GATE CSE NOTES

by

Joyoshish Saha

Introduction

HLL
↓
Preprocessor
↓ Pure HLL
Compiler
↓ Assembly language
Assembler
↓ M/C code (relocatable)
Loader/Linker
↓
Executable code / absolute M/C code

#include file inclusion
#define Macro expansion

* Single, Multipass Compiler.
  - Single: More memory, Less time. → as more memory needed for all phases
  - Multipass: Less memory, More time → for context switch

* Type info is stored by semantic analyser.

* If no declaration is there for identifier, LA saves info at symbol table.

✓ Various phases

HLL
↓
Lexical Analysis
↓ stream of tokens
Syntax Analysis
↓ Parse tree
Semantic Analysis (type checking)
↓ Parse tree (Semantically verified)
Intermediate Code Generation
↓ Three Address Code.
Code optimization (optional)
↓
Target Code Generation
↓
Assembly

Symbol table manager — Read or Write — contains all identifiers

Error handler

- Front end of compilation (LA, Sy A, Se A, ICG)
- Back end " " (TCG)
- Code opt^n neither front nor back end.

→ Example

$x = a + b * c ;$ --- deleting comments.

⇓

Lexical Analyser LEX

Pattern id^r $l(l+d)^*$

↳ Need to be variable, can't be a fn name or array name

⇓

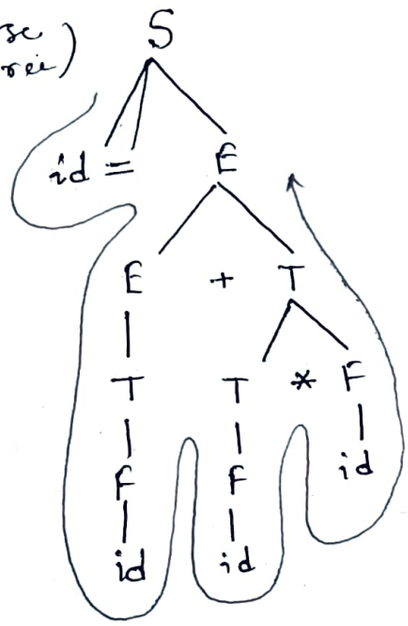
id = id + id * id ← Stream of tokens

⇓

YACC **Syntax Analyser** ⇐

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

(Parse tree)



⇒

yield of the parse tree and syntax analyser need to be same. (otherwise syntax error)

Semantic analyser

⇓

(Parse tree semantically verified)

⇓

Intermediate Code Generator

⇓

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

3-address code (max. 3 addr.)

⇓

Code optimiser

⇓

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

⇓

Target code generator

⇓

```
mul  R1, R2
add  R0, R2
mov  R2, x
```

a → R0
b → R1
c → R2

Writing code that assembler can understand

✓ Till ICG, we can implement in any machine. Next 2 phases need to be changed for different machines.

Till ICG
→ Front end

CO — ICG
CO: neither front nor back
ICG: → Back end

LANCE

* Symbol Table.

(allows us to find the record for each identifier quickly & to store or retrieve data from the record.)

Data structure created & maintained by compilers in order to store information about the occurrences of various entities such as variable names, function names, objects, classes, interfaces etc.

The information is collected by the analysis phase of a compiler & used by the synthesis phase to generate target code.

→ Usage of symbol table by phases:

Phase	Usage.
1. Lexical analysis	Creates new entries for each new identifiers.
2. Syntax analysis	Adds information regarding attributes like type, scope, dimension, line of reference & line of use.
3. Semantic analysis.	Uses the available information to check for semantics & is updated.
4. Intermediate code generation	Information in symbol table helps to add temporary variable's information.
5. Code optimization	Information in symbol table used in machine-dependent optimization by considering address & aliased variables' information.
6. Target code generator	Generates the code by using the address information of identifiers.

→ Symbol table entries.

Each entry in the symbol table is associated with attributes that support the compiler in different phases. These attributes are:

1. Name
2. Size
3. Dimension
4. Type
5. Line of declaration
6. Line of usage
7. Address

* Symbol table implemented as:

	Insertion	Search
LL	O(1)	O(n)
Hash table	O(1)	O(1)
AVL tree	O(lgn)	O(lgn)

eg.

Name	Type	Size	dim	LOD	LOU	Addr.
RAVI	char	4	1	..	..	..
AGE	int	2	0	..	..	..

- All the attributes are not of fixed size.

→ Limitation of fixing the size of symbol table:

i) If chosen small, it can't store more variables.

ii) If chosen large, lot of space wasted.

So, the size of symbol table should be dynamic to allow the increase in size at compile time.

→ Operations on the symbol table.

Dependent on whether the language is block-structure or non-block structured.

✓ Non-block structured: Contains only one instance of the variable declaration and its scope is through out the program.

For non-block structured languages the operations are:

→ Insert
→ Lookup

```
{ int i;
...
}
```

✓ Block structured: Here the variables may be redeclared if its scope is within that block.

Operations are: Insert
Lookup
Set
Reset.

```
{ int i;
...
{ int i;
...
}
}
```

G'12. Access time (Lookup time) of the symbol table is logarithmic if it is implemented by a

a) Linear list (O(n).)
✓b) Search tree (O(log$_2$ n)) → for binary (k for general - no of children max)
c) Hash table (O(1))
d) None

* LA, SyA, SeA will work at a time by synchronising with each other, so that within one pass all 3 compute their work, otherwise 3 passes needed.

• Implementation of Symbol table	Insertion time	Lookup time	Disadvantages.
(1) Linked list			- Lookup time directly proportional to table size.
(a) Ordered list			
→ Array	O(n)	O(log n)	- Every insertion operation preceded with lookup operation. (no duplicate elems)
→ Linked list	O(n)	O(n)	
(b) Unordered list	~~O(1)~~ O(1)	O(n)	
(2) Self organising list. (unsorted)	O(1)	O(n)	Poor performance when less frequent items are searched.
(3) Search tree	$O(\log_k n)$	$O(\log_k n)$	We have to always keep it balanced.
(4) Hash table	O(1)	O(1)	When there are too many collisions the time complexity increases to O(n).

* Lexical analyser. — Grammars.

- Lexemes → Tokens.
- Removing comments
- Removing white spaces
- If error, show (by providing row, col no.)

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

tokens = 25

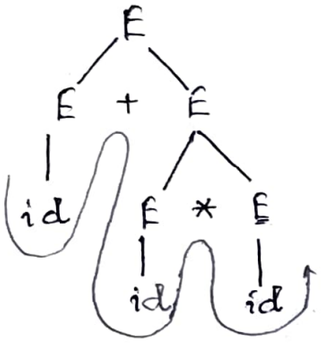
- printf (" %d asd " , & x) ;
  1 2 3 4 5 6 7 8

No. of tokens. — 8

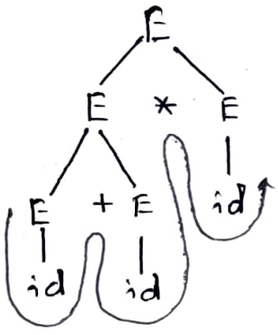
- Grammar G = (V, T, P, S).
- LA uses DFA for tokenization. LA only phase that reads prog. char by char.

