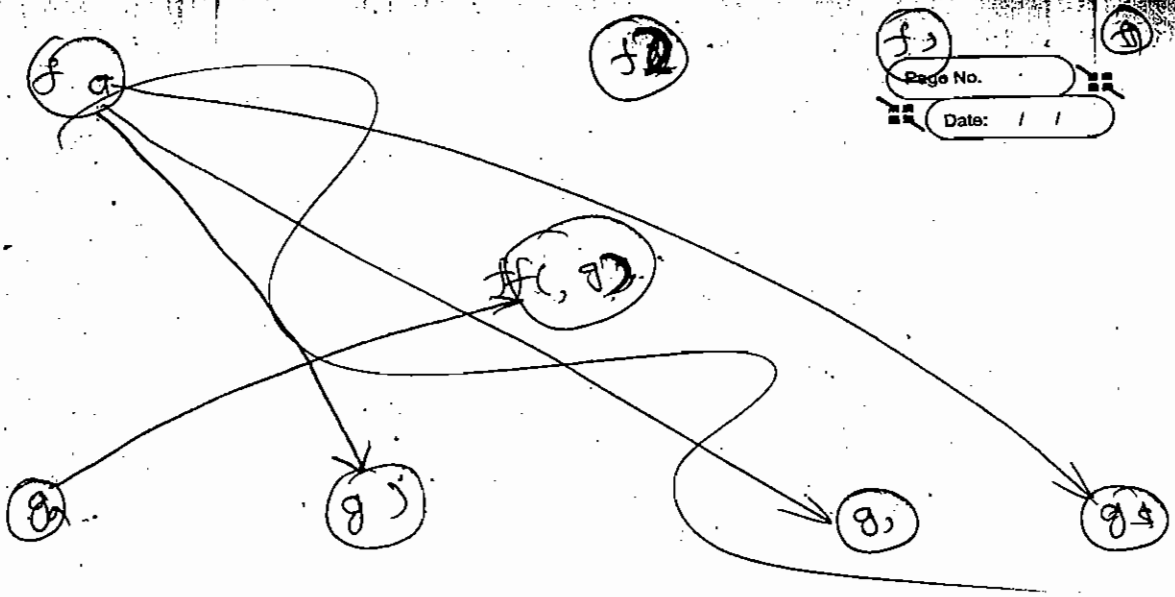
Disadvantage - There are no blank entries in function which will make it difficult for reporting errors.

	a	(	)	,	$
a			$\gtrdot$	$\gtrdot$	$\gtrdot$
(			$\doteq$	$\lessdot$	
)			$\gtrdot$	$\gtrdot$	$\gtrdot$
,			$\gtrdot$	$\gtrdot$	$\gtrdot$
$					

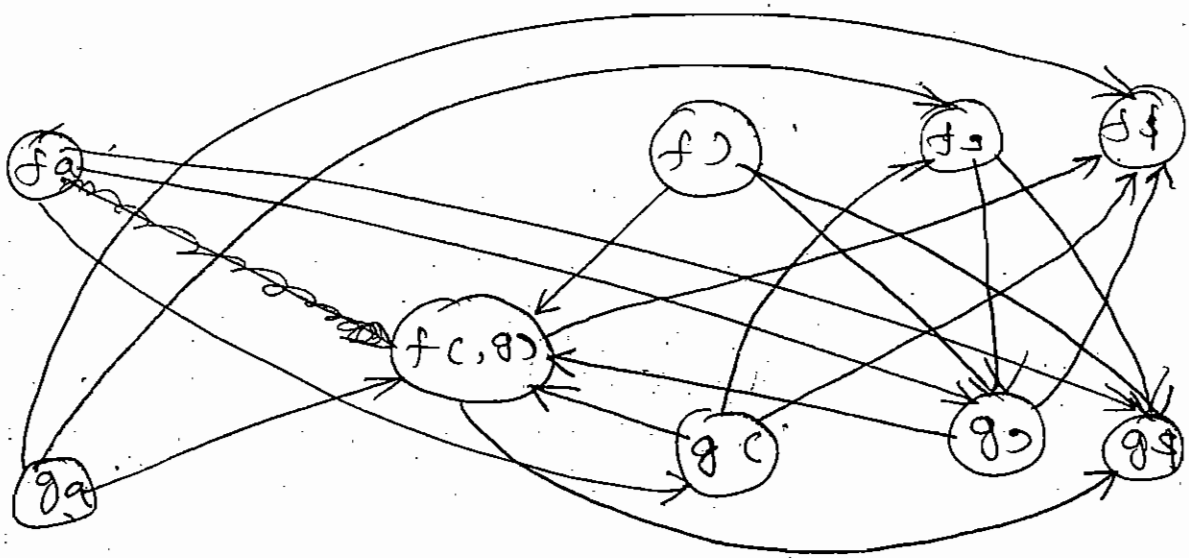
	a	(	)	,	$
a			·>	·>	·>
(	<·	<·	≐	<·	·>
)			·>	·>	·>
,	<·	<·	·>	·>	·>
$	<·	<·	<·	<·	∅



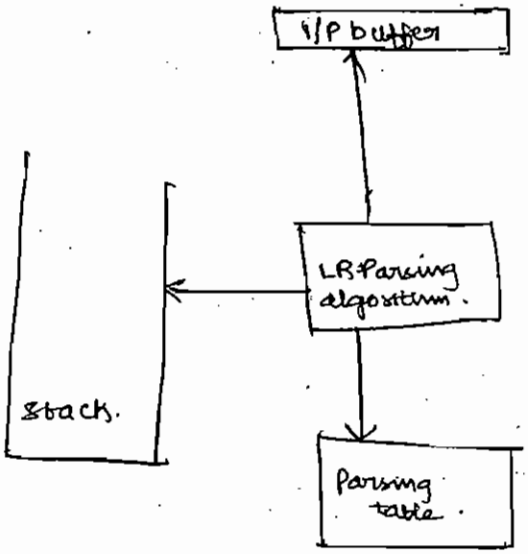
	a	(	)	,	$
f	2		2	2	0
g	3	3	1	1	0



$fa \rightarrow g_3$.



	a	(	)	,	$
f	4	1	2	2	0
g	3	3	3	2	0



Parsing Algorithm:

If X is top of the stack and a is lookahead symbol then.

1. action (X, a) = Si then shift a and state i onto the stack. & increment the input pointer.
2. If action(X, a) = Ri and ith production is A → β then fill the top of the stack with goto(Xi, a) after popping 2x mod β symbols from the stack. where Xi is state below A.

If action (X, a) = accept then it is successful completion.

example:

S → AA.

A → aA | b.

	a	b	$	S	A
				1	2.
0	S3	S4			
1			Accept		
2	S3	S4			
3	S3	S4			
4	r3	r3	r3		
5					

$S \rightarrow AA$

$A \rightarrow aA \mid b$.

	Action			Goto	
	a	b	$	S	A
0	S_2	S_4		1	2
1			accept		
2	S_3	S_4			5
3	S_3	S_4			6
4	γ_3	γ_3	γ_3		
5	γ_1	γ_1	γ_1		
6	γ_2	γ_2	γ_2		

S_4 → state number.

γ_2 → production no.

LR Parsing algorithm is same for all the LR parsers, LR(0), SLR, LALR, CLR. but parsing table is different for each parser.

Construction of LR parsing table. (general procedure used by all 4 LR parsers). Given a grammar take augmented grammar.

Create canonical collection of LR items

Draw DFA and construct table

LR(0) item: A production with dotted some point at RHS.

$A \rightarrow \cdot XYZ$

$A \rightarrow X \cdot YZ$

$A \rightarrow XY \cdot Z$.

$A \rightarrow XYZ \cdot$ Final item - item with dot in the end.

To construct a canonical collection, we use 2 functions closure and goto.

1. Closure(I) : input is LR(0) items, output LR(0) items initially add every item from I to closure(I). i.e add every input to output.

If A derives $\alpha . B\beta$ in I, and $B \to \gamma$ is in G, then add $B \to .\gamma$ to closure(I) repeat this process until no more items can be added to closure of I.

$E \longrightarrow E+T \mid T$

$T \longrightarrow T*F \mid F$

$F \longrightarrow id$

$S \longrightarrow E$

$E \longrightarrow E+T \mid T$

$T \longrightarrow T*F \mid F$

$F \longrightarrow id$

closure($S \to .E$) = { $S \to .E$,
$E \to .E+T \mid .T$
$T \to .T*F \mid .F$
$F \to .id$

$S \to AA$

$A \to aA \mid b$

$S' \to S$
$S \to AA$
$A \to aA \mid b$

closure($S' \to .S$) = { $S \to .AA$
$A \to .aA \mid .b$ }

Goto(I, X)

closure($A \to \alpha X . \beta \mid A \to \alpha . X\beta$ is in I)

1. Find transition
2. Apply closure.

$E' \to .E$
$E \to .E+T \mid .T$
$T \to .T*F \mid .F$
$F \to .id$

I

Goto(I, T) = $E \to T.$
$T \to T.*F$

closure($E \to T.$
$T \to T.*F$)

Goto (I_i, X) =

constructing canonical collection C

To i.e. closure of augmented production with a . at the beginning

repeat
  for each grammar symbol X and for each
    I_i in C which Goto (I, X) is not empty
    and not in C
    add Goto (I_i, X) to C

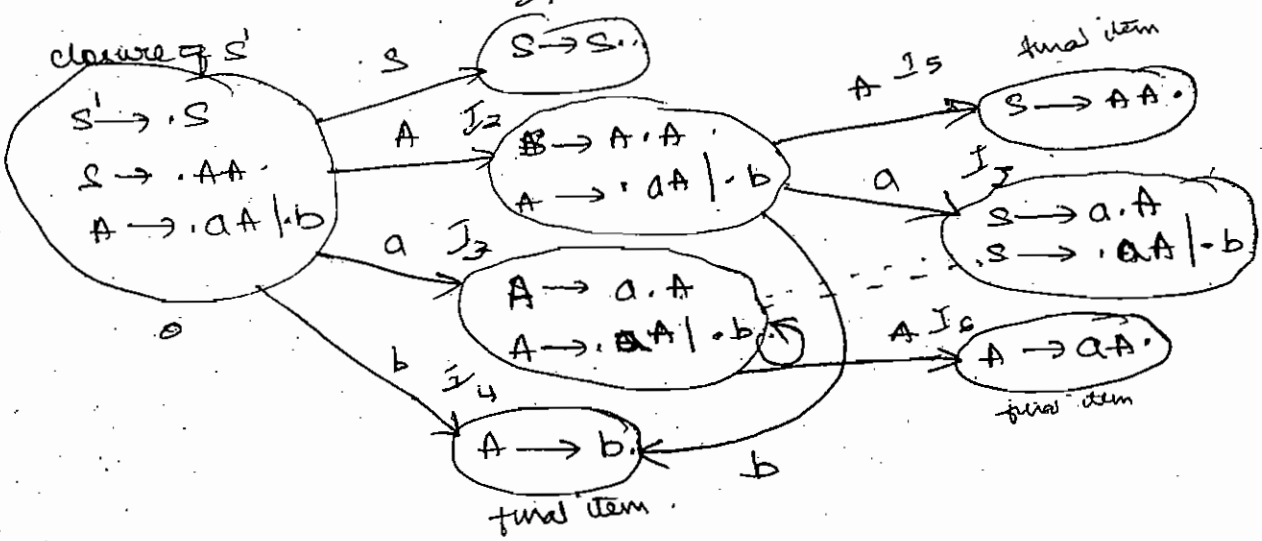
until no more new items can be added to C

Find the canonical collection for the following Grammar

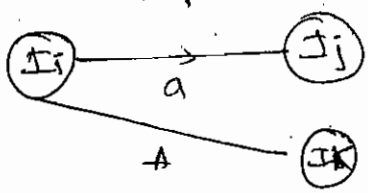
$S \rightarrow AA$
$A \rightarrow aA \mid b$

$S' \rightarrow S$
$S \rightarrow AA$
$A \rightarrow aA \mid b$

canonical collection of G.



constructing parsing table.



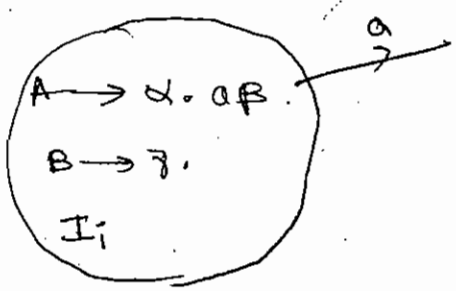
I	action a	Goto A
i	S_j	k
j		
k		

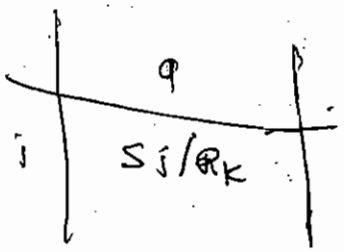
same for all parsers.

LR(0) parser it uses 0 lookaheads in taking the parsing decision. Here every row is pure shift or reduce

	a	b	$	S	A
0	S_3	S_4		1	2
1	Acc	Acc	Acc		
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		

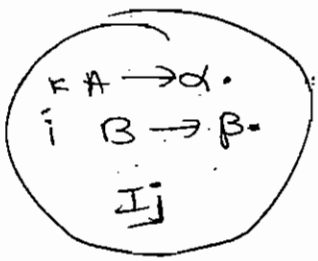
Checking whether a Grammar is LR(0) or not



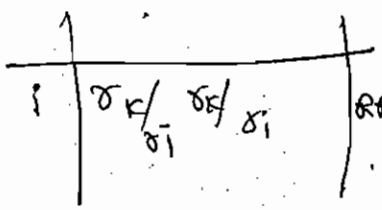


SR conflict

RR



presence of two item causing conflict.



RR conflict

Page No.
Date: / /

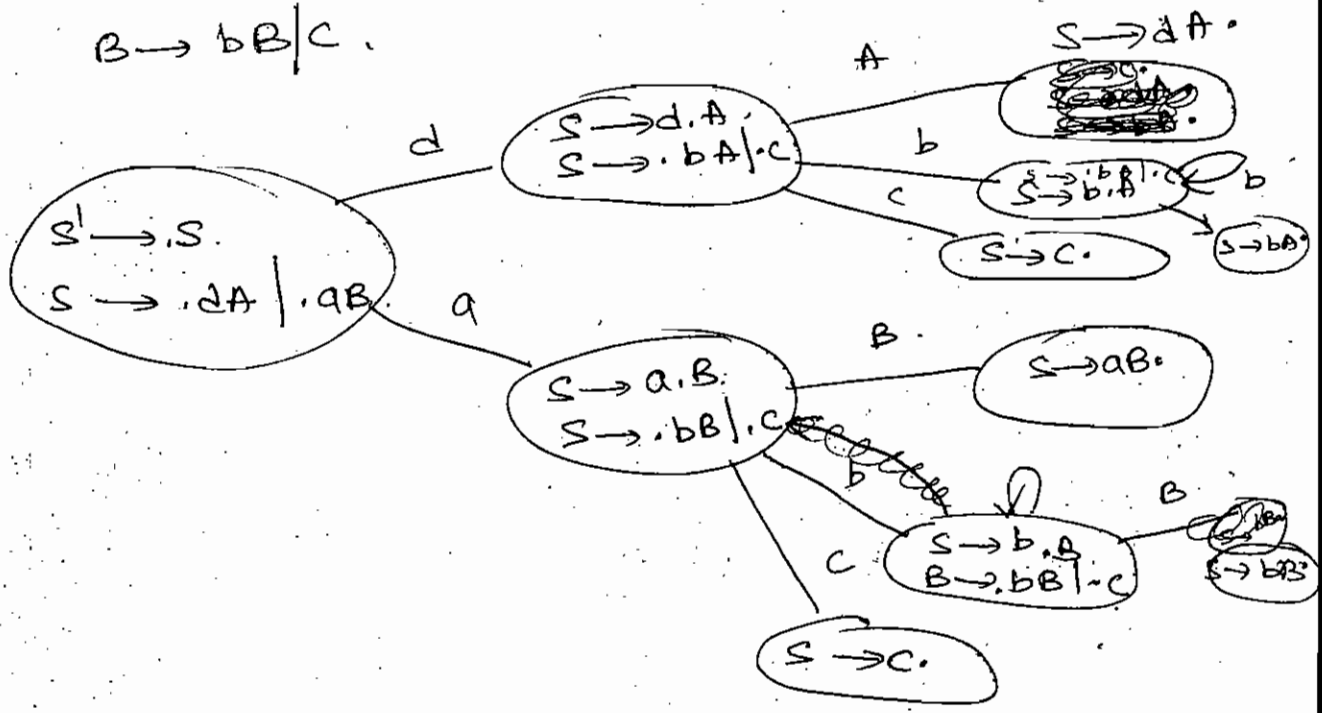
Find whether following Grammar is LR(0) or not.

$S' \rightarrow S$.

$S \rightarrow dA \mid aB$.

$A \rightarrow bA \mid c$.

$B \rightarrow bB \mid c$.

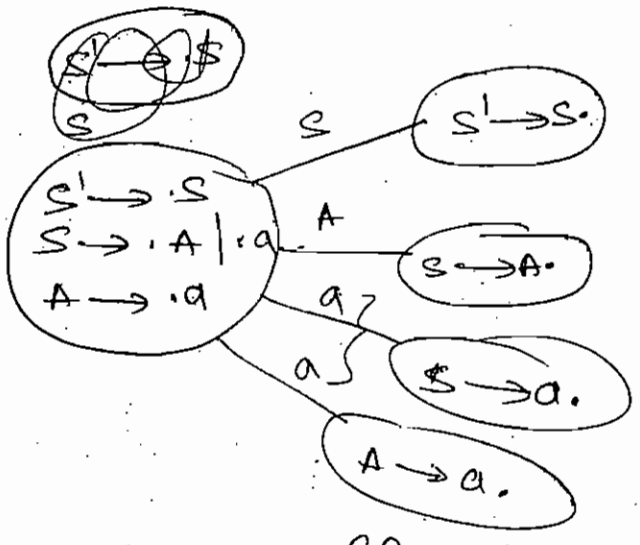


LR(0) Grammar.

$S' \rightarrow S$.

$S \rightarrow A \mid a$.

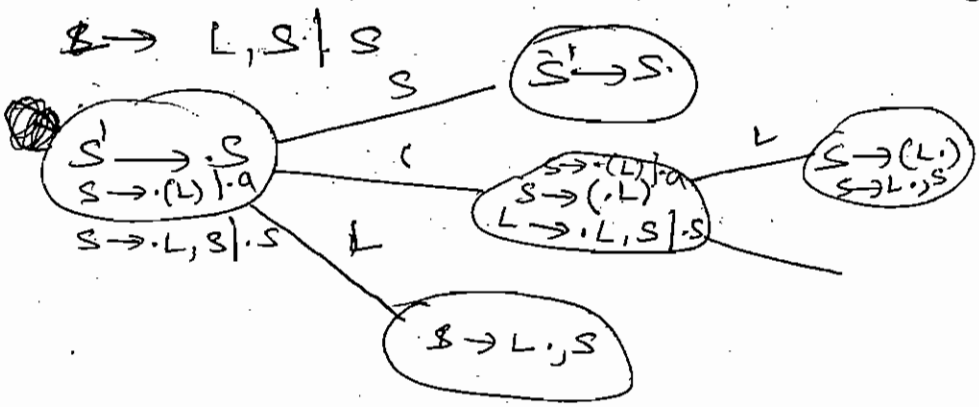
$A \rightarrow a$.



RR conflict.

$S \rightarrow (L) \mid a$

$L \rightarrow L,S \mid S$

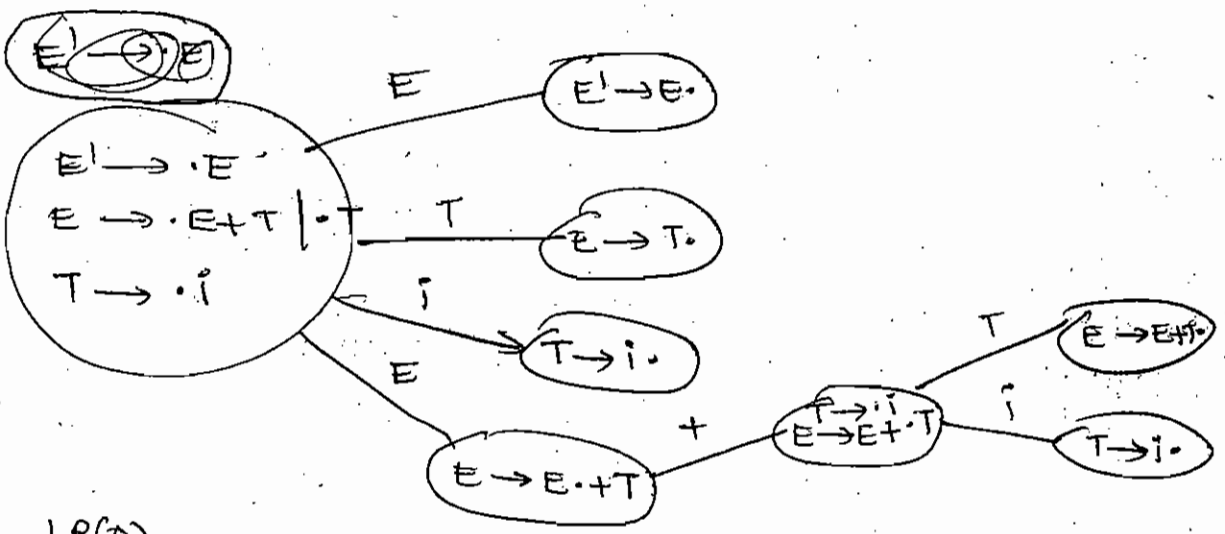


LR(0)

$E' \rightarrow E$

$E \rightarrow E+T \mid T$

$T \rightarrow i$

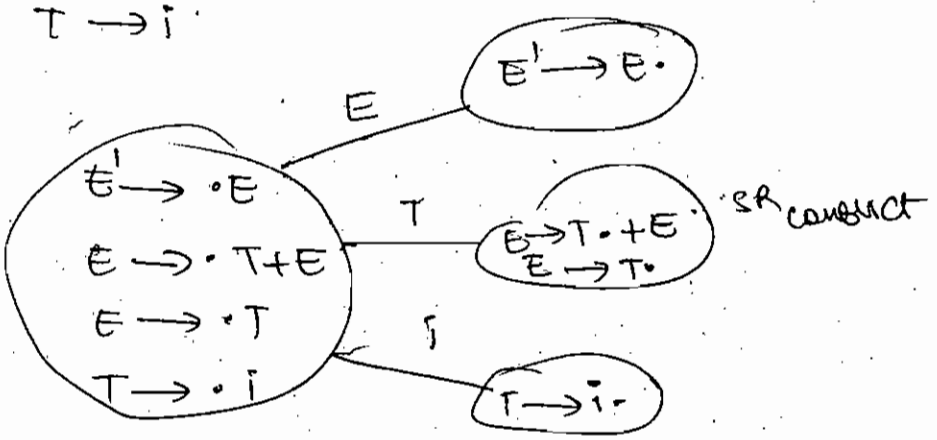


LR(0)

$E \rightarrow E$

$E \rightarrow T+E \mid T$

$T \rightarrow i$



	i	+	$	E	T
0	S_3			1	2
1					
2	r_2	S_4/r_2	r_2		
3	r_3	r_3	r_3		
4	S_3			5	2
5	r_1	r_1	r_1		

SLR(1)

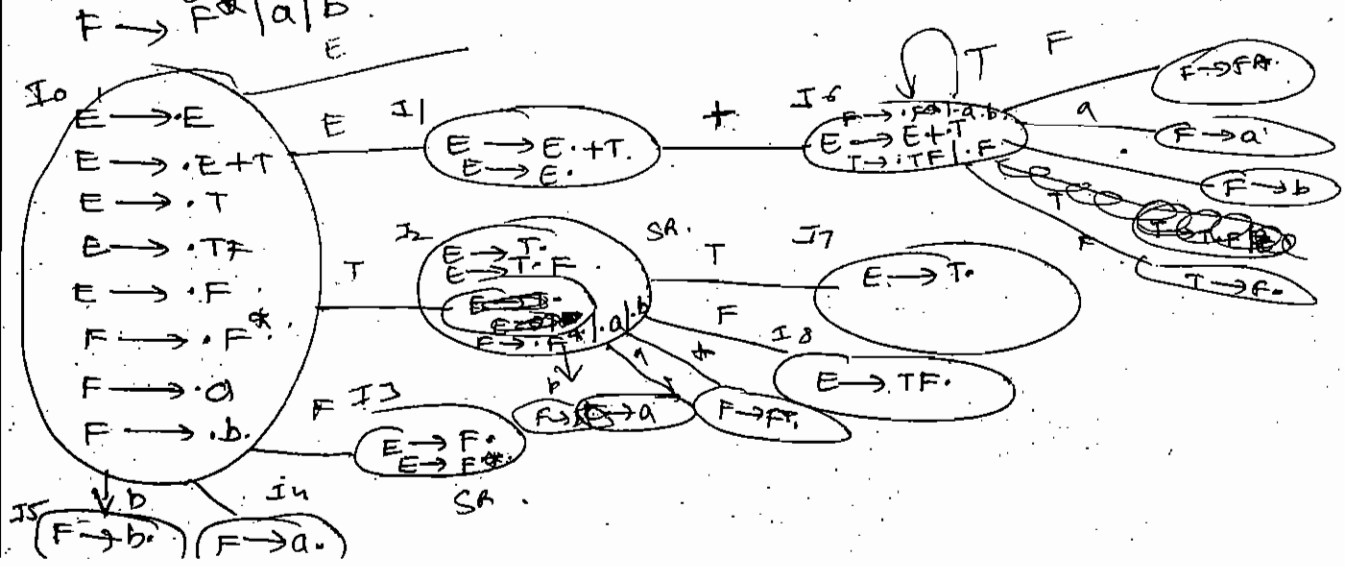
Same as LR(0) except that if there is any final item in a state instead of placing reduce action in the entire row, we place reduce action only in the Follow (LHS) of final item.

Follow (T) = +, $.

	i	+	$	E	T
0	S_3			1	2
1					
2		S_4	r_2		
3		r_3	r_4		
4				5	2
5			r_1		

States with conflict are inadequate

$E' \to E$ ①
$E \to E+T \mid T$ ② ③
$T \to TF \mid F$ ④ ⑤
$F \to F^* \mid a \mid b$ ⑥ ⑦ ⑧



State	a	b	$	S	A	B
I_2	S_4	S_5	r_2	r_2		
I_3	r_4	r_4	S_8	r_4	r_4	
I_{11}	r_9	r_3	S_8	r_3	r_3	
I_9	S_4	S_5		r_1	r_2	

it is SLR(1)

P272 4.37

$S' \longrightarrow S$

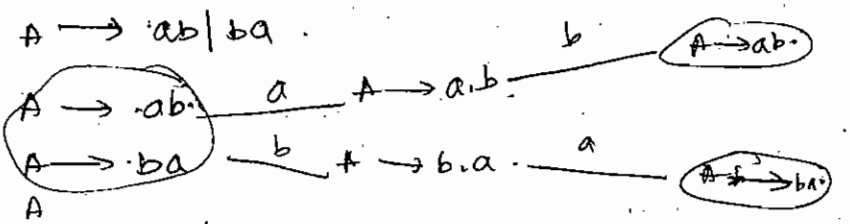
$S \longrightarrow AaAb \mid BbBa$

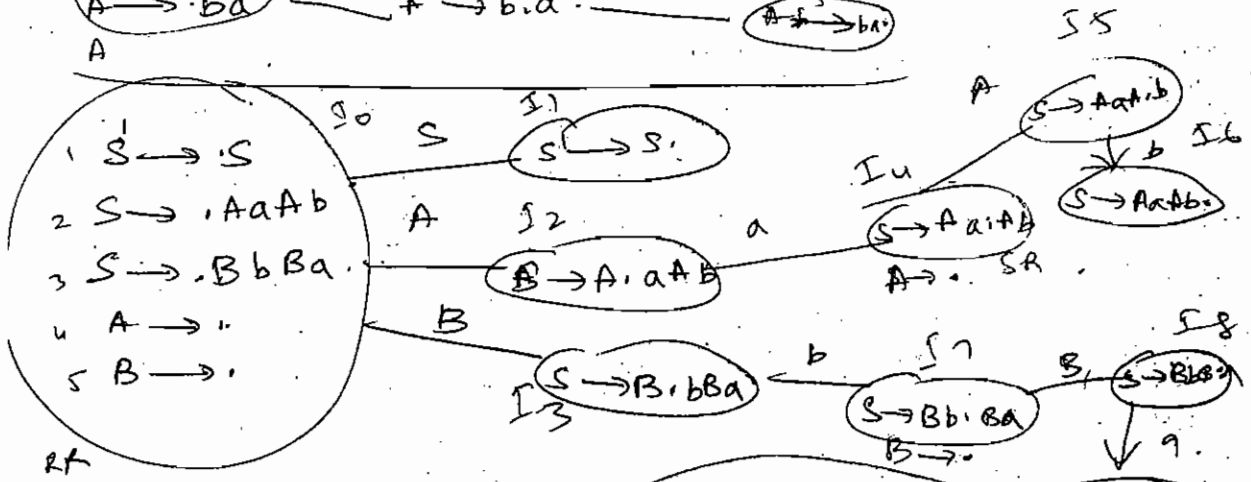
$A \longrightarrow \in$

$B \longrightarrow \in$

$A \longrightarrow Aab \mid aAb \mid ab \mid bBa \mid Bba \mid ba$

$A \longrightarrow ab \mid ba$



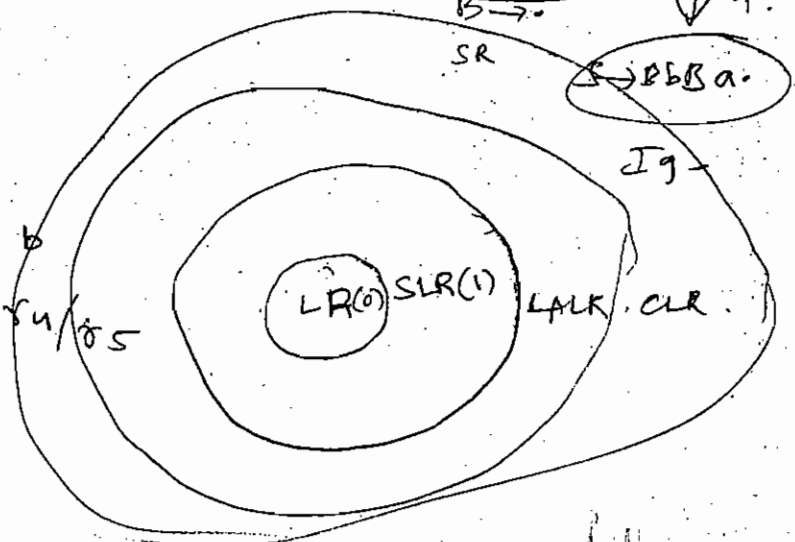


Follow(B) = b, $

Follow(A) = a, b.

	a	b
0	r_4/r_5	r_4/r_5

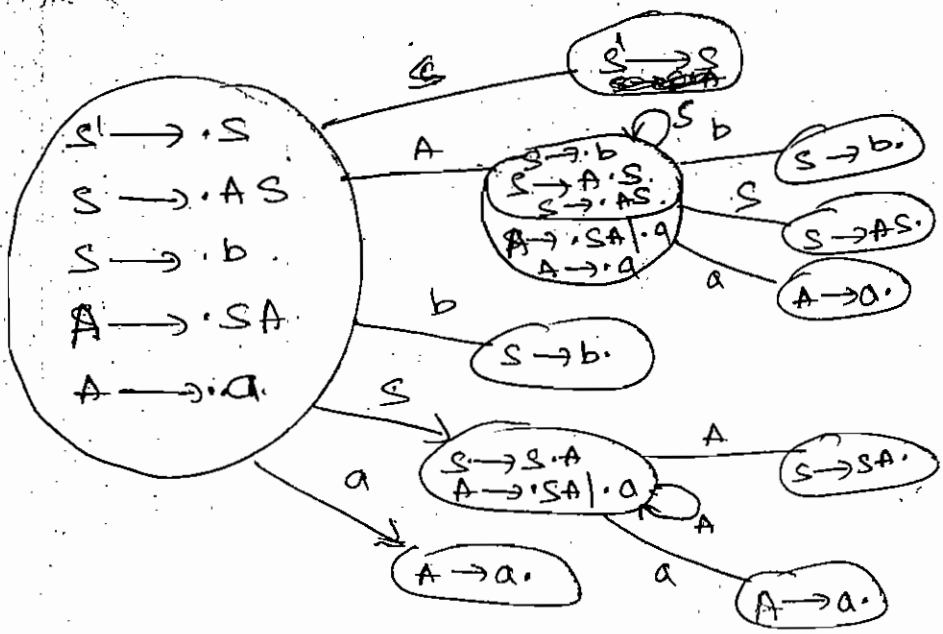
Not SLR(1)

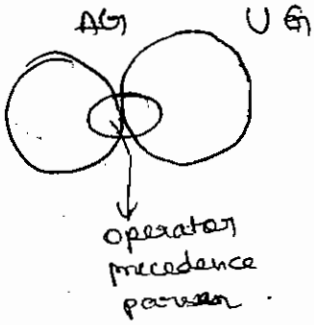


$S' \longrightarrow S$

$S \longrightarrow AS \mid b$.