eg. $E \to E+E \mid E*E \mid id$

$E \Rightarrow id + id * id$



$2 + 3 * 4$
$= 14$

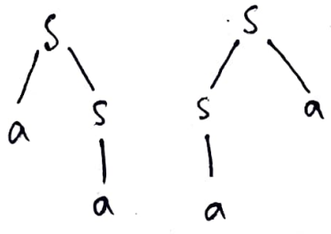


$2 + 3 * 4$
$= 20$

↘ ↓ Ambiguous. (>1 parse trees)

eg. $S \to aS \mid Sa \mid a$

$w = aa.$

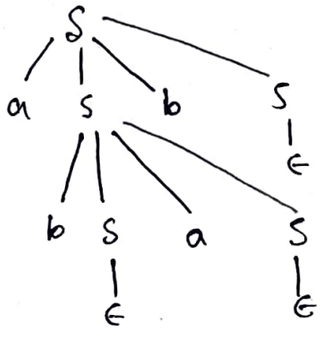


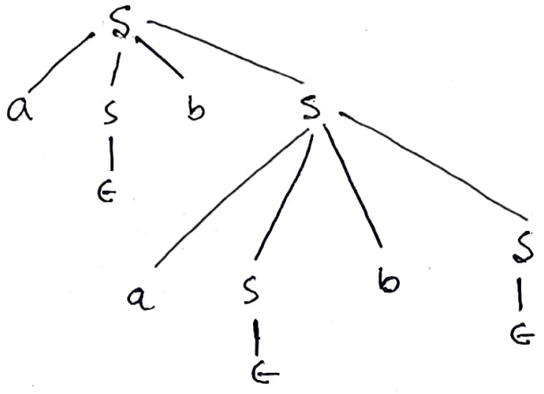
ambiguous.

~~w~~ { ~~Ambiguity~~ problem is undecidable.

eg. $S \to aSbS \mid bSaS \mid \in$.

$w = abab.$

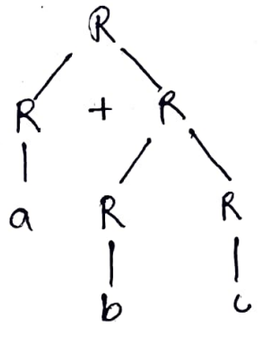


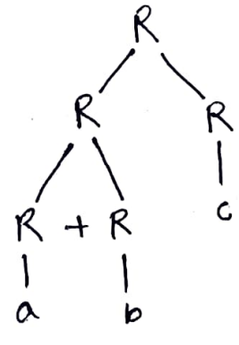


ambiguous.

eg. $R \to R+R \mid RR \mid R^* \mid a \mid b \mid c.$

$w = a+bc$





ambiguous.

→ Errors & their recovery in lexical analysis.

- Errors –
  1. Numeric literals that are too long
  2. Long identifiers
  3. Ill-formed numeric literals. | int a = $123
  4. Input characters that are not in the source language.

- Error recovery –
  1. Delete. : Unknown characters are deleted. Known as panic mode recovery.

  eg. "charr" corrected as "char" deleting 'r'

  2. Insert : An extra or missing character is inserted to form a meaningful token.

  eg. "cha" corrected as "char".

  3. Transpose. : Based on certain rules we can transpose 2 characters.

  eg. "whiel" can be corrected as "while"

  4. Replace : Based on replacing one character by another.

  eg. "chrr" can be corrected as "char" by replacing 'r' with 'a'.

→ Disambiguity rules. (Precedence, Associativity - left or right)

eg. E → E + E / E * E / id

↓

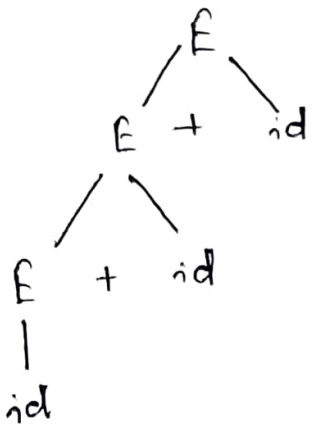
id + id + id (2 parse trees)

- Rules of associativity failed

id + id * id (2 parse trees)

- Operator precedence failed.

$E \rightarrow E + id \;/\; id$



✓ Left recursive →
* Left associative

$$\left\{\begin{array}{l} E \rightarrow E + T \;/\; T \\ T \rightarrow T * F \;/\; F \\ F \rightarrow id \end{array}\right.$$

Unambiguous.

Level in grammar.

+ ↘
*

(+ at higher level
* at lower level).

eg. bExp → bExp or bExp
ambiguous ↗ / bExp and bExp
/ not bExp / True / False.

$E \rightarrow E$ or F / F
$F \rightarrow F$ and G / G.
$G \rightarrow$ not G / True / False.

Unamb

Precedence.

¬ > ∧ > ∨

¬ unary
∧, ∨ left associative.

eg. $R \rightarrow R+R \;/ RR / R^* / a/b/c$.
(ambiguous)

$E \rightarrow E + T / T$.
$T \rightarrow TF / F$
$F \rightarrow F^* / a/b/c$.

Unamb.

Precedence

* > · > +.

+, · left associative

eg. G'

$A \rightarrow A \$ B / B$
$B \rightarrow B \# C / C$
$C \rightarrow C @ D / D$
$D \rightarrow d$.

Left recursive.
→ Left associative.

$:> $
:>
@ :> @.

precedence
$ <· # <· @

Rules
- Lowest precedence should be first derived.
- When left recursive/left associative form a derivation like $A \rightarrow A \$ B / B$

Imp

Highest precedence evaluated at the end of the grammar.

* eg. $E \to E * F / F + E / F$

$F \to F - F / id$

Decide the precedence & associativity rules. $\Rightarrow$

+, * same precedence

$- > +, *$

* left associative

+ right associative.

$* > *$

$+ < +$.

- **Recursion**

Left

$A \to A\alpha / \beta$

$\beta\alpha^*$ lang.

↓

Leads to infinite recursion.

So eliminate LR.

Right

$A \to \alpha A / \beta$.

$\alpha^*\beta$. lang.

Rule:

$A \to A\alpha / \beta$ LR

$\Updownarrow$

$\begin{cases} A \to \beta A' \\ A' \to \epsilon / \alpha A' \end{cases}$ RR

eg. $\underset{A}{E} \to \underset{A}{E} + T / \underset{\beta}{T}$

$\underset{A}{E} \quad \underset{A}{E} \quad \underset{\alpha}{+\,T}$

$\begin{cases} E \to TE' \\ E' \to \epsilon / +TE' \end{cases}$

eg. $S \to (L) / x$

$L \to L, S / S$

$\bar{A} \quad A\alpha \quad \beta$.

$\begin{cases} S \to (L) / x \\ L \to SL' \\ L' \to \epsilon / , SL' \end{cases}$

eg. $A \to A\alpha_1 / A\alpha_2 / A\alpha_3 / \cdots$

$/\beta_1 / \beta_2 / \beta_3 / \cdots$

$A \to \beta_1 A' / \beta_2 A' / \beta_3 A' / \cdots$

$A' \to \epsilon / \alpha_1 A' / \alpha_2 A' / \alpha_3 A' / \cdots$

eg. $\underset{A}{S} \to \underset{A}{S}\,\underset{\alpha}{0S1S} / \underset{\beta}{01}$

$\begin{cases} S \to 01S' \\ S' \to \epsilon / 0S1SS' \end{cases}$

• Grammar

Ambig. (X) → Unambig. (precedence, associativity)

Grammar

LR (X) → RR

$A \to A\alpha/\beta$ ⇓ $A \to \beta A'$, $A' \to \epsilon/\alpha A'$

Grammar

Non-deterministic

Deterministic. (generate new derivation having Common prefix)

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

Left factoring (to get rid of non-determinism). or common prefixes

eg. $S \to iEtS/iEtSeS/a$ Ambiguous (iEtiEtSeS)

$E \to b.$

⇓ Left factoring

$S \to iEtSS'/a$

$S' \to \epsilon/eS$ ① Deterministic.