$A \longrightarrow SA \mid a$.





NOT SLR(1)

pumping lemma.

$S \rightarrow Aa \;|\; bAc \;|\; dc \;|\; bda$.

① ② ③ ④

$A \rightarrow d$.

⑤

~~da~~ bdc dc bda.

d(a+c) bd(c+a).

(d + bd)(a+c)

Not LL(1).

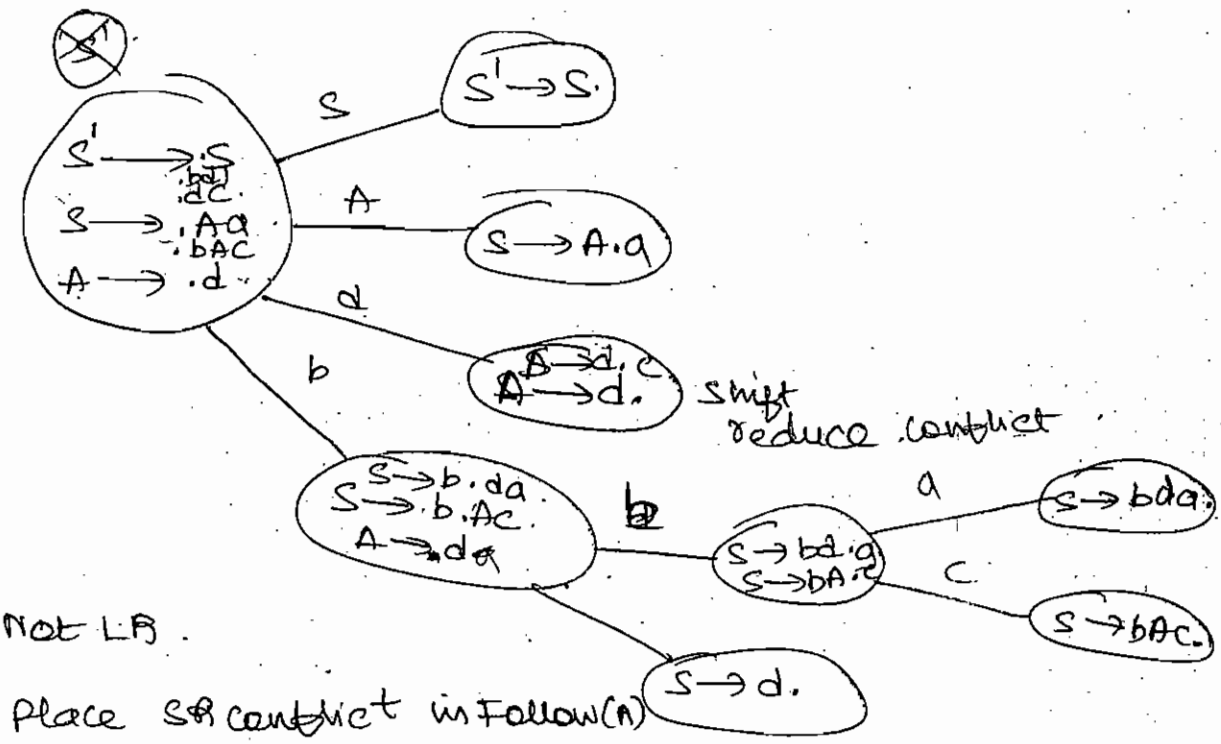
	a	b	c	d
S		②/④	①/③/⑤	
A				

∴ not LL(1).

LR(0).

$S' \rightarrow S$.	0
$S \rightarrow Aa$.	1
$S \rightarrow bAc$.	2
$S \rightarrow dC$.	3
$S \rightarrow bda$.	4
$A \rightarrow d$.	5

Y/S.



Not LR.

Place SR conflict in Follow(A)

= a, c.

Not SLR.

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

redefine closure and goto functions to find canonical collection of LR(1) items.

closure(I) =

1. Initially add every item from I to closure(I)

If A derives $A \to \alpha . B\beta, \$$ is in I and $B \to \gamma$ is in G, then add $B \to .\gamma$ to I, First($\beta\$$)

If $A \to \alpha . B\beta\$$ add

$B \to .\gamma$, First($\beta\$$)

repeat this process for every newly added item.

$S \to AA$

$A \to aA \mid b$

$S' \to S$

$S' \to .S, (\$)$

$S \to .AA, (a/b)\$$

$A \to .aA \ (a,b)$

$A \to .b \ (a,b)$

$S' \to .S, \$$

$S \to .AA, \$$

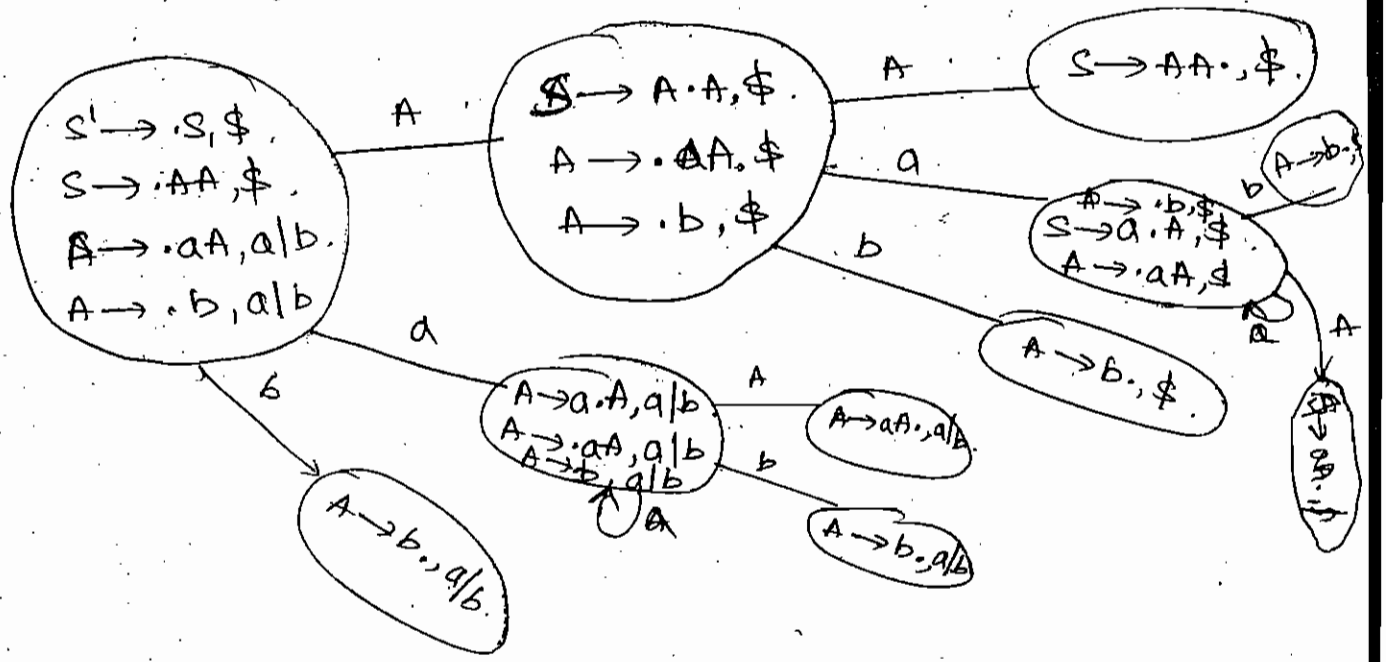
$A \to .aA, a/b$

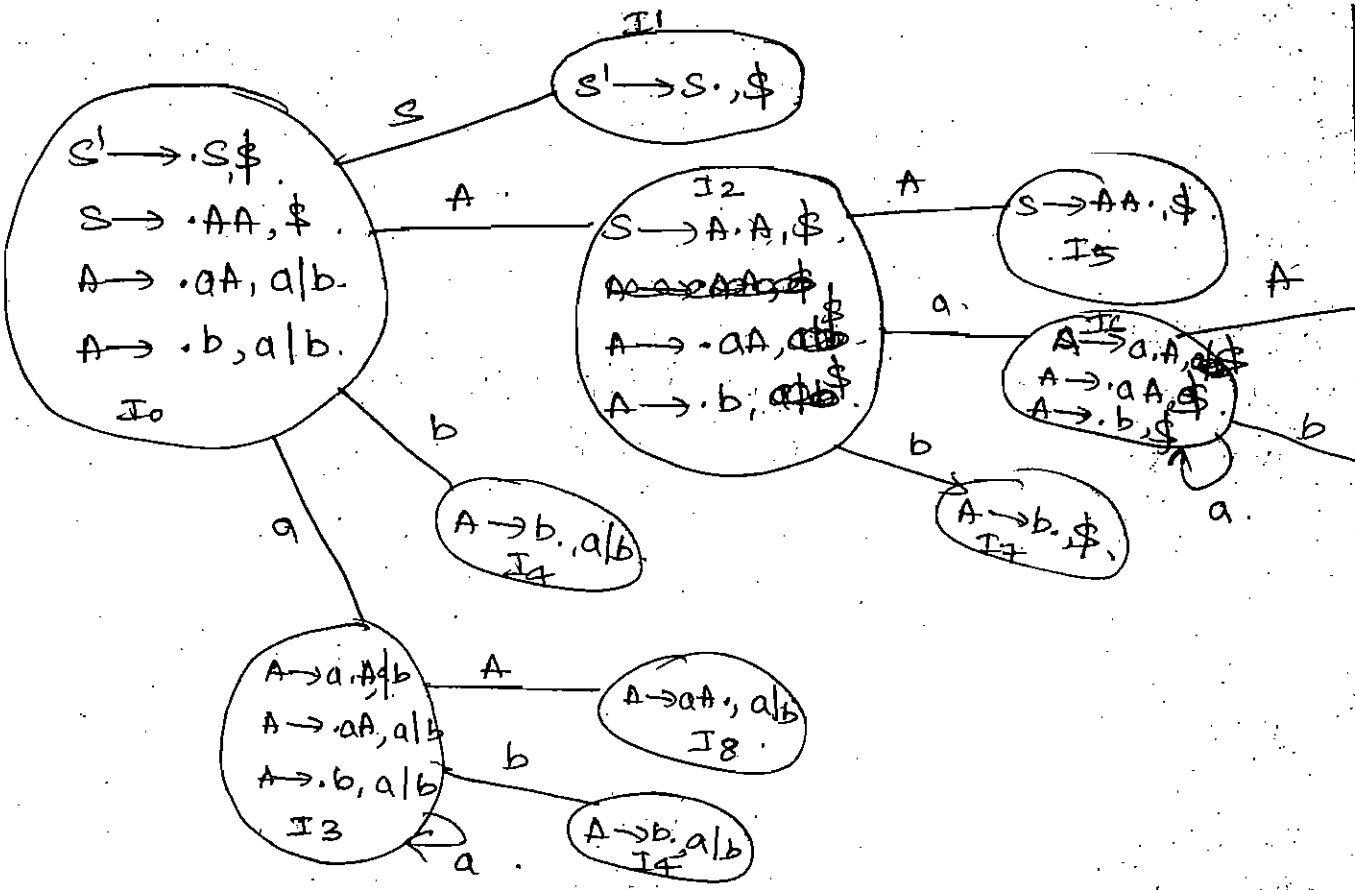
$A \to .b, a/b$

Goto[I, X]

while finding the transition, lookahead remains constant.

while applying closure, lookahead may change.





CLR(1)

	a	b	$	S	A
I_0	S_3	S_4		1	2
I_1			r_0		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	r_3	r_3			
I_5			r_1		
I_6	S_6	S_7			
I_7			r_3		9
I_8	r_2	r_2	r_3		
I_9			r_2		

SLR(1) table same as LR(0) and SLR(1) except that reduce moves will be placed only in the lookaheads of final item.

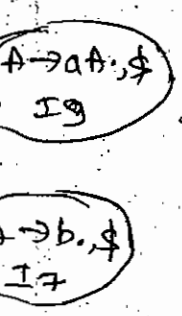
LALR(1) Parsing table.

Page No.
Date: / /

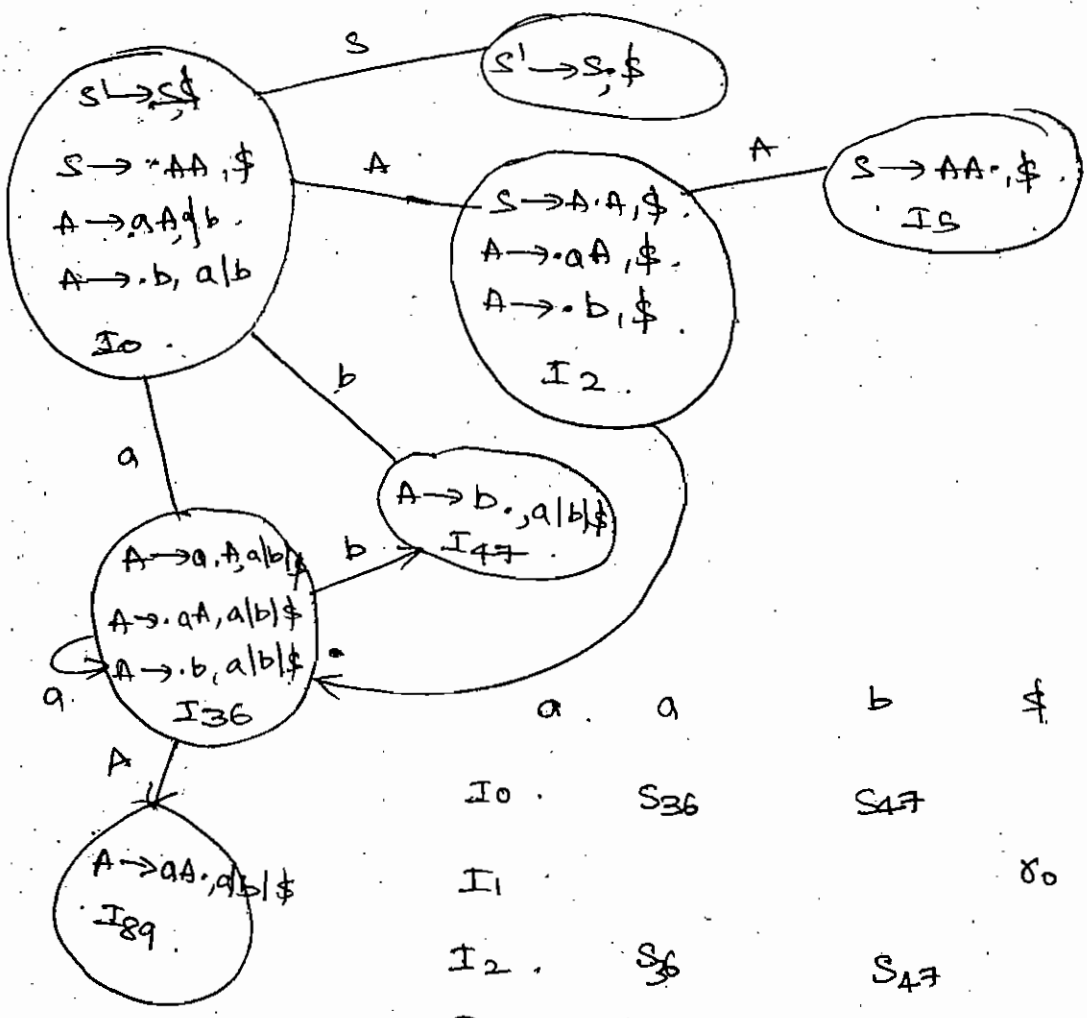
Reduce the number of states or rows, by combining two states which differ only in lookahead.

merge lookaheads.

In the above example, (I_3, I_6), (I_4, I_7) (I_8, I_9)



I_{36}, I_{47} I_{89}.



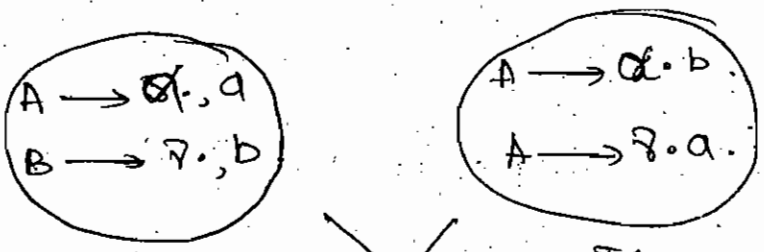
	a	a	b	$	S	A
I_0		S_{36}	S_{47}		1	2
I_1				γ_0		5
I_2		S_{36}	S_{47}			89
I_{36}		S_{36}	S_{47}			
I_{47}		γ_3	γ_3	γ_3		
I_5				γ_1		
I_{89}		γ_2	γ_2	γ_2		

$I_i \xrightarrow{a} I_j$

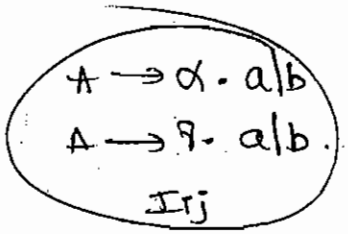
j^{th} row in 'a' column place S_j

Though there are no RR conflicts in CLR(1), still, there may be RR conflicts in LALR(1).

example:



Iij	a	b
	r_1/r_2	r_1/r_2



RR conflict

If there are no SR conflicts in CLR(1), there will be no SR conflicts in LALR(1) also.

$S \rightarrow AaAb$ ①

$S \rightarrow BbBa$ ②

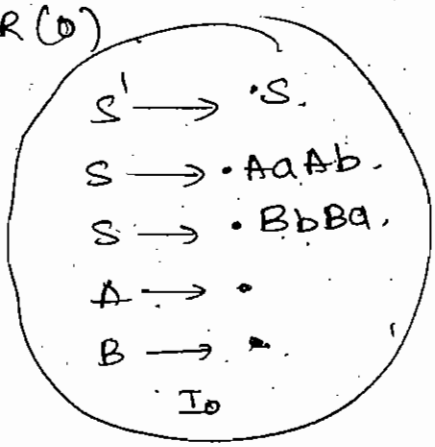
$A \rightarrow \epsilon$ ③

$B \rightarrow \epsilon$ ④

LL(1)

	a	b	ϵ
S	①	②	

LR(0)



I_0	a	b
	r_3/r_4	r_3/r_4

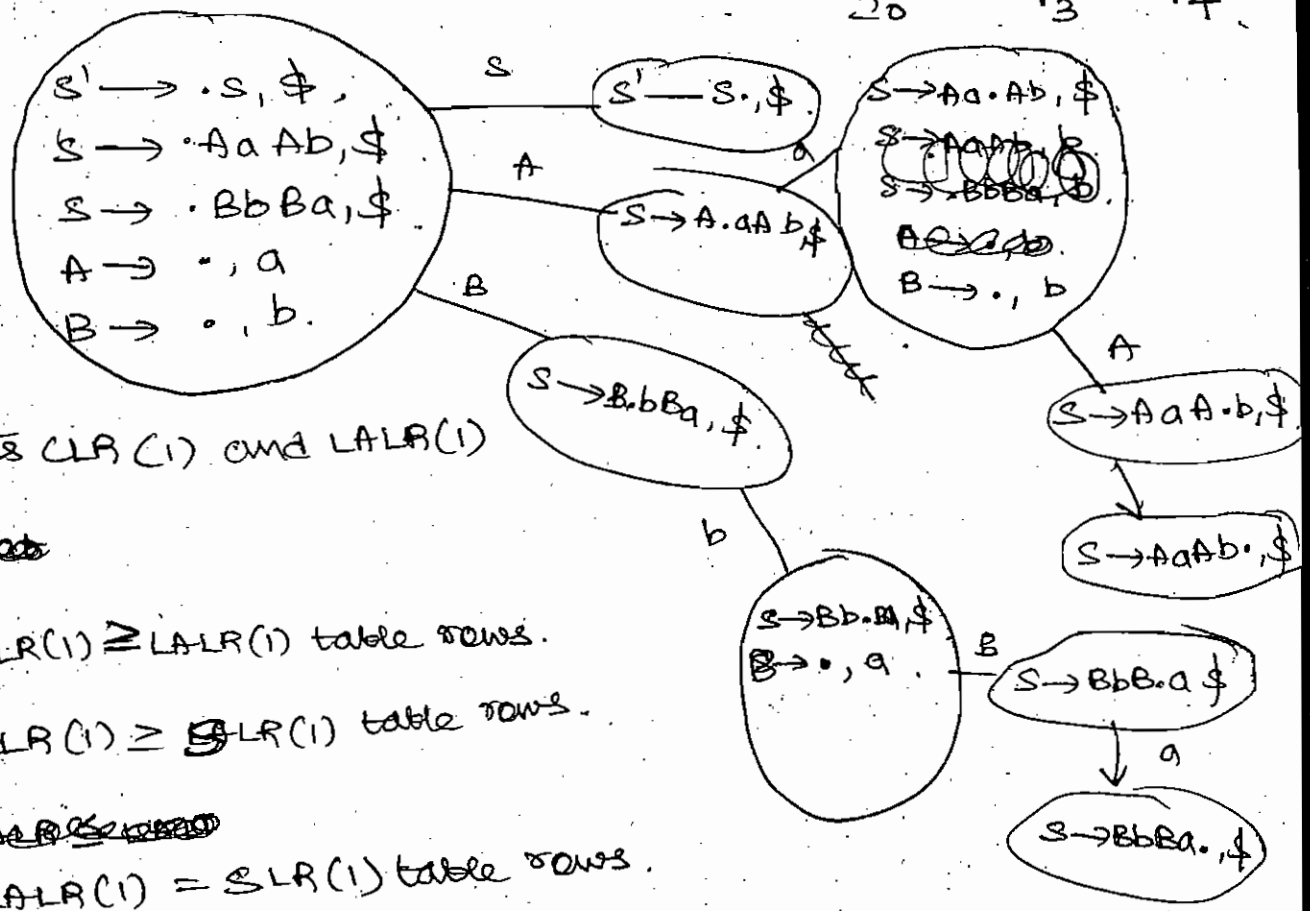
SLR(1)

	a	b
I_0	r_3/r_4	r_3/r_4

Page No.
Date: / /

not SLR(1)

	a	b
I_0	r_3	r_4



LR(1) and LALR(1)

CLR(1) ≥ LALR(1) table rows.

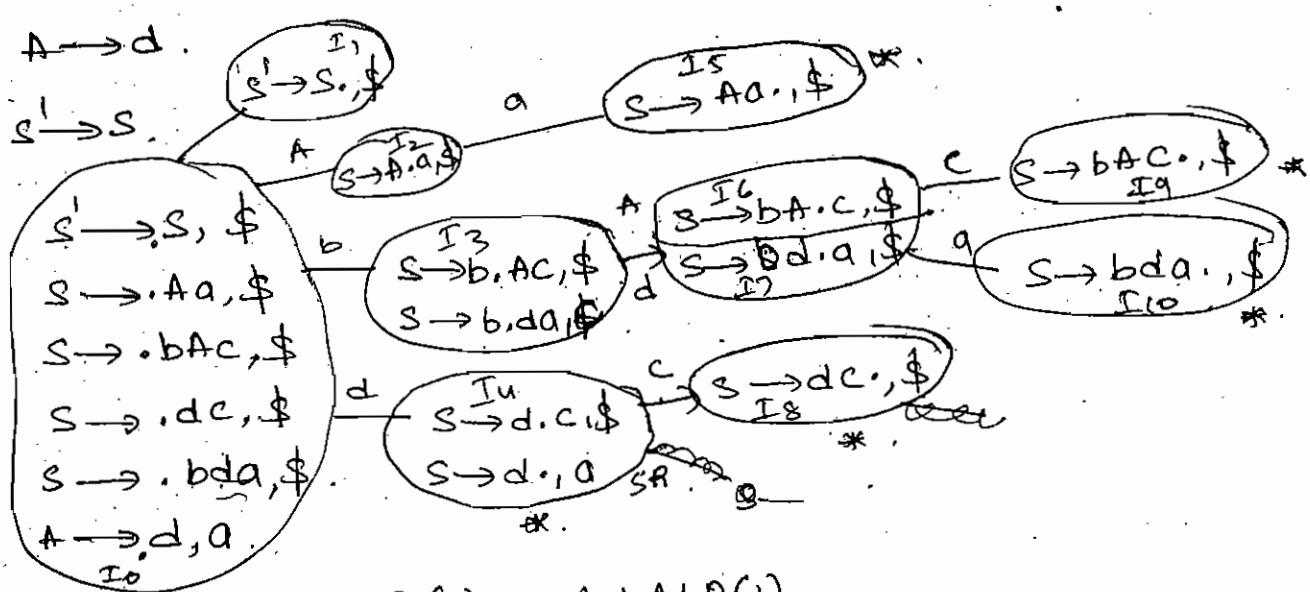
CLR(1) ≥ SLR(1) table rows.

LALR(1) = SLR(1) table rows.

S → Aa | bAc | dc | bda.

A → d.

S' → S.

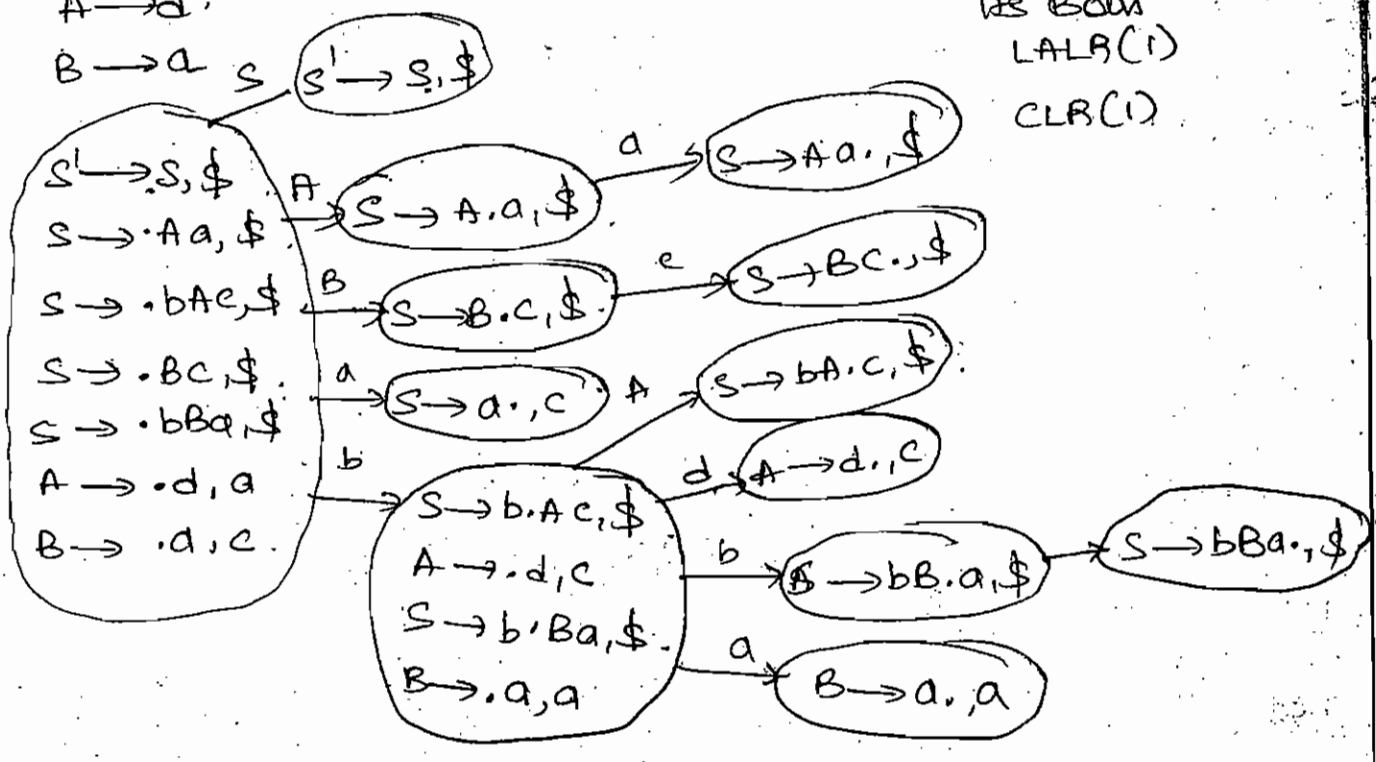


CLR(1) and LALR(1)

$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$.

$A \rightarrow d$.

$B \rightarrow a$

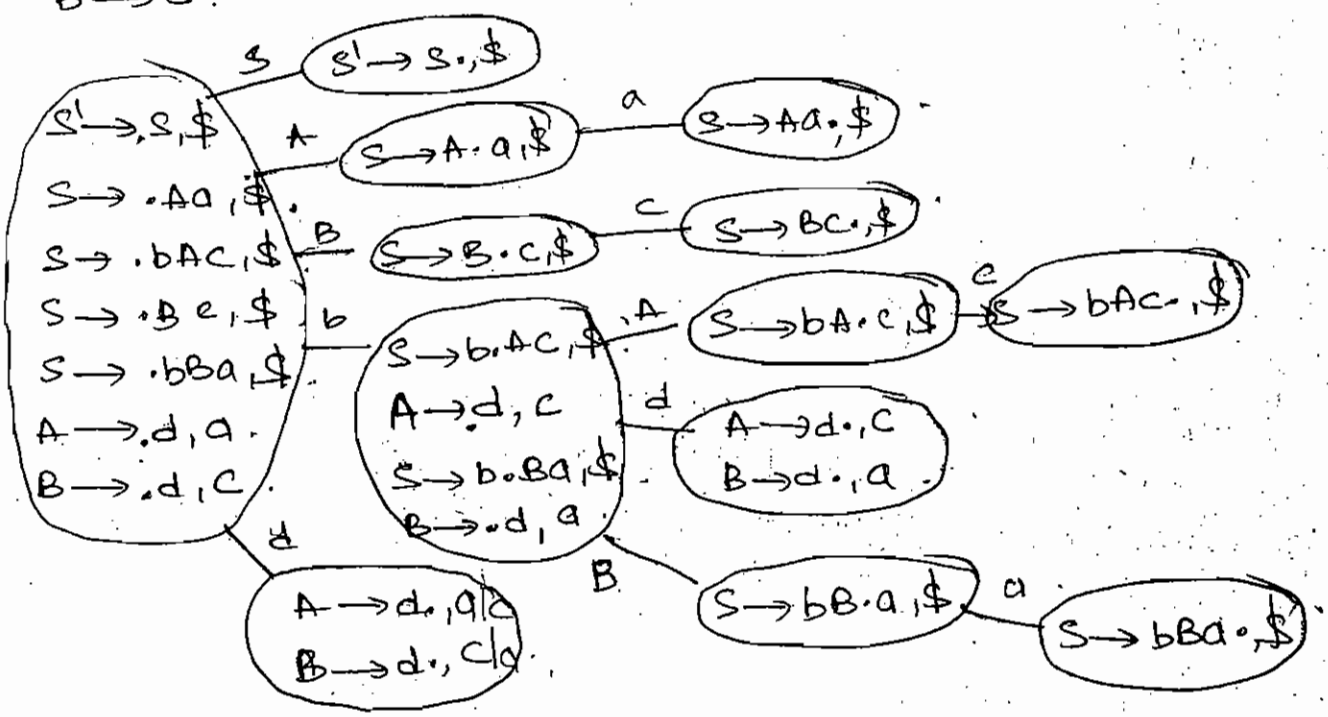


its BOTH
LALR(1)
CLR(1)

$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$.