$E \to b.$

Eliminating nondeterminism does not affect ambiguity.

eg. $S \to aSSbS/aSaSb/abb/b.$

Max common prefix as. or a

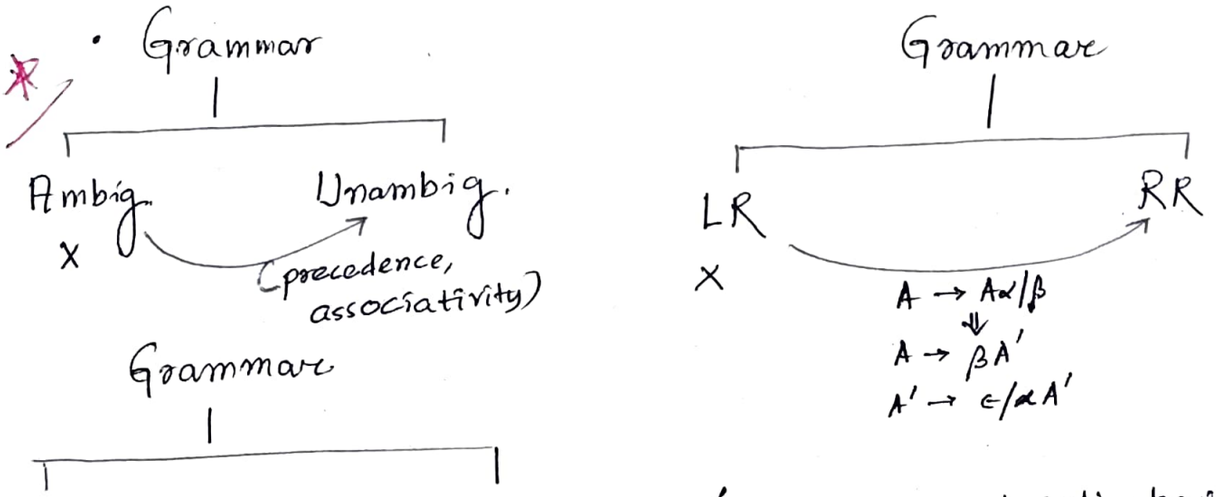
$S \to as'/b.$

$S' \to SSbS/SaSb/bb$

$S' \to SS''/S''$

$S'' \to SbS/aSb.$

Eliminating common prefix



Left factoring (to get rid of non-determinism or common prefixes)

eg. $S \rightarrow iEtS / iEtSeS / a$ Ambiguous (iEt iEtSeS)

$E \rightarrow b.$

Left factoring

$S \rightarrow iEtSS' / a$

$S' \rightarrow \epsilon / eS$ Deterministic.

$E \rightarrow b.$

Eliminating nondeterminism does not affect ambiguity.

eg. $S \rightarrow aSSbS / aSaSb / abb / b.$

Max common prefix aS. or a

$S \rightarrow aS' / b.$

$S' \rightarrow SSbS / SaSb / bb$

$S' \rightarrow SS'' / S''$

$S'' \rightarrow SbS / aSb.$

Eliminating common prefix

eg. Eliminating common prefix.

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

⇓

$S \rightarrow bSS' / a$

$S' \rightarrow SaaS / SaSb / b.$

$S' \rightarrow SaS'' / b$

$S'' \rightarrow aS / Sb$

Parsers Parsers

* Parsers (Amb. x)

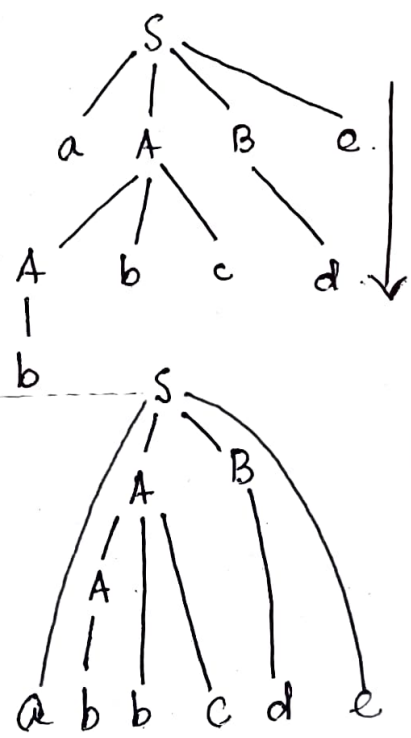
- Top down parsers (TDP)
  - TDP with full backtracking
    - Brute force method
  - TDP without backtracking (LRx NDx)
    - Recursive descent
    - Non-recursive descent
      - LL(1) ← (LRec^{n} x, Amb x, Non-det^{m} x)
- Bottom up parsers (BUP) (SR parsers Shift-reduce)
  - Operator precedence parser (Amb.)
  - LR parsers
    - LR(0) SLR(1) LALR(1) CLR(1)
    - (Amb. x)

eg. $S \rightarrow aABe$

$A \rightarrow Abc / b$

$B \rightarrow d.$

$w = abbcde$



TD Leftmost derivation

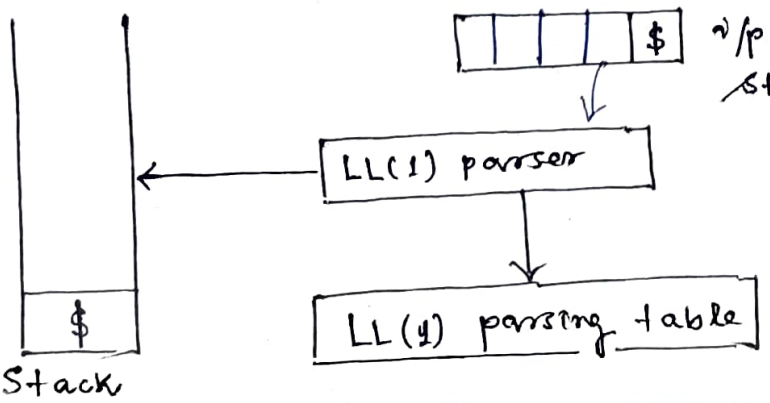
Which prod^{n} to choose for the variable?

∵ BU Reverse rightmost derivation

When to reduce a symbol?

* LL(1) parsers.

- Scan from left to right — L
- Using left most derivation — L
- 1 - No of lookaheads. — 1



(used to indicate end of i/p & it contains the string to be parsed followed by $)

Stack is used to contain a sequence of grammar symbols with a $ at the bottom.

- **First()** — Symbol that will be there at first of every derivation from grammar. (Only terminal)

eg. $S \rightarrow aABCD$

$A \rightarrow b/\epsilon$

$B \rightarrow c$

$C \rightarrow d$

$D \rightarrow e$.

first (S) = a

first (A) = b/e

first (B) = c.

- **Follow()** — Which is the terminal that can follow a variable in the process of derivation?

eg. ✓ Follow (S) = $.

$S \rightarrow ABCD$.

$A \rightarrow b/\epsilon$

$B \rightarrow c$

$C \rightarrow d$

$D \rightarrow e$.

follow (B) = d

Follow (A) = c

follow (C) = e

follow (D) = $

Follow(S) = ~~follow~~ first (A).

— First and follow

First and Follow sets are needed so that the parser can properly apply the needed prodⁿ rule at the correct position.

✓ first(α) is a set of terminal symbols that begin in strings derived from α.

eg. $A \rightarrow abc / def / gh$

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

Rules for calculating first().