$A \rightarrow d$

$B \rightarrow d$.



CLR not LALR.

$S \rightarrow A$

$A \rightarrow AB \mid \epsilon$

$B \rightarrow aB \mid b$

$A \rightarrow A\alpha \mid \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A \mid \epsilon$

Page No.

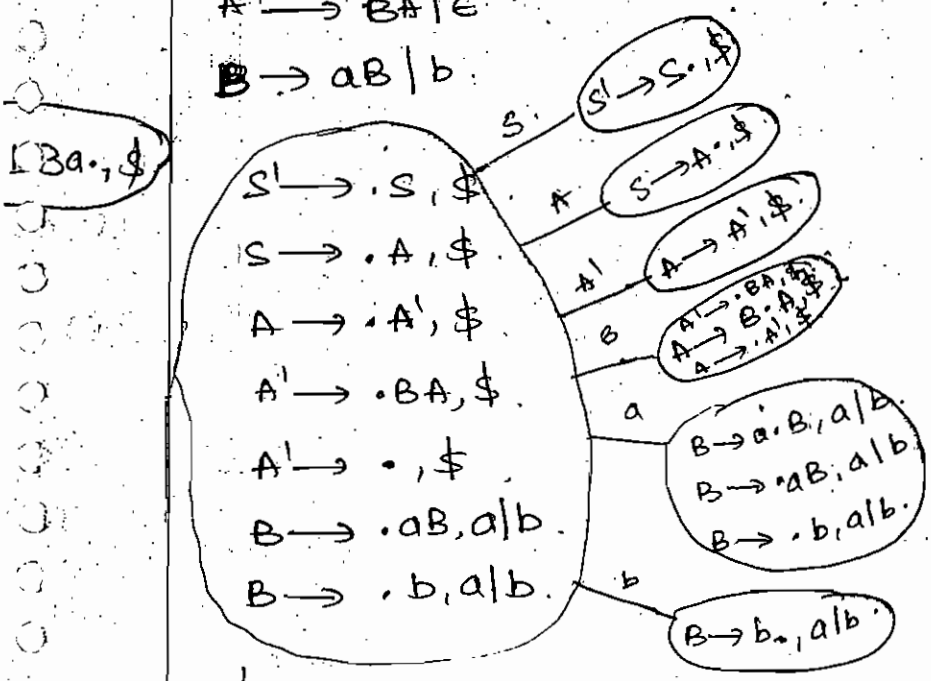
Date: / /

$S \rightarrow A$ $\quad \alpha = B$

$A \rightarrow A'$ $\quad \beta = \epsilon$

$A' \rightarrow BA \mid \epsilon$

$B \rightarrow aB \mid b$

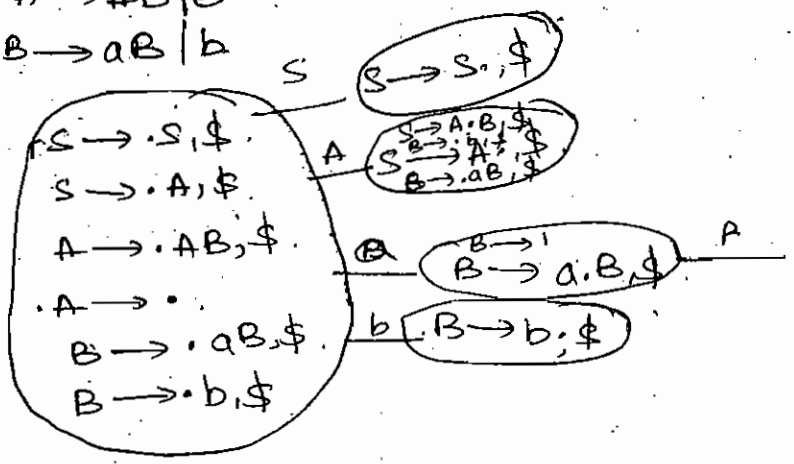


$S' \rightarrow S$

$S \rightarrow A$

$A \rightarrow AB \mid \epsilon$

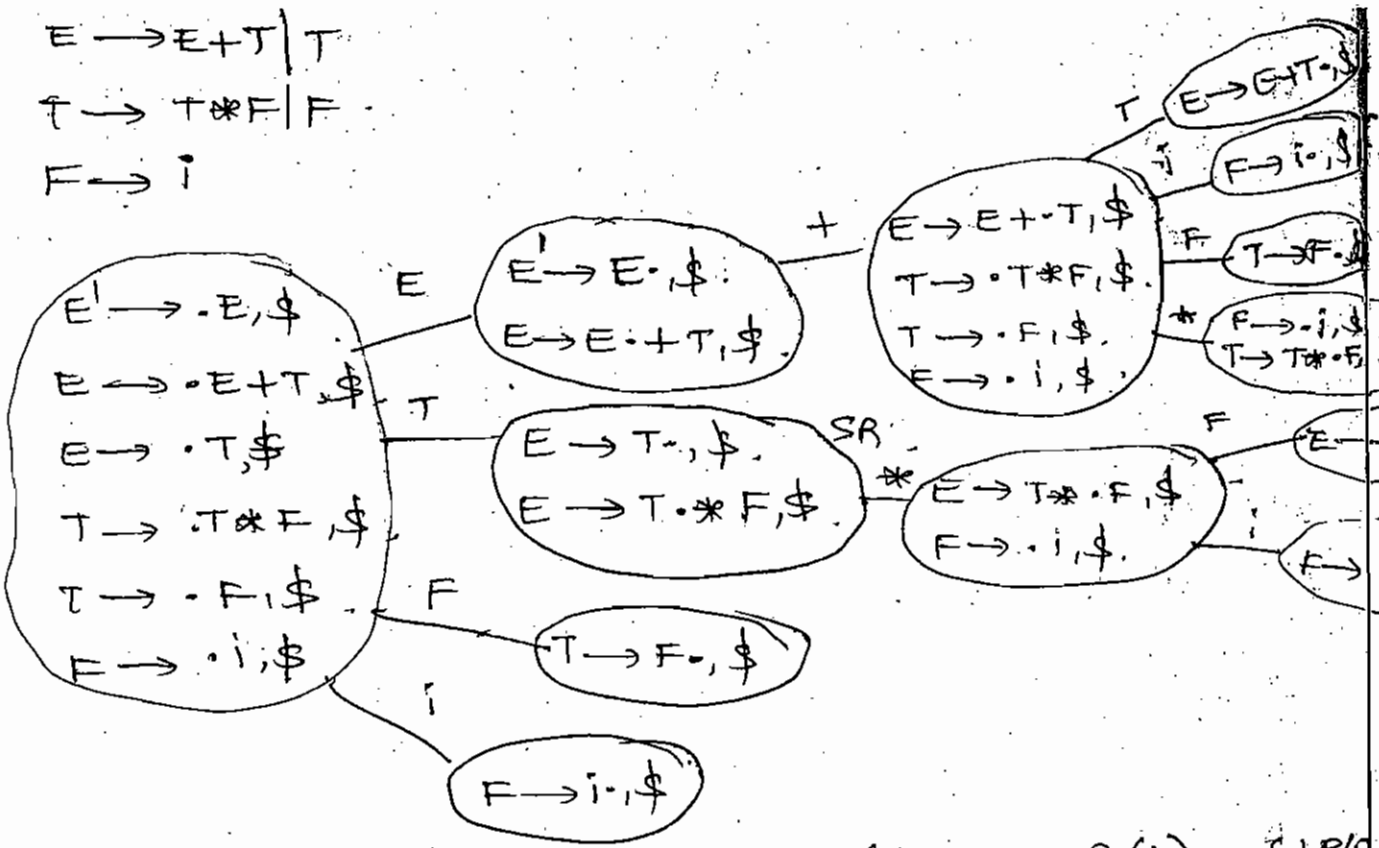
$B \rightarrow aB \mid b$



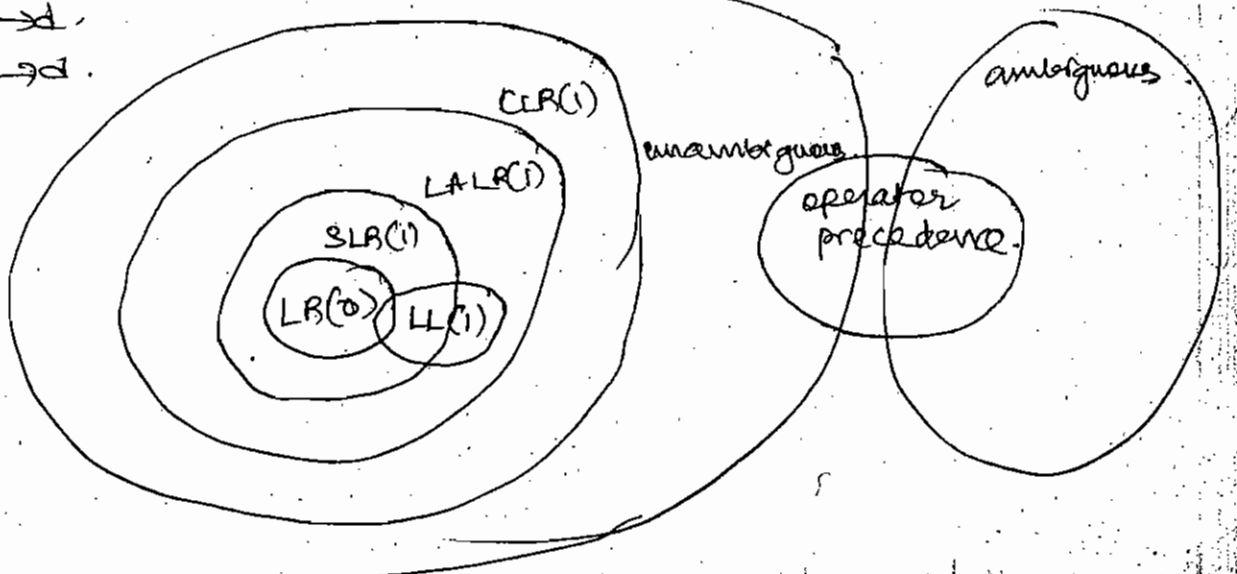
$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow i$



	LL(1)	LR(0)	SLR(1)	LALR(1)	CLR(1)
1) $S \rightarrow a$	✓	✓	✓	✓	✓
2) $E \rightarrow E+T \mid T$ $T \rightarrow i$	×	✓	✓	✓	✓
3) $E \rightarrow E+T \mid T$ $T \rightarrow T*F \mid F$ $F \rightarrow i$	×	×	✓	✓	✓
4) $S \rightarrow Aa \mid bAc \mid dc \mid bda$ $A \rightarrow d$	×	×	×	✓	✓
5) $S \rightarrow Aa \mid bAC \mid Bc \mid bBa$ $A \rightarrow d$ $B \rightarrow d$	×	×	×	×	✓



There are some grammars which cannot be parsed by any parser (unambiguous)

Page No.
Date: / /

Consider SLR(1) and LALR(1) table for a context free grammar. Which of the following is true.

a. GOTO of both tables may be different — FALSE

b. Shift entries are identical in both tables. — TRUE

c. Reduce entries in tables may be different — TRUE

d. Reduce entries in tables may be different — FALSE

Goto same in all tables.

Shift entries identical in all tables.

~~Every entries identical in all tables.~~

CLR(1)

SLR(1) — n_1 states

LALR(1) — n_2 states

CLR(1) — n_3 states

$n_3 \geq n_2 = n_1$

$n_1 = n_2$.

$n_3 \geq n_1$.

$S \rightarrow cC$

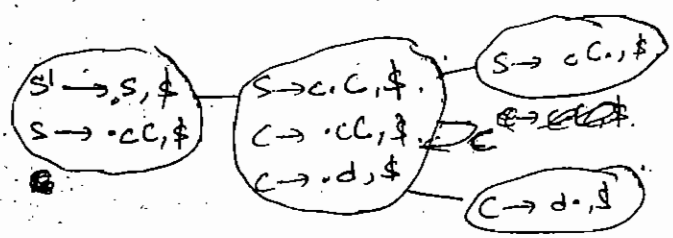
$C \rightarrow cC \mid d$.

a LL(1) ✓

b SLR(1) but not LL(1).

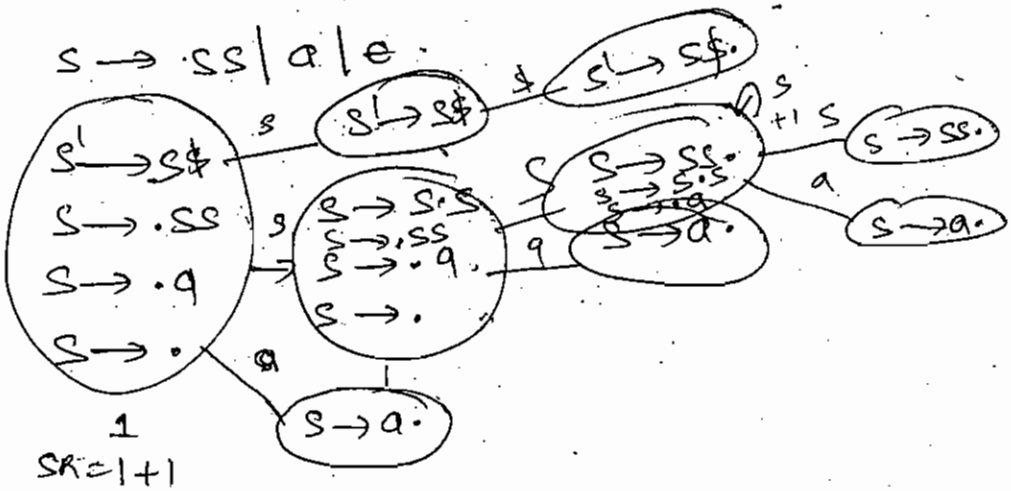
c. LALR(1) but not SLR(1)

d. CLR(1) but not LALR(1).



Find the number of SR and RR conflicts in a LALR with LR(0) items for the

$S \to SS \mid a \mid \epsilon$.



SR = 2.

RR = 0.

$E \to E+T \mid E*n \mid n$.

For $n+n*n$, the handles in right sentential form of reduction

a) $n, E+n, E+n*n$.

b) $n, E+n, E+E*n$.

c) $n, n+n, n+n*n$

d) $n, E+n, E*n$. ✓

$S \to (S) \mid a$.

SLR(1)	LR(1)	LALR(1)
n1	n2	n3.

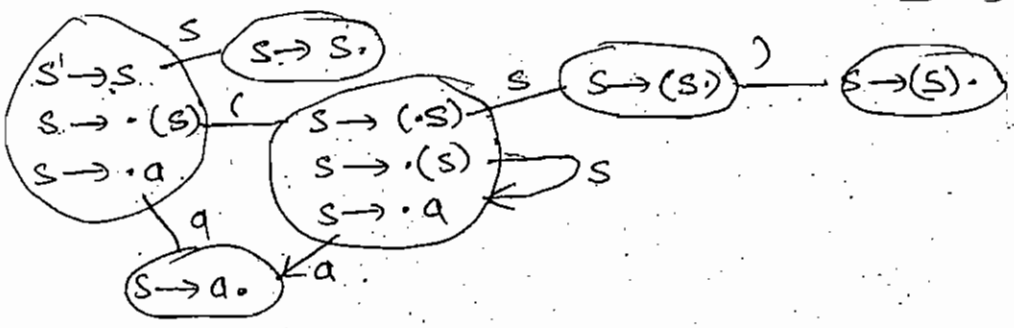
a) $n_1 < n_2 < n_3$

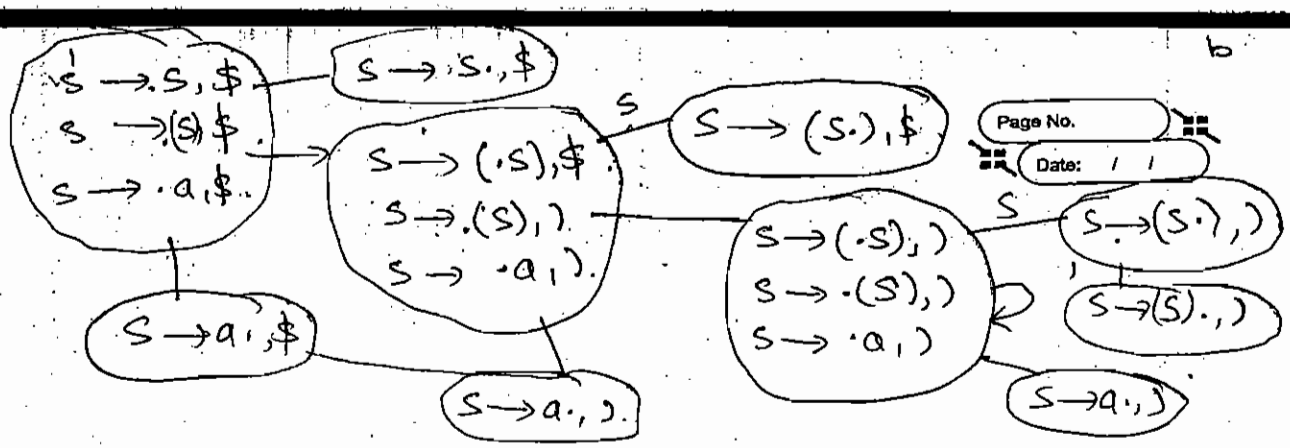
b) $n_1 = n_3 < n_2$

c) $n_1 \geq n_3 = n_2$.

d) $n_1 \geq n_3 \geq n_2$ ✓

CLR(1) also known as LR(1).





Advantages of ambiguous Grammars:

$E \rightarrow E+E \mid E*E \mid i$

ambiguous

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow i$

unambiguous.

1. Ambiguous Grammar G are always shorter & intuitive & more natural than equivalent unambiguous Grammars.
2. Precedence & associativity of the operator can be changed later with ambiguous grammars.
3. No wastage of time unambiguous Grammars waste time by carrying out reductions like $E \rightarrow T \rightarrow F \rightarrow i$

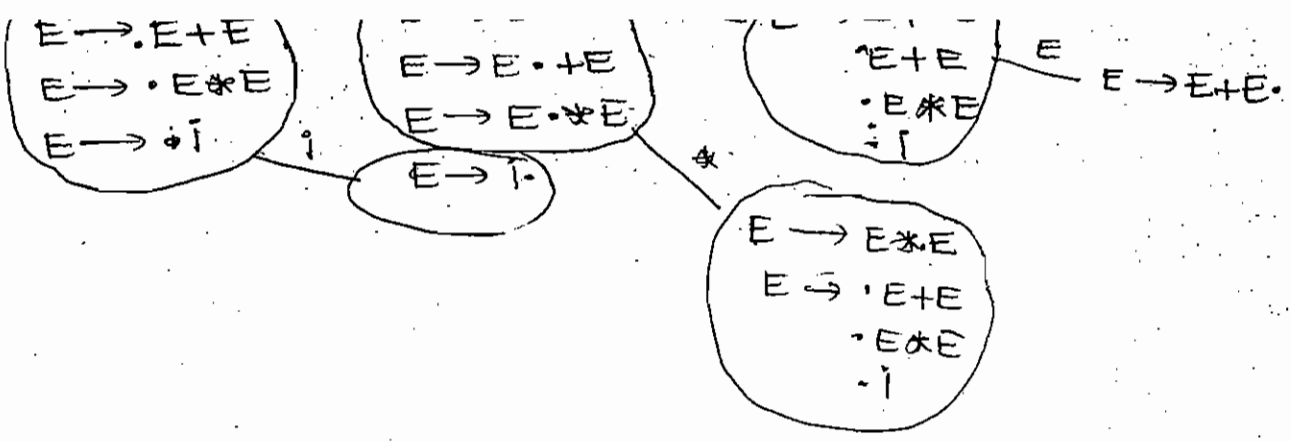
Constructing parser for ambiguous Grammars:

Using any parser for ambiguous Grammar results in multiple entries in the parsing table. If we can resolve them into single entry, we can give parsers for ambiguous Grammars, without any confusion.

ex: $E \rightarrow E+E \mid E*E \mid i$

YACC will resolve SR conflict in favour of shift and RR conflict in favour of first reduction or rule is favoured.

This conflict resolution does not give proper parsing solution in all cases as it might give undesired associativity (Right) in above Grammer to '+' operator



$S \to iS \mid iSeS \mid a$ · SR on seeing iS.

Reduce 'else' not matched with closest 'if'.

shift 'else' matched with closest 'if'

We get SR conflict on shift 'else' associated with closest 'if' or reduce 'else' associated with farthest 'if'

Action depends on Grammar.

SDT - Syntax Directed Translation:

Grammar + Semantic rules = SDT

examples for using SDTs.

To store or retrieve type information into symbol table

To issue error messages.

To perform consistency checks like parameter checking

Type checking

To build syntax trees.

syntax tree - condensed form of parse tree.

To Generate intermediate code or target code.

SDT for evaluation of a expression.

$E \rightarrow E_1 + T \mid T$ { E.val = E_1.val + T.val | E.val = T.val }

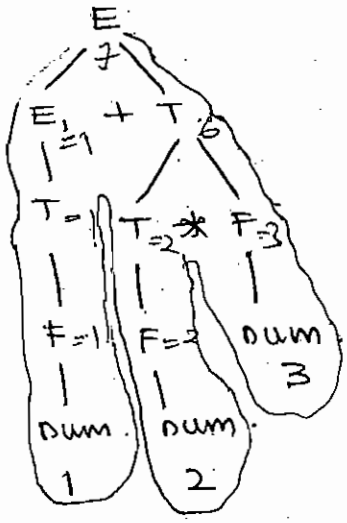
$T \rightarrow T_1 * F \mid F$ { T.val = T_1.val * F.val | T.val = F.val }

$F \rightarrow num$. { F.val = num.val }

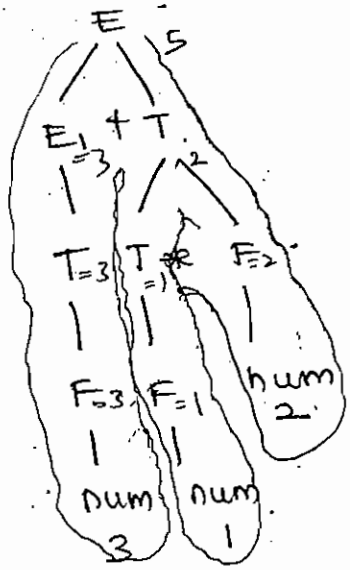
semantic rules.

E.val $\rightarrow$ E_1.val

1 + 2 * 3.



3 + 1 × 2.



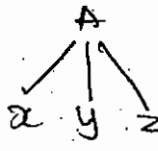
~~Attributes~~

Synthesized

Attribute value at a node is evaluated in terms of attributes of its children.

ex:

$A \rightarrow xyz$.

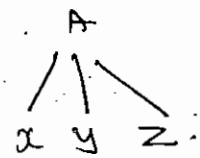


$A_i = f(x_i, y_i, z_i)$

Inherited.

Attribute values at a node is evaluated in terms of attributes of its siblings or parent.

$A \rightarrow xyz$.



$x_i = A_i$

$x_i = y_i + z_i$

$x_i = f(A_i, y_i, z_i)$

Example:

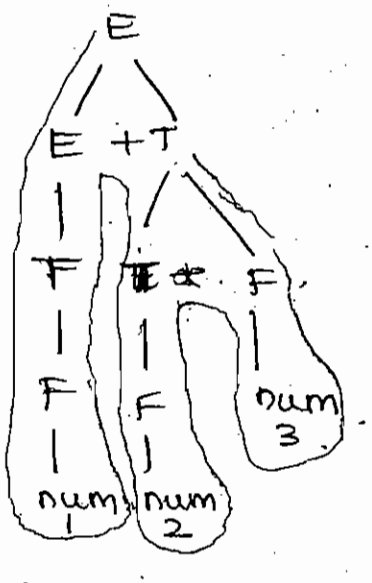
SDT for converting infix to postfix.

$E \rightarrow E+T \mid T$ { Print ('+'); } | { }

$T \rightarrow T*F \mid F$ { Print ('*'); } | { }

$F \rightarrow num.$ { Print(num.lvalue)

1+2*3



~~1 2 3 * +~~

1 2 3 * +

Simple type checker

~~E → E + E | E == E~~

$E \rightarrow E+E$ { $E_1.type == E_2.type$? $E.type = E_1.type$: Error }

$E \rightarrow E_1 == E_2$ { $E.type = E_1.type$? $E_1.type$: print error }

$E \rightarrow (E)$ { $E_1.type == E_2.type$? ~~$E_1.type$ print (error)~~ }

$E \rightarrow True.$ { $E.type = boolean.$

$E \rightarrow false$ { $E.type = boolean.$

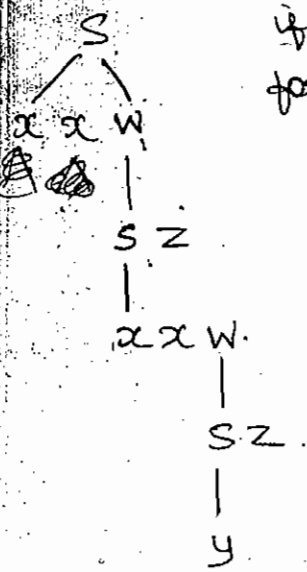
$E \rightarrow num.$ { $E.type = integer$ $E.value = num.$

$S \rightarrow xxW$ | y {pf(1)}.

$W \rightarrow Sz$ {pf(2)}

{pf(3)}

if above SDT is carried out by LR parser for the string xxxxyzz.



23131

$E \rightarrow E * T$ E.val = E.val * T.val.

T. E.val = T.val.

$T \rightarrow F - T$ T.val = F.val - T.val.

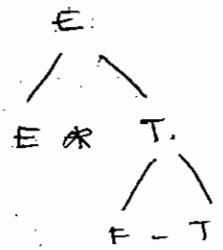
F. T.val = F.val.

$F \rightarrow 2$ F.val = 2.

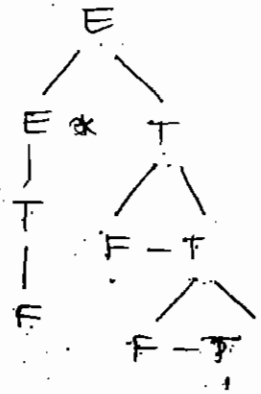
4 F.val = 4

$(4-(2-4)) * 2.$

$4+2=6 \times 2 = 12$.



compute total no of reductions performed.



10

every non terminal is reduction

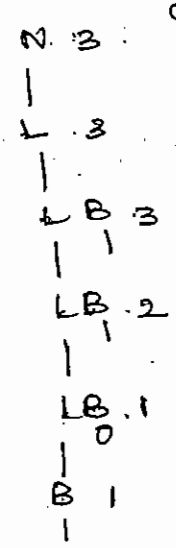
SDT to count no of 1s in a binary number.

$(0+1)^*$ RE

$N \rightarrow L$	$\{N.num = L.num\}$
$L \rightarrow LB$	$\{L.num = L.num + B.num\}$
B	$\{B.num = B.num$
$B \rightarrow 0$	$\{B.num = 0$
1	$\{B.num = 1\}$

1011.

9 reductions.



SDT to convert binary to decimal.

to decimal

$N \rightarrow L$	0
$L \rightarrow LB$	1 1
B	$10 = 2$
$B \rightarrow 0$	$1 \times 2 + 0 =$
1	

$M \rightarrow NX2 + N \mid N$

1010

8
2

2^{0-ml}

$N \rightarrow L$	$\{N.val = L.val\}$
$L \rightarrow LB$	$\{L.val = L.val \times 2 + B.val\}$
B	$\{L.val = B.val\}$
$B \rightarrow 0$	$\{B.val = 0\}$
1	$\{B.val = 1\}$

3

$N \to L$ { N.val = L.val, M.count = L.count }.

$L \to LB$ { L.val = L.val * 2 + B.val, L.count = B.count }.

B { L.val = B.val, L.count = B.count }.

$B \to 0$ { B.val = 0, B.count = 1 }.

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

$M \to L_1 . L_2$ { N.val = L_1.val * 2 + L_2.val / $2^{L_2.count}$ }.

L { M.val = L.val, M.count = L.count }.

$\frac{3}{2^2} = \frac{3}{4}$

$E \to E \# T$ { E.val = E.val * T.val }.

T { E.val = T.val }.

$T \to T \$ F$ { T.val = T.val + F.val }.

F { T.val = F.val }.

$F \to num$ { F.val = num }

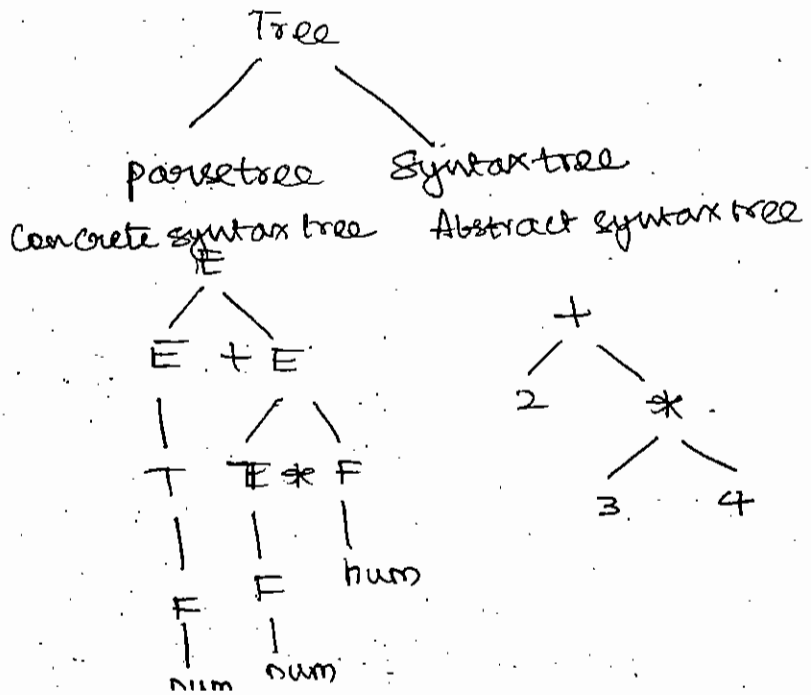
$2 \# 3 \$ + 5 \# 6 \$ 4$.

$2 * 3 + 5 * 6 + 4$.

$2X \quad 8 \quad X10$

$16 \times 10 = 160$

SDT that uses only synthesized attributes is referred to as S-attributed definition.



Parse tree practically gives syntax of Grammar & so it is called concrete syntax tree.

SDT to build syntax tree

$E \rightarrow E + T$. { E.nptr = makenode(E.nptr, +, T.nptr) }

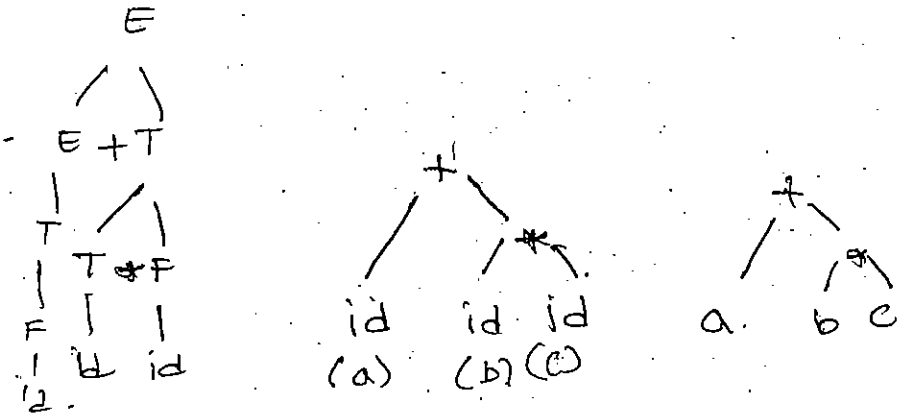
T { E.nptr = T.nptr }.