1. For a production $X \rightarrow \epsilon$,

   first (X) = {ϵ}

2. For any terminal 'a',

   first (a) = {a}

3. For a production $X \longrightarrow Y_1 Y_2 Y_3$

- First(X)

If $\epsilon \notin$ First (Y_1), then First (X) = First (Y_1)

If $\epsilon \in$ First (Y_1), then First (X) = { First (Y_1) $- \epsilon$ } $\cup$ First $(Y_2 Y_3)$

- First $(Y_2 Y_3)$

If $\epsilon \notin$ First (Y_2), then First $(Y_2 Y_3)$ = First (Y_2)

If $\epsilon \in$ First (Y_2), then

First $(Y_2 Y_3)$ = { First $(Y_2) - \epsilon$ } $\cup$ First (Y_3).

Follow(α) is a set of terminal symbols that appear immediately to the right of α.

Rules for calculating follow()

1. For the start symbol S, place $ in Follow(S).
2. For any production $A \longrightarrow \alpha B$,

Follow (B) = follow (A).

$X \rightarrow tAc$
$A \rightarrow \alpha B$
$X \rightarrow t\alpha Bc$

3. For any production rule $A \rightarrow \alpha B \beta$.

- If $\epsilon \notin$ First (β), then

Follow (B) = First (β)

- If $\epsilon \in$ First (β), then

Follow (B) = { First $(\beta) - \epsilon$ } $\cup$ (Follow (A).)

NB.

1. ϵ may appear in the first function of a nonterminal. ϵ will never appear in the follow function of a nonterminal.
2. Before calculating the first & follow, eliminate Left recursion from the grammar, if present.

3. We calculate the follow function of a non-terminal by looking where it is present on the RHS of a production rule.

e.g.

	First	Follow
$S \rightarrow ABCDE$	{a, b, c}	{$}
$A \rightarrow a / \epsilon$	{a, ε}	{b, c}
$B \rightarrow b / \epsilon$	{b, ε}	{c}
$C \rightarrow c$	{c}	{d, e, $}
$D \rightarrow d / \epsilon$	{d, ε}	{e, $}
$E \rightarrow e / \epsilon$	{e, ε}	{$}

eg.

	First	Follow.
$S \rightarrow Bb / Cd$	{a, b, c, d}	{$}
$B \rightarrow aB / \epsilon$	{a, ε}	{b}
$C \rightarrow cC / \epsilon$	{c, ε}	{d}

eg.

Imp

	First	Follow *
$E \rightarrow TE'$	{id, (}	{$,)}
$E' \rightarrow +TE' / \epsilon$	{+, ε}	{$,)}
$T \rightarrow FT'$	{id, (}	{$,), +}
$T' \rightarrow *FT' / \epsilon$	{*, ε}	{$,), +}
$F \rightarrow id / (E)$	{id, (}	{*, +,), $}

eg.

Imp

	First *	Follow *
$S \rightarrow ACB / CbB / Ba$	{d, g, h, ε, b, a}	{$}
$A \rightarrow da / BC$	{d, g, h, ε}	{h, g, $}
$B \rightarrow g / \epsilon$	{g, ε}	{$, a, h, g}
$C \rightarrow h / \epsilon$	{h, ε}	{g, b, $, h}

eg.

	First	Follow.
$S \to aABb$.	$\{a\}$	$\{\$\}$
$A \to c / \epsilon$	$\{c, \epsilon\}$	$\{d, b\}$
$B \to d / \epsilon$	$\{d, \epsilon\}$	$\{b\}$

eg.

	First	* Follow
$S \to aBDh$	$\{a\}$	$\{\$\}$
$B \to cC$	$\{c\}$	$\{g, f, h\}$
$C \to bC / \epsilon$	$\{b, \epsilon\}$.	$\{g, f, h\}$
$D \to EF$	$\{g, f, \epsilon\}$	$\{h\}$
$E \to g / \epsilon$	$\{g, \epsilon\}$	$\{f, h\}$
$F \to f / \epsilon$.	$\{f, \epsilon\}$	$\{h\}$

eg.

$S \to A$
$A \to aB / Ad$
$B \to b$
$C \to g$

Grammar is left recursive.

After eliminating left recursion,

$S \to A$
$A \to aBA'$
$A' \to dA' / \epsilon$
$B \to b$
$C \to g$

	First	Follow
S	$\{a\}$	$\{\$\}$
A	$\{a\}$	$\{\$\}$ i.e. Follow (S)
A'	$\{d, \epsilon\}$	$\{\$\}$ i.e. Follow (A)
B	$\{b\}$	$\{First(A') - \epsilon\} \cup Follow(A) = \{d, \$\}$
C	$\{g\}$	NA

eg.

	First	Follow
$S \to (L) / a$	$\{(, a\}$	* $\{\$, , ,)\}$
$L \to SL'$	$\{(, a\}$	$\{)\}$
$L' \to , SL' / \epsilon$.	$\{, , \epsilon\}$	* $\{)\}$

Follow(L)

$\{\$\} \cup \{First(L') - \epsilon\} \cup$ Follow (L) $\cup$ follow(L'

eg. $S \rightarrow AaAb / BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$.

First (A) = {ϵ} First (B) = {ϵ}.

First (S) = {First (A) − ϵ} ∪ First (a) ∪ {First (B) − ϵ} ∪ First (b) = {a, b}

Follow (S) = {$}

Follow (A) = First (a) ∪ First (b) = {a, b}

Follow (B) = First (b) ∪ First (a) = {a, b}

✓ Eliminating left recursion

eg.

$E \rightarrow E + T / T$	$E \rightarrow TE'$
$T \rightarrow TXF / F$	$E' \rightarrow +TE' / \epsilon$
$F \rightarrow (E) / id$	$T \rightarrow FT'$
	$T' \rightarrow X FT' / \epsilon$ $\quad F \rightarrow (E) / id$

First (E) = First (T) = First (F) = {(, id}

First (F) = {(, id} First (E′) = {+, ϵ}

First (T′) = {X, ϵ}

First (T) = First (F) = {(, id}

Follow (E) = {$,)}

Follow (E′) = Follow (E) = {$,)}

Follow (T) = {First (E′) − ϵ} ∪ Follow (E) ∪ Follow (E′)

= {+, $,)}

Follow (T′) = Follow (T) = {+, $,)}

Follow (F) = {First (T′) − ϵ} ∪ Follow (T) ∪ Follow (T′)

= {X, +, $,)}.

→ Construction of LL(1) parsing table.

	First	Follow
$E \to TE'$	{id, (}	{$,)}
$E' \to +TE' / \epsilon$	{+, ϵ}	{$,)}
$T \to FT'$	{id, (}	{+,), $}
$T' \to *FT' / \epsilon$	{*, ϵ}	{+,), $}
$F \to id / (E)$	{id, ϵ}	{*, +,), $}

put under follow set

Rule:

All null prod^{n}s are put under follow set of the symbol & remaining prod^{n}s lie under first set of the symbol.

LL(1) parsing table –

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

Blank entries → error entry.

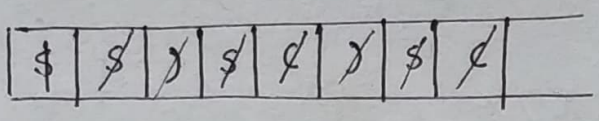
$S \to (S) / \epsilon$

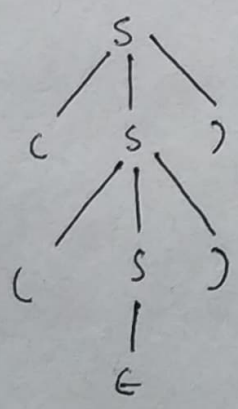
	(	)	$
S	$S \to (S)$	$S \to \epsilon$	$S \to \epsilon$

{ In the case of more than one entry in a cell, grammar may not be feasible for parsing.