$T \rightarrow T * F$ { T.nptr = makenode(T.nptr, *, F.nptr) }

F { T.nptr = F.nptr }

$F \rightarrow id$. { F.nptr = makenode(null, id, name, null); }.

Let us assume that there is a procedure makenode which will create a node with three fields and return a pointer to that node there is attribute called nptr. with every Grammar symbol.



Generally always a terminal symbol & start symbol only takes Synthesised attributes.

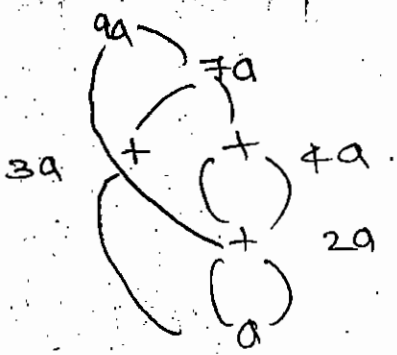
DAG. Directed acyclic Graph.

Repetitions are not allowed like in trees. and so it is used for optimization.

P: 327.

5.3.

a + a + (a + a + (a + a + a + a))



In above SDT in order to create DAG instead of syntax tree make changes to make node function. If we try to create a node already existing just return pointer of existing node.

In case of bottomup parsers, we carry out semantic actions whenever, we reduce a production. If we have to perform SDT in case of topdown parser, place a dummy variable for every semantic action. whenever this dummy variable appears on the top of the stack perform the semantic action associated with that variable.

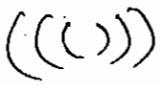
SDT to count no of balanced parenthesis.

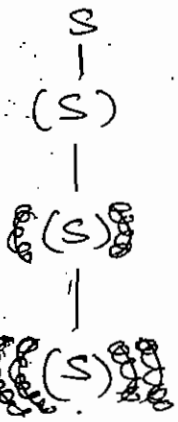
$S \rightarrow (S) \quad \{ S.count = S.count + 1 \}$

$\epsilon \quad \{ S.count = 0 \}$

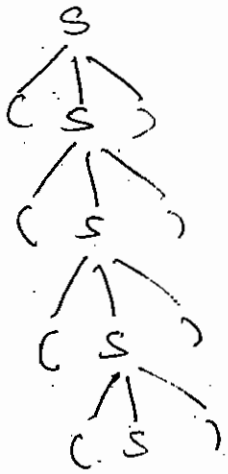
$S \rightarrow (S) \#1$

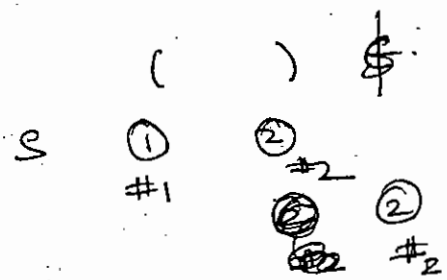
$\epsilon \quad \#2$





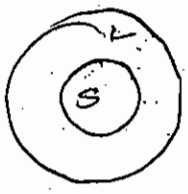
4

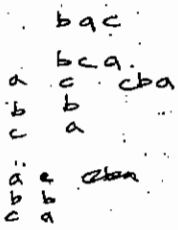




Whether it is Top down or Bottom up parsing output is always same.

SDT





S-attributed

postfix SDT

Uses only synthesized attributes.

$A \longrightarrow XYZ$.

$A.S = X.S + Y.S + Z.S$.

$X.S = A.S$ X wrong.

L-attributed

Uses both synthesized and inherited attributes

1. Each inherited attributed is restricted either to inherit from parent or left sibling only

$A_i \longrightarrow XYZ$.

$X_i = A_i$

$Y_i = X_i + A_i$

$Z_i = X_i + Y_i + Z_i$

Semantic actions are placed at right end of productions.

$A \longrightarrow BC\{\}$.

Semantic actions can be placed anywhere on RHS

$A \longrightarrow BC\{\}$

$A \longrightarrow \{\}BC$

$A \longrightarrow B\{\}C$.

Attributes are evaluated during bottom up parsing

$A \longrightarrow LM\{\}$ $A_i = f(L_i, M_i)$

Attributes are evaluated by traversing parsetree DFS left to right

$A \longrightarrow LM\{L_i = f(A_i);$

$M_i = f(L.s);$

$A.s = f(M.s);$

$\}$

a) S-attributed
b) L-attributed
c) Both
d) None ✓

$A \rightarrow QR \{ R_i = f(A.i) \quad Q.i = f(R.i) \quad A.s = f(Q.s) \}$

$A \rightarrow BC \{ B.S = A.S \}$.

a) S-attributed
b) L-attributed ✓
c) both
d) None

Every S-attributed is L-attributed

S-Attributed definition for converting infix to postfix.

$E \rightarrow E+T \{ print(+) \}$.

$T \quad \{ \}$

$T \rightarrow T*F \{ print(*); \}$

$F \quad \{ \}$

$F \rightarrow num \quad \{ print(num.lvalue) \}$

Eliminate left recursion from Grammar so that it can be parsed by topdown parsers.

Consider semantic rules as dummy variable ✓ eliminate recursion.

$E \rightarrow E+T \quad \{ print(+) \}$ #1.

T

$T \rightarrow T*F \quad \{ print(*) \}$ #2.

F.

$F \rightarrow num \quad \{ print(num) \}$ #3

both S and L attributed

$E \rightarrow TE'$

$E' \rightarrow +T\#_1 E' \mid \epsilon$.

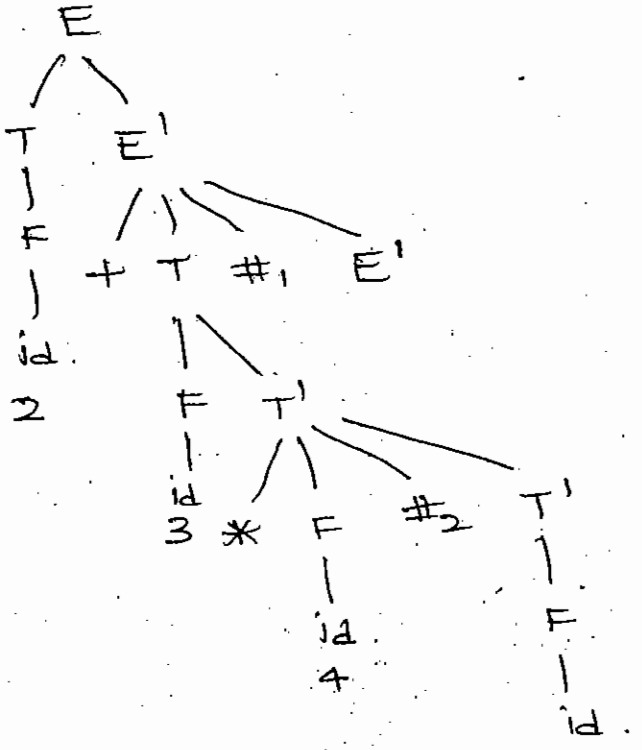
$T \rightarrow FT'$

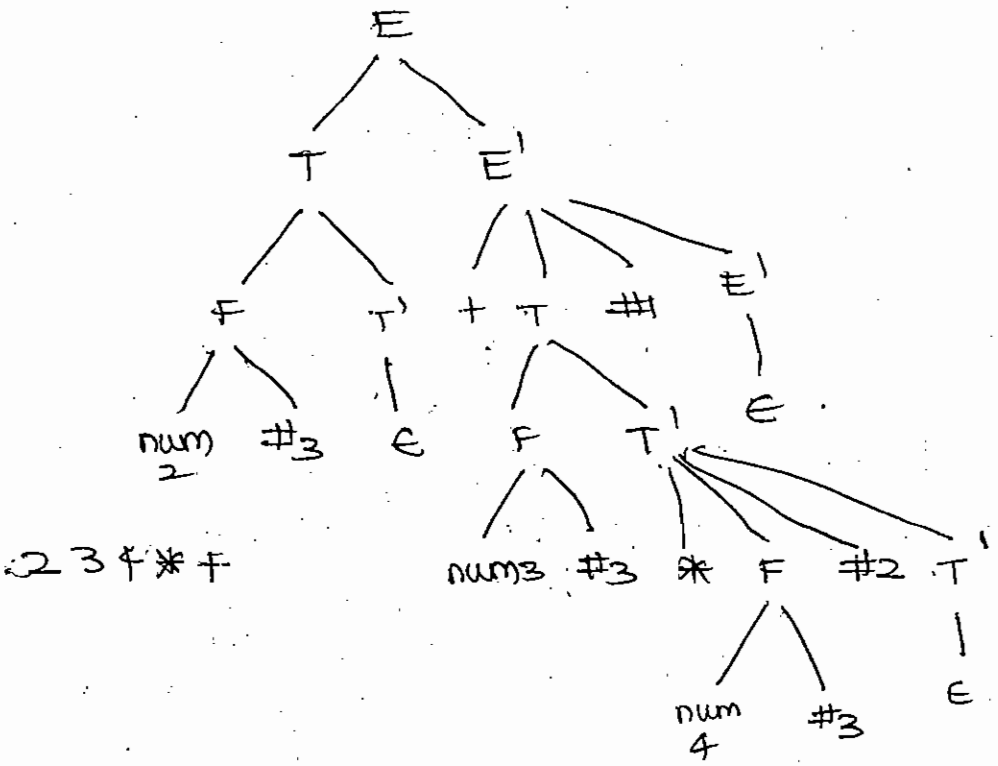
$T' \rightarrow *F\#_2 T' \mid \epsilon$.

$F \rightarrow num \#_3$.

L-attributed not S-attributed

The resulting SDT is not S-attributed.





Note: Whether it is S-attributed or L-attributed SDT output is always same.

Page No.
Date: / /

Converting L-attributed to S-attributed.

$A \rightarrow B \overset{M}{\{\}} C$ L not S.

$A \rightarrow BMC$. Both L and S.

$M \rightarrow \epsilon \{\}$

Use dummy variable and pull it out.

$E \rightarrow TE'$

$E' \rightarrow +T \overset{M}{\{} print(x); \{E'$ L not S.

ϵ.

$T \rightarrow i$

$E \rightarrow TE'$

$E' \rightarrow +TME'$ Both L and S.

$E' \rightarrow \epsilon$.

$M \rightarrow \epsilon \{ print(x); \}$.

$T \rightarrow i$

Convert following SDT to a SDT that is

a. postfix SDT.

b. has no left recursion.

$A \rightarrow A \overset{M}{\{} a \} B$

$B \{ b \}$.

$B \rightarrow 0 \{ c \}$.

$A \rightarrow AMB \mid B \{ b \}$

$M \rightarrow \epsilon \{ a \}$.

$B \rightarrow 0 \{ c \}$

$A \rightarrow A\alpha \mid \beta$.

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$.

$A \rightarrow B \overset{\#1}{\{} b \} A'$

$A' \rightarrow \overset{\#2}{\{} a \} BA' \mid \epsilon$.

$B \rightarrow 0 \{ c \}$.

$A \rightarrow BM_1A'$

$A' \rightarrow M_2BA' \mid \epsilon$.

$B \rightarrow 0 \{ c \}$

$M_1 \rightarrow \epsilon \{ b \}$.

$M_2 \rightarrow \epsilon \{ a \}$.

SDT to store type information into symbol table.

$D \rightarrow TL$ { L.type = T.type } . L-attribute

$T \rightarrow$ int { T.type = int } S-attribute

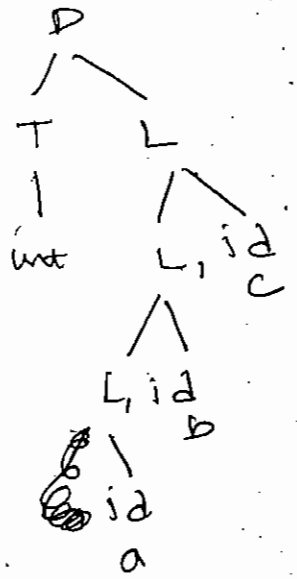
char { T.type = char } S → attribute.

$L \rightarrow L_1$, id { L_1.type = L.type } L-attribute

id { id.type = id.type } S-attribute

add type to symbol table (id.name, L.in)

int x, y, z



General procedure common for both L and S attributed

ip string parse it and construct parse tree.

traverse parse tree to

input string → Parse → Parse tree → Traverse → dependency graph.

↓ topological sort

give output

Evaluating an L-attributed definition with both synthesised and inherited attributes

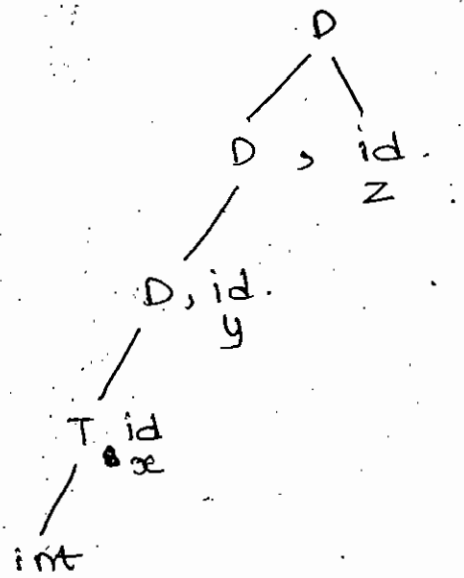
Page No.
Date: / /

Traverse the parse tree depth first left to right.
Evaluate the inherited attribute when you visit it for the first time.
Evaluate the synthesised attribute when the node is visited for the last time.

To convert to S-attributed definition from L-attributed defn, we need to avoid inherited attributes.
It is not possible with the same grammar and so we have to rewrite the grammar.

$D \longrightarrow D_1, id$ { addtype(id.name, D_1.type) }
$T\ id$ { D.type = T.type, addtype(id.name, T.type) }

$T \longrightarrow int$ { T.type = int }
$char$ { T.type = char }.



int x, y, z

$E_1 \rightarrow$ number $\{E_1.val = number.lval\}$

$E_2 + E_3$ $\{E_1.val = E_2.val + E_3.val\}$

$E_2 * E_3$ $\{E_1.val = E_2.val * E_3.val\}$

yacc a given above Grammar as input.

a. It detects recursion & eliminate.

b. It detects RR conflict and resolves in favour of shift

c. It detects SR conflict and resolves in favour of reduce

d. It detects SR conflic and resolves in favour of shift ✓

Consider the expression 3*2+1. what associativity & precedence properties does generated parse tree realises. for above resolution

6+2 = 8 3*2+1 3 = 9

L-R.

precedence + > *.

A. Equal precedence & Left associativity, exp evaluated to 7

B. Equal precedence & right associativity exp evaluated to 9.

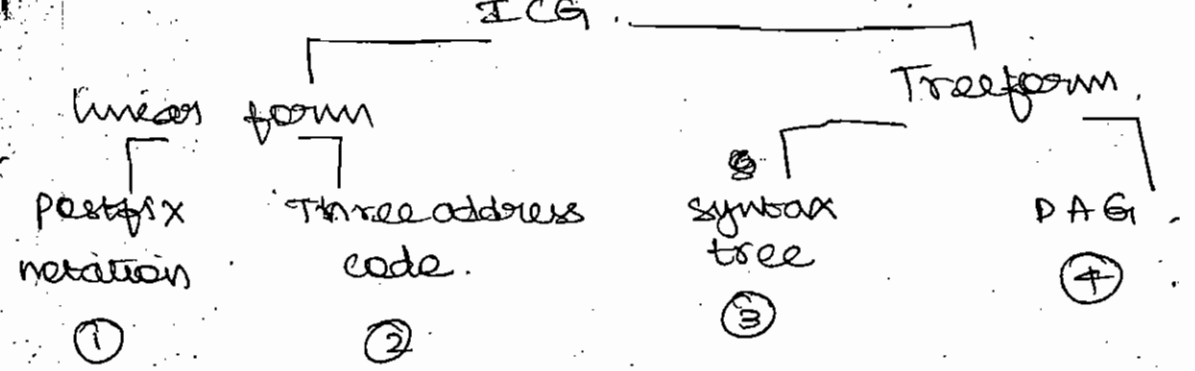
C. * or > +, both are ~~left~~ left associative exp to 7

d. + > * both are left associative exp to 9 ✓

Intermediate Code Generation:

2 advantages

1. Retargetting is supported.
2. Machine independent optimizations are also supported.



① $(a+b) * (a+b+c)$.