$\omega = (())\$$

Stack





{ • Any grammar that is left recursive / non-deterministic can't be used for LL(1) parsing.

$S \to AaAb / BbBa$

$A \to \epsilon \qquad B \to \epsilon$

	a	b	$
S	$S \to AaAb$	$S \to BbBa$	
A	$A \to \epsilon$	$A \to \epsilon$	
B	$B \to \epsilon$	$B \to \epsilon$	

Every entry has exactly 1 grammar. So, we can construct parsing table from it.

LL(1) parsing example.

(1) $S \rightarrow aABb$

$A \rightarrow c / \epsilon$

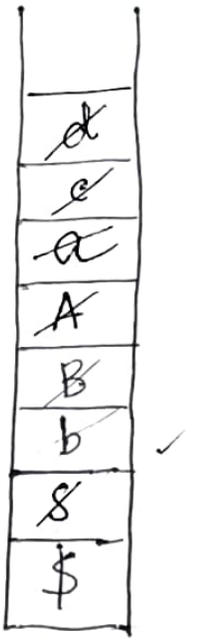
$B \rightarrow d / \epsilon$

$w = acdb$

LL(1) table

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

Stack	Input	Moves
$\$S$	$acdb\$$	$S \rightarrow aABb$
$\$bBAa$	$acdb\$$	matched
$\$bBA$	$cdb\$$	$A \rightarrow c$
$\$bBc$	$cdb\$$	-
$\$bB$	$db\$$	$B \rightarrow d$
$\$bd$	$db\$$	-
$\$b$	$b\$$	-
$\$$	$\$$	accepted.



(2) $S \rightarrow (S) \mid \epsilon$.

	(	)	$
S	$S \rightarrow (S)$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$

$w = (())$.

Stack	Input	Moves
$\$S$	$(())\$$	$S \rightarrow (S)$.
$\$)S($	$(())\$$	-
$\$)S$	$())\$$	$S \rightarrow (S)$
$\$))S($	$())\$$	-
$\$))S$	$))\$$	$S \rightarrow \epsilon$
$\$))$	$))\$$	-
$\$$	$\$$	acc. ✓

→ Check whether the grammars are LL(1) or not :

1. $S \to aSbS / bSaS / \epsilon$

	a	b	$
S	$S \to aSbS$, $S \to \epsilon$	$S \to bSaS$, $S \to \epsilon$	

2 entries ⇒ Not LL(1)

2. $S \to aABb$
$A \to c / \epsilon$ — c, b, d
$B \to d / \epsilon$ — d, b

LL(1)

3. $S \to A / a$ — a
$A \to a$ — a

Not LL(1) $\{a\} \cap \{a\}$

	a
S	$S \to A$, $S \to a$
A	$A \to a$

Ambiguous.

4. $S \to aB / \epsilon$ — a, $
$B \to bC / \epsilon$ — b, $
$C \to cS / \epsilon$ — c, $

LL(1)

5. $S \to AB$
$A \to a / \epsilon$ — a, $, b
$B \to b / \epsilon$ — b, $

LL(1)

6. $S \to aSA / \epsilon$ — a, c, $
$A \to c / \epsilon$ — c, $, c

Not LL(1)

	a	c	$
S	$S \to aSA$	$S \to \epsilon$	$S \to \epsilon$
A		$A \to c$, $A \to \epsilon$	

7. $S \to A$
$A \to Bb / Cd$ — a, b, c, d
$B \to aB / \epsilon$ — a, b
$C \to cC / \epsilon$ — c, d

Need not to check. One choice

LL(1)

8. $S \rightarrow aAa/\epsilon$ $\underline{a}$ $\$$ $\underline{a}$ Not LL(1)

$A \rightarrow abS/\epsilon$.

9. $S \rightarrow iEtSS'/a$ i, a

$S' \rightarrow eS/\epsilon$ e, e Not LL(1).

$E \rightarrow b$

* Recursive Descent Parser. 1 -fun^n for every variable

$E \rightarrow iE'$

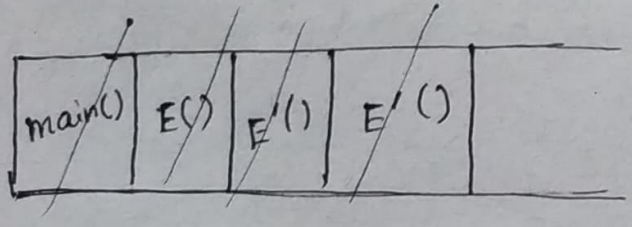
$E' \rightarrow +iE'/\epsilon$.

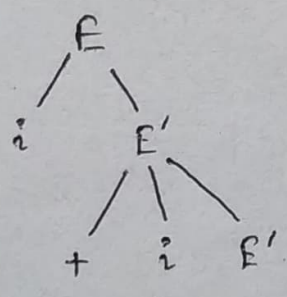
```
match (char t){
    if (l==t)
        l= getchar();
    else
        printf ("error");
}
main (){
    E();
    if (l== '$')
        printf ("Parsing
                success");
}
```

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

Stack by OS used.
No special stack.

eg. i+i$





$\epsilon \rightarrow$ neither var nor terminal

* Operator Precedence Parser.

- Operator grammar.

$E \rightarrow E+E \mid E*E \mid id$ ✓

- No prod^n in RHS is $\in$.
- 2 variables shouldn't be adjacent to each other.

$E \rightarrow EAE \mid id$
$A \rightarrow + \mid *$
} ✗ Not a operator grammar.

$S \rightarrow SAS \mid a$
$A \rightarrow bSb \mid b$
} ✗. $\Rightarrow$ $S \rightarrow SbSbS \mid SbS \mid a$, $A \rightarrow bSb \mid b$ } ✓

- Operator precedence parser can also work on ambiguous grammars. (using operator rel^n table).

- Operator relation table -

$E \rightarrow E+E \mid E*E \mid id$

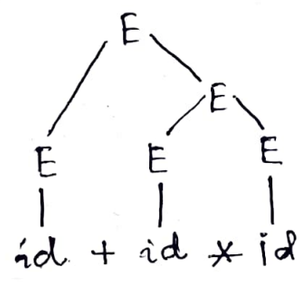
$+ \lessdot *$

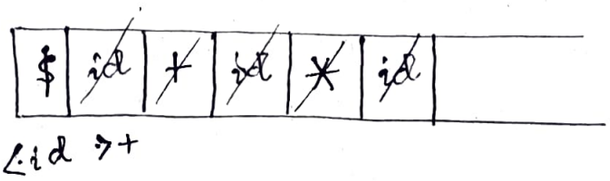
left \ right	id	+	*	$
id	=	$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	-

$ has least precedence

id has highest precedence

eg. id + id * id $



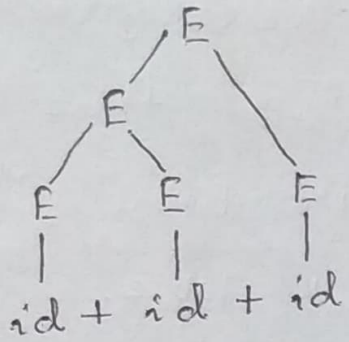


$\lessdot id \gtrdot +$

$\lessdot$: push $\gtrdot$: pop ⇝ reduce

eg. id + id + id $

$	~~id~~	~~+~~	~~id~~	~~*~~	~~id~~	

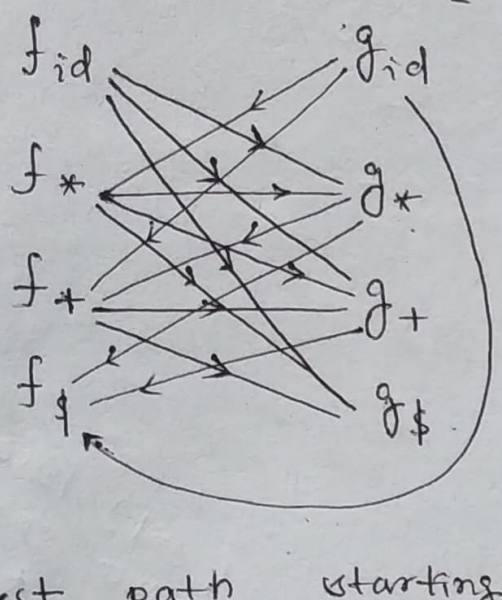


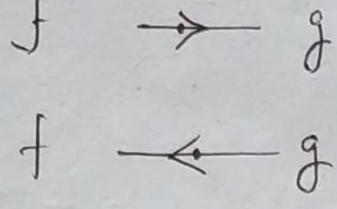
- Size of operator rel^n table $O(n^2)$

n = # operators.

This is a disadvantage.

So, we go for operator fun^n table.





If graph has no cycle, then only proceed.

Longest path starting from node.

$f_{id} \rightarrow g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$ 4

$g_{id} \rightarrow f_* \rightarrow g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$ 5

Operator fun^n table $\Rightarrow O(2n) \rightarrow O(n)$ size.

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

length of longest path as entry

→ Less size of table.

eg. $\frac{f*}{4} > \frac{g+}{1}$

$* > +$

eg. $\frac{f+}{2} > \frac{g+}{1}$

left + higher precedence

- One disadvantage of fun^n table is even though there are blank entries in operator rel^n table, there is no blank entry in fun^n table.

Error detecting capability of fun^n table is less. (even when no comparison, fun^n table has entry).

eg. P → SR/S

Para-graph R → bSR / bS X

S → wbS / w

w → L* w / L

L → id

Not operator grammar

P → SbSR / SbS / S

P → SbP / SbS / S

S → wbS / w

w → L* w / L

L → id

right recursive

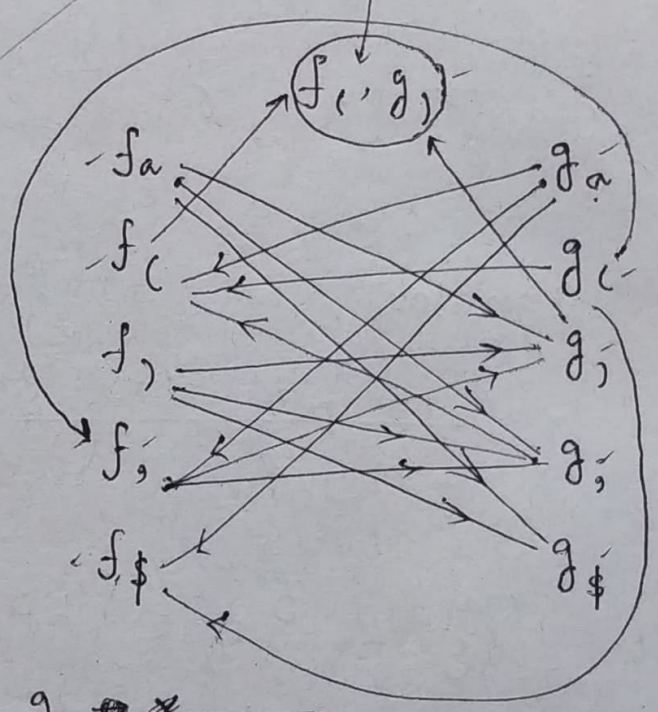
op. rel^n table

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

eg.

	a	(	)	,	$
a			⋗	⋗	⋗
(	⋖	⋖	=	⋖	
)			⋗	⋗	⋗
,	⋖	⋖	⋗	⋗	
$	⋖	⋖			

(and) have same precedence, so merge $f_($ & $g_)$ & make this transition



$f_a \to g_) \to f_(, g_)$ ✗ $g_a \to f_) \to g_) \to f_(, g_)$

$f_a \to g_\$$ ✗ $g_a \to f_) \to g_) \to f_(\to f_(, g_)$

$g_(\to f_) \to g_) \to f_(\to f_(, g_)$

$f_(\to f_(, g_)$

$g_) \to f_(, g_)$

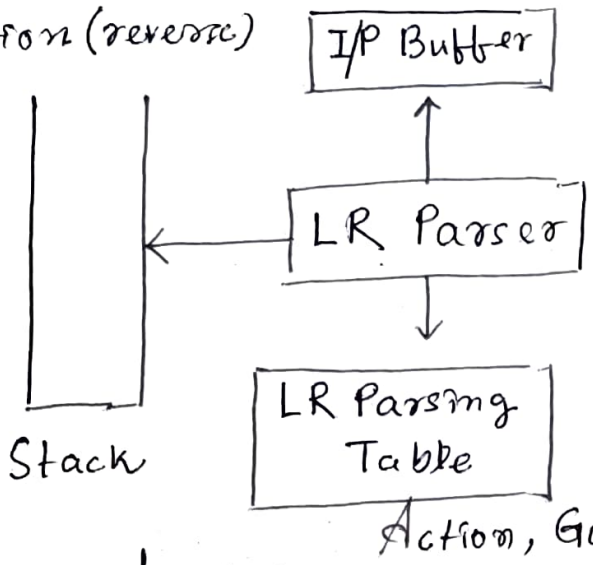
$f_1 \rightarrow g_1 \rightarrow f_0$ $\boxed{\rightarrow f_0, g_0}$

Thus,

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

* LR Parsers. Left-right scanning
Right most derivation (reverse)

- LR(0)
- SLR(1) Simple LR
- LALR(1) Look Ahead LR
- CLR(1) Canonical LR.



LR(0), SLR(1) } Canonical collection of LR(0) items.

LALR(1), CLR(1) } Canonical collection of LR(1) items.

0, 1
No. of input symbols of the look ahead used to make no. of parsing decision.

eg. G grammar

$S \rightarrow AA.$
$A \rightarrow aA/b.$

Add one prod^n $S' \rightarrow S$

Augmented grammar.

An LR(0) item is a production G with dot at some position on the right side of the production.

LR(0) items are useful to indicate that how much of the i/p has been scanned up to a given point in the process of parsing. In LR(0), we place the reduce node in the entire row.

Inserting • symbol at the first position for every production in G.

Canonical collection of LR(0) items.
- $S' \rightarrow \cdot S$
- $S \rightarrow \cdot AA$
- $A \rightarrow \cdot aA$
- $A \rightarrow \cdot b$

- $S \rightarrow \cdot AA$ seen nothing
- $S \rightarrow A \cdot A$ seen A
- $S \rightarrow AA \cdot$ seen everything

Think like • is parsing through string.

When we have seen everything on the RHS of a prod^n reduce it.

Add augment prod^n to the I0 state & compute the closure.

I0 = closure ($S' \rightarrow \cdot S$)

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

Add all prod^{n}s starting with S in to I0 state because $\cdot$ is followed by the non-terminal. So, the I0 state becomes

I0 = $S' \rightarrow \cdot S$
$S \rightarrow \cdot AA$

Add all prod^{n}s starting with A in modified I0 state because $\cdot$ is followed by the non-terminal. So, the I0 state becomes -

I0 = $S' \rightarrow \cdot S$
$S \rightarrow \cdot AA$
$A \rightarrow \cdot aA$
$A \rightarrow \cdot b$.

Now, what the $\cdot$ can see next $\rightarrow$ S, A, a, b.