$ab+ab+c+*$:

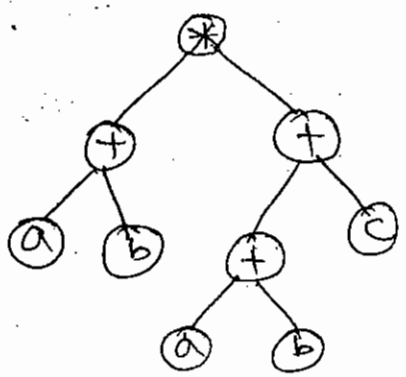
② $t_1 = a+b$.

$t_2 = a+b$.

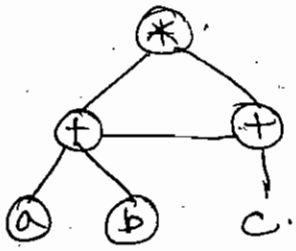
$t_3 = t_2+c$.

$t_4 = t_1 * t_3$.

③



④



Types of Three address code.

1) $x = y \; op \; z$.

2) $x = op \; z$.

3) $x = y$

4) if x relop y goto L

5) goto L

6) $a[i] = x$

$y = a[i]$

7) $x = *p$.

$y = \&x$.

SDT to generate three address code

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

$E \rightarrow E + T$ { E.place = newtemp(); gen(E.place = E.place + T.place); }

T { E.place = T.place }.

$T \rightarrow T * F$ { T.place = newTemp(); gen(T.place = T.place * F.place); }

F { T.place = F.place }

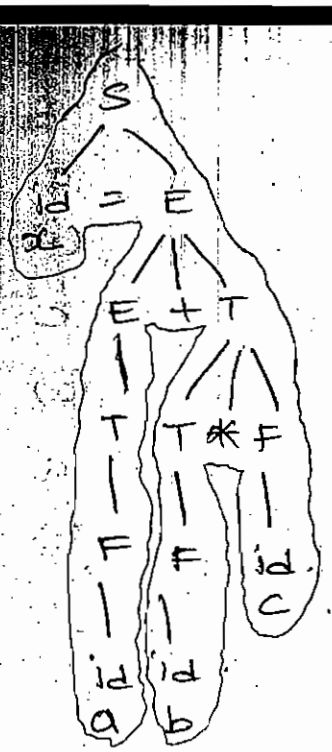
$F \rightarrow id$ { F.place = id.name }

- place is used to store identifier name of temporary variables.

temp(); is used for generating unique temporary variables.

gen(); is used to generate the code.

$x = a + b * c$.



$t_1 = b * c$

$t_2 = a + t_1$

$x = t_2$

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

$E \rightarrow E_1 + E_2$ { t = newtemp();
gen (t = E1.place + E2.place);
E.place = t; }.

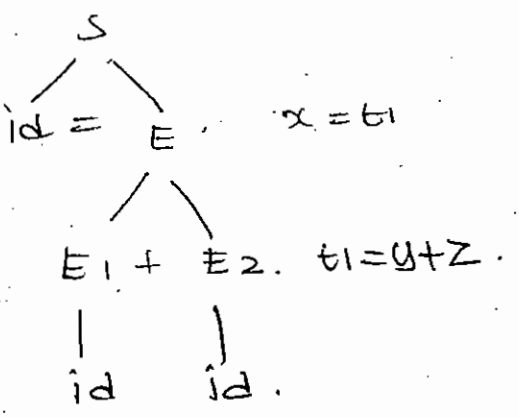
$E \rightarrow id$ { E.place = id.name }.

a) $x = y + z$.

b) $t_1 = y + z$ $x = t_1$ ✓

c) $t_1 = y$, $t_2 = t_1 + z$, $x = t_2$

d) $t_1 = y$ $t_2 = z$ $t_3 = t_1 + t_2$ $x = t_3$



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

Quadruples

	op	op1	op2	result
1) $t_1 = a+b$	+	a	b	t1
2) $t_2 = -t_1$	-	t1		t2
3) $t_3 = c+d$	+	c	d	t3
4) $t_4 = t_2 * t_3$	*	t2	t3	t4
5) $t_5 = a+b$	+	a	b	t5
6) $t_6 = t_5+c$	+	t5	c	t6
7) $t_7 = t_4+t_6$	+	t4	t6	t7

Triples

	op	op1	op2
1)	+	a	b
2)	-	(1)	
3)	+	c	d
4)	*	(2)	(3)
5)	+	a	b
6)	+	(5)	c
7)	+	(4)	(6)

Advantage Quadruples:

Statements can be moved around.

Disadvantage.

too much space is wasted.

Triples.

space is not wasted.

statements cannot be moved around.

Indirect

space is effective

statement can be moved around

When we have any complex expression change it to postfix then create any of the 3 IC8 forms.

ex: ((a*(b+c))/(d+e))

abc+*de+/

$t_1 = b+c$

$t_2 = a * t_1$ $t_3 = d+e$ $t_4 = t_2 / t_3$

triple		indirect triple
op1	op2	
a	b	(15)
(1)		(12)
c	d	(13)
(2)	(3)	(14)
a	b	(15)
(5)	c	(16)
(4)	(6)	(17).

Relational expression like a<b. can be represented in 2 ways

1. Value representation (t= a<b)
2. Flow of control representation (if a<b ~~goto~~
   t=1
   else
   t=0).

if (a<b)
  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 ;

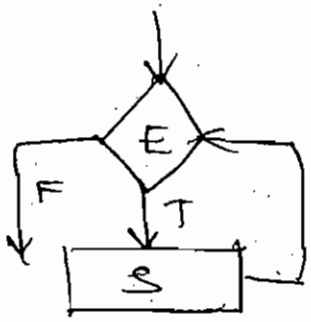
SDT : a<b and c<d. or e<f.

1 if(a<b) goto.
2 t1=1 goto
4 if(c<d) goto.
5 t2=1. goto
8 if(e<f) goto.
9 t3=1. goto.

a<b and c<d or e<f

```
100  if (a<b) goto 103
101  t1=0
102  goto 104
103  t1=1
104  if (c<d) goto 107
105  t2=0
106  goto 108
107  t2=1
108  if (e<f) goto 111
109  t3=0
110  goto 112
111  t3=1
112  t4=t1 and t2
113  t5 = t4 or t3
```

while E do S.

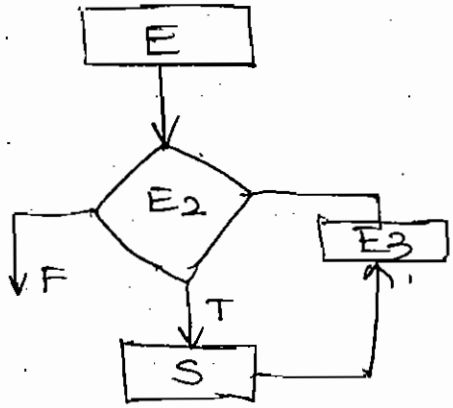



```
L: if (E==0) goto L1
   S
   goto L
L1
```

while (a<0) do x=y+z

```
L: if a<0 goto L1
   goto Last
L1: t = y+z
   x = t
   goto L
Last:
```

for(E1; E2; E3) S.



for(i=0; i<10; i++).

a = b+c

i = 0;

L : if i==10 goto Last

t1 = b+c. a = t1

t2 = i+1 i = t2

goto L

Last:

switch(i+j)

{

case 1 : a = b+c;
break;

case 2 : p = q+r;
break;

default : x = y+z;
break;

}

test: if(t==1) goto L1

if (t1==2)) goto L2

goto L3;

~~if((i+2)==3)~~

~~b1 = b+c.~~

~~a = t1;~~

t1 = i+j

goto test

L1 : t1 = b+c.

a = t1

goto last;

L2; t2 = q+r;

p = t2;

goto last

L3 & t3 = y+z

x = t3

goto last

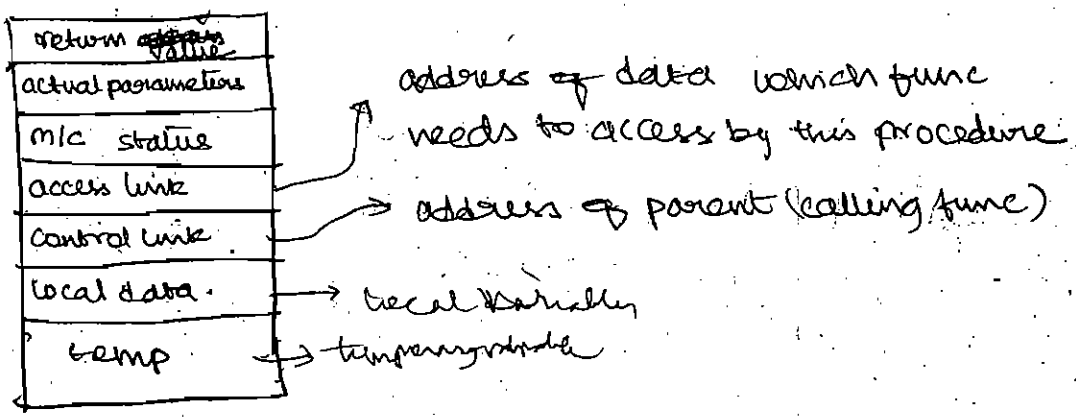
```
x = A[y,z]
t1 = y * 20
t2 = t1 + z
t3 = t2 * sizeof(int)
t4 = base(A)
t5 = t4 [ t3 ]
x = t5
```

```
main()
{
  int i = 1;
  int a[10];
  while (i <= 10)
    a[i] = 10;
}
```

```
    i = 1
L1  if (i <= 10) goto L2
    goto last
L2: t1 = i * W
    t2 = base(a)
    t2[t1] = 10;
    goto L1
Last:
```

compile time	Runtime
1. declaration of variable	Binding of variable
2. scope	lifetime
3. definition of procedure	activation of procedure

Activation Record :



Runtime environment

Storage allocation strategies.

Page No.
Date: / /

1. static
   - allocation is done at compile time.
   - binding do not change at runtime
   - 1 activation record per procedure

disadvantage: recursion is not supported
- size of the data object must be known at CT
- Data structures cannot be created dynamically

2. stack allocation
   - whenever a new activation begins, activation record is pushed onto stack and whenever activation ends activation records are popped off.
   - locals are bound to fresh storage
   - recursion is supported.

disadvantage: local variables cannot be retained once activation ends.

3. Heaps
   - Allocation & deallocation can be done in any order

disadvantage: heap management is overhead.

Summary:

activations can have.

1. permanent life times in case of static allocation
2. nested lifetime in case of stack allocation
3. Arbitrary lifetime in case of heap allocation

optimization.

machine independent	machine dependent
1. loop optimization a) code motion. frequency reduction b) loop unrolling. c) loop jamming.	1. Register allocation 2. use of addressing modes 3. peephole optimization. a. redundant ~~load~~ store elimination b. flow of control options c. strength reduction d. use of machine idioms
2. Folding constant propagation	
3. Redundancy elimination	
4. Strength reduction	

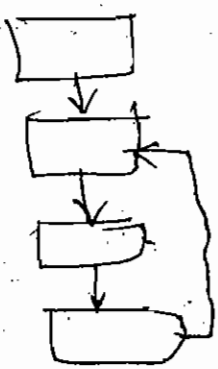
machine independent optimization

1. In order to apply these optimizations first loops in ICR for detecting loops we use control flow analysis using program flow graph.

Basic block:

sequence of 3 address statements where control enters at the beginning & leaves only at the end without any jump or halts.

Divide the program into basic blocks.



If there is a cycle in the graph then it indicates loop.

Page No.
Date: / /

Algorithm for finding basic blocks.

1. We have to identify the leaders in the program. A basic block is starting from one leader to the next leader but not including the next leader.

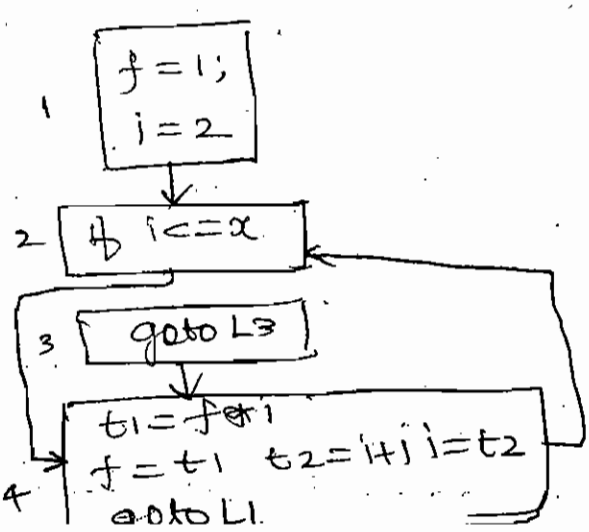
Identifying leaders in basic blocks.

1. first statement is a leader.
2. statement that is target of a goto is a leader. conditional or unconditional jump.
3. statement that follows immediately a conditional or unconditional jump is a leader.

example:

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

```
f=1;                    — L
i=2;
if i<=x goto L2         — L
  goto L3               — L
t1=f*i                  — L
f=t1; t2=i+1 i=t2;
goto L1
goto calling procedure  — L
```



cycle.

cycle in the graph indicates loop.

frequency reduction

moving the code from high frequency region to low freq region. also called code motion.

Ex: while(i<5000)

A = sin(x) / cos(x) * i

⇓

temp = sin(x) / cos(x)

while (i<5000)

A = temp * 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 ~~unexpected~~ independent loops

Page No.
Date: / /

gcc optimization

level 1

level 2

level 3 may lead to error.

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

Folding.

replacing an expression that can be computed at compile time by its values.

```
2+3+c+b.
```

⇓

```
5+c+b
```

Disadvantage: folding is local optimization within a basic block.

folding should ~~not~~ be applied for unsubscripted variable

while applying to floating point ~~no~~ of commutativity and associativity should be applied.

```
(11+2.8)+0.3
```

⇓

~~13.8+0.3~~

11+2 = 13

```
11+(2.8+0.3)
```

⇓

11+3 = 14

folding should not introduce additional errors in program ex divide by zero or sqrt of negative no

Redundancy elimination

A = B+C.
D = 2+B+3+C.
~~D<C 5555~~ ⇓.

A = B+C.
D = 2+A+3.

in eliminating redundancy DAG are useful.

Strength reduction

~~reducing~~ replacing a costly operation by a cheaperone

A = A*2.

⇓

A = A<<1

A = A*2^n
A = A<<n

Algebraic simplification

it is.

A = A+0. x = x*1 eliminate

Machine dependent

1. Register allocation — local allocation
   — global allocation

depends on architecture & machine instruction supported

2. Addressing mode.

3. Peephole optimization

a. redundant load/store elimination

x = y+z
a = x+k
~~5555~~ MOV y,R0 ADD z,R0 MOV R0,x ⇒
MOV x,R0 ADD k,R0 MOV R0,A

Flow of control optimization.

avoid jumps on jumps

```
L1: jmp L2
    .
    .
L2: jump L3.
    .
    .
L3: jump L4
```

↓

```
Li: jump L4.
```

eliminate dead code.

```
#define x 0.
if (x).
{
    deadcode.
}
```

use of machine idioms.

take the advantage of arch instruction set available.

```
i = i+1.
MOV R, Bo.
ADD Bo, 1
MOV Bo, i
```

↓

```
inc i
```

MOV y, Bo, ADD Z, Bo MOV R
ADD x, Bo MOV Bo, A.

C.D – Aho-Ul-Man K.V, Notes,

Digital – Kohavi

DBMS – Navathe

DAA +DS – Intro to Algo by Coreman

C – Dennis Richie.

TOC – Notes + Aho-Ul-Man.

Principle of Programming Language – Deve Shetty

O.S – Galvin

S.Engg. – Roger Teresmanne.

Gigapedia. co