(1) I1 = Goto (I_0, S) = closure ($S' \rightarrow S \cdot$)
= $S' \rightarrow S \cdot$

Prod^n is reduced, so close the state.

(1) I1 = $S' \rightarrow S \cdot$

(2) I2 = Goto (I0, A) = closure ($S \rightarrow A \cdot A$)
= $S \rightarrow A \cdot A$
$A \rightarrow \cdot aA$
$A \rightarrow \cdot b$.

from I2 { Goto (I2, a) = closure (A → a·A) = (same as I3)
Goto (I2, b) = closure (A → b·) = (same as I4)

③ I3 = Goto (I0, a) = closure (A → a·A).

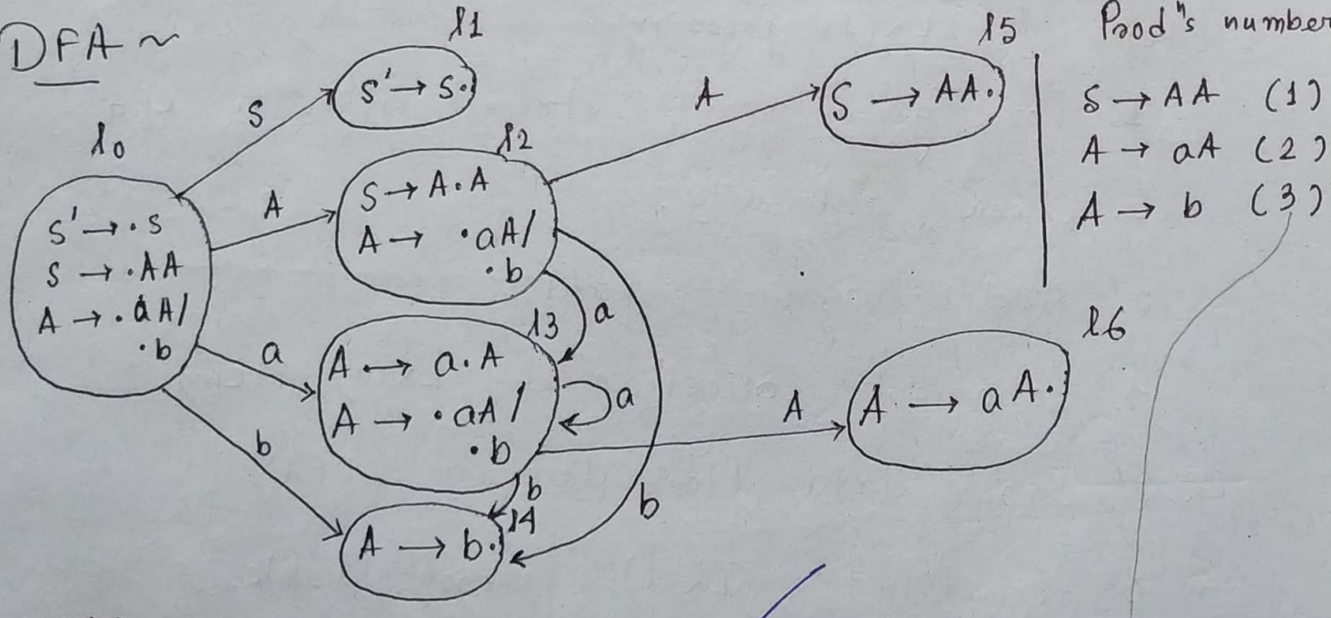
Add prod^n's starting with A in I3.

A → a·A
A → ·aA
A → ·b.

Goto (I3, a) = closure (A → a·A) = (same as I3)
Goto (I3, b) = closure (A → b·) = (same as I4)

④ I4 = Goto (I0, b) = closure (A → b·) = A → b·
I5 = Goto (I2, A) = closure (S → AA·) = S → AA·
I6 = Goto (I3, A) = closure (A → aA·) = A → aA·

DFA ~



LR(0) Table

If a state is going to some other state on a terminal then it corresponds to a shift move (S_K). If a state is going to some other state on a variable then Goto move. If a state contains the final item in the particular row then write reduce node completely.

States	Action			Goto	
	a	b	$	A	S
I_0	S_3	S_4		2	1
I_1			accept		
I_2	S_3	S_4		5	
I_3	S_3	S_4		6	
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

LR Parsing Example: $w = aabb$.

Stack	Input.
0	aabb $
0a3	abb $
0a3a3	bb $
0a3a3b4	b $
0a3a3b4 (reduce)	b$
0a3a3A6 (reduce. $I_6 \xrightarrow{b} r_2$ reduce using 2nd production)	b$.
0a3A6 ($I_6 \xrightarrow{b} r_2$)	b$.
0A2	b$.
0A2b4 ($I_4 \xrightarrow{\$} r_3$)	$.
0A2A5	$
~~0A2A~~	
0S1	$

Accepted.

Stack has 0 state (I_0) initially. Input ends with $.

See the top of stack (right most of stack) & the leftmost symbol of input.

Suppose, at first (0, a) $I_0 \xrightarrow{a} S_3$ (see table) Then, put the input symbol along with shifted state on top of stack. $\Rightarrow$ 0a3 And eliminate the symbol from input.

Case - St. IP

...4 b$

$I_4 \xrightarrow{b} r_3$

Says to reduce with production no. 3, i.e. $A \rightarrow b$.

Put A on top of stack. Don't remove b from input yet.

* After putting A, in stack, see last state and based on the table put the goto state. $I_3 \xrightarrow{A} I_6$

Example – parsing using the LR(0) table.

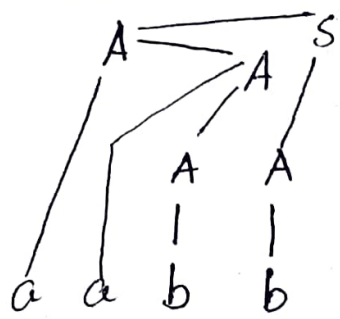
w = aabb.

a a b b $
↑ ↑ ↑ ↑ ↑

0	a	3	a	3	b	4	A	6	A	6	A	2	b	4	A	5	S	1

Stack

Accept.



(Example – VC. Joseph p104. Elements of CD).

- SLR(1). Difference in the parsing table with LR(0). To construct SLR(1) table, we use canonical collection of LR(0) items.

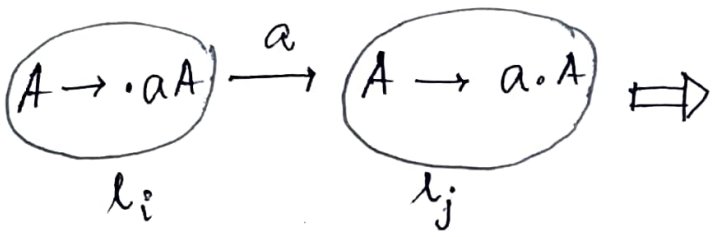
(In the SLR(1) parsing, we place the reduce move only in the follow of left hand side.)

Steps in SLR(1) passing:

1. For given input string, write a CFG.
2. Check ambiguity.
3. Add augment prod^n.
4. Canonical collection of LR(0) items.
5. Draw data flow diagram (DFA)
6. Construct SLR(1) parsing table.

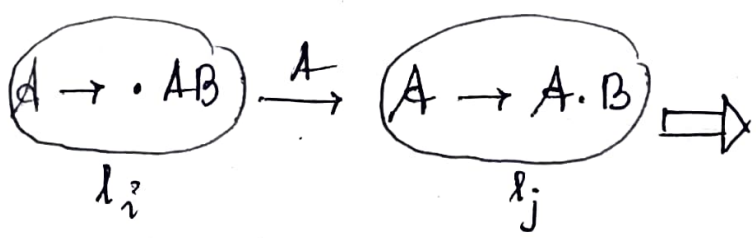
Table construction:

a) If a state (I_i) is going to some other state (I_j) on a terminal - then it corresponds to a shift move in the action part.



State	Action		Goto
	a	$	A
l_i	S_j		
l_j			

b) If a state (l_i) is going to some other state (l_j) on a variable then it corresponds to go to move in the Goto part.

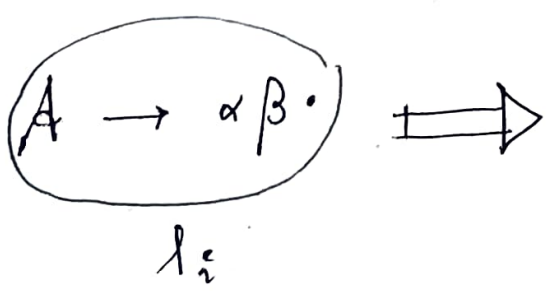


State	Action		Goto
	a	$	A
l_i			j
l_j			

c) If a state (l_i) contains the final item like $A \to ab.$ which has no transitions to the next state then the prod^n is known as reduce prod^n. For all terminals x on FOLLOW(A), write the reduce entry along with their prod^n numbers.

eg. $S \to .Aa$ 1
$A \to a\beta.$ 2

Follow (S) = {$}
Follow (A) = {a}.



State	Action			Goto	
	a	b	$	S	A
l_i	r_2				

29. Previous one

$S \longrightarrow AA$ (1) $S' \longrightarrow S$

$A \longrightarrow aA$ (2)

$A \longrightarrow b$. (3)

I4 $A \longrightarrow b.$

I5 $S \longrightarrow AA.$

I6 $A \longrightarrow aA.$

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

Follow (S) = {$}

~~Q6~~

States	Action			Goto	
	a	b	$	A	S
I0	S_3	S_4		2	1
I1			acc.		
I2	S_3	S_4		5	
I3	S_3	S_4		6	
I4	r_3	r_3	r_3		
I5			r_1		
I6	r_2	r_2	r_2		

→ If a state contains 2 reduce items, it's a reduce/reduce (r/r) conflict.

I5

$A \longrightarrow a.$
$B \longrightarrow b.$ r/r conflict.

If some column in the action part (a terminal for a state) contains both shift and reduce moves, it's a shift-reduce (s/r) conflict.

$A \longrightarrow b.$
$X \longrightarrow X.a$ s/r conflict

I6

Transition on a reduce by $A \longrightarrow b$.

→ To avoid some s/r & r/r conflicts, SLR(1) maintains Follow (LHS).

→ $A \to a.$
$B \to b.$

It's an r/r conflict only if Follow (A) and Follow (B) sets intersect, otherwise it's not a conflict.

$$\text{Follow}(A) \cap \text{Follow}(B) \neq \phi \Rightarrow \text{Conflict.}$$

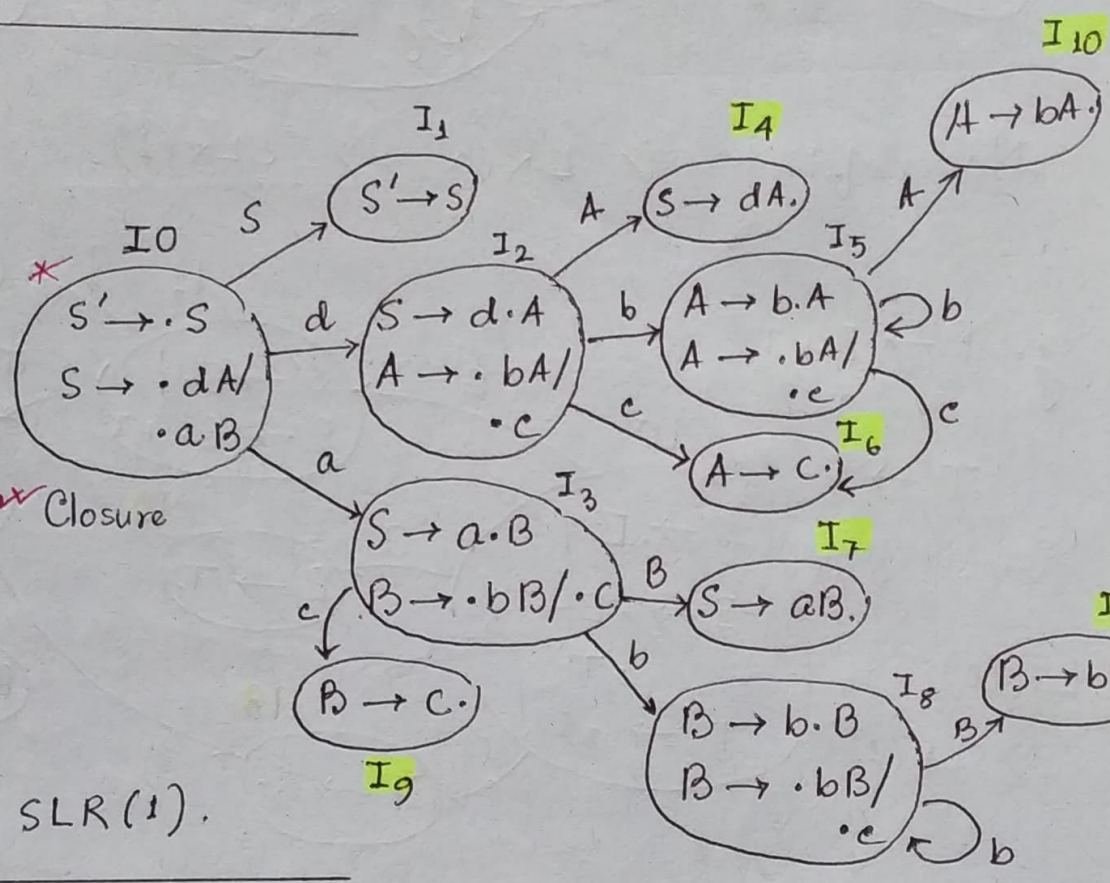
---

eg.

$S \to dA / aB$

$A \to \underline{b}A / c$

$B \to \underline{b}B / c$



LL(1) ✓

No conflicts

$\Rightarrow$ LR(0) $\Rightarrow$ SLR(1).

---

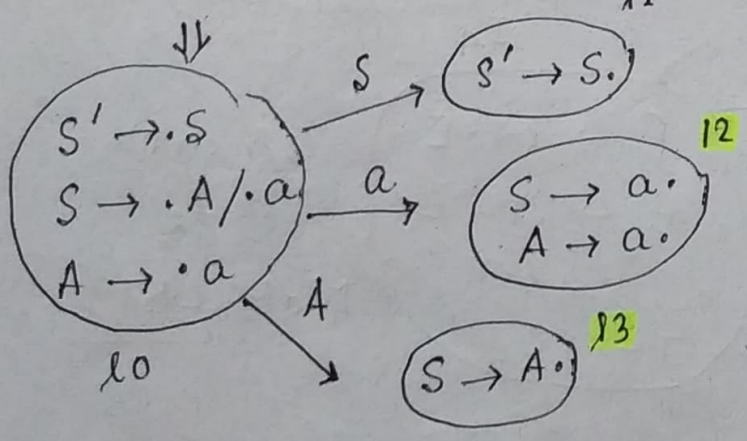
eg.

$S \to A / a$

$A \to a$

Ambiguous.

Not LL(1).



Not LR(0) (12 r/r conflict)

follow(S) = {$}

∩

follow(A) = {$}

≠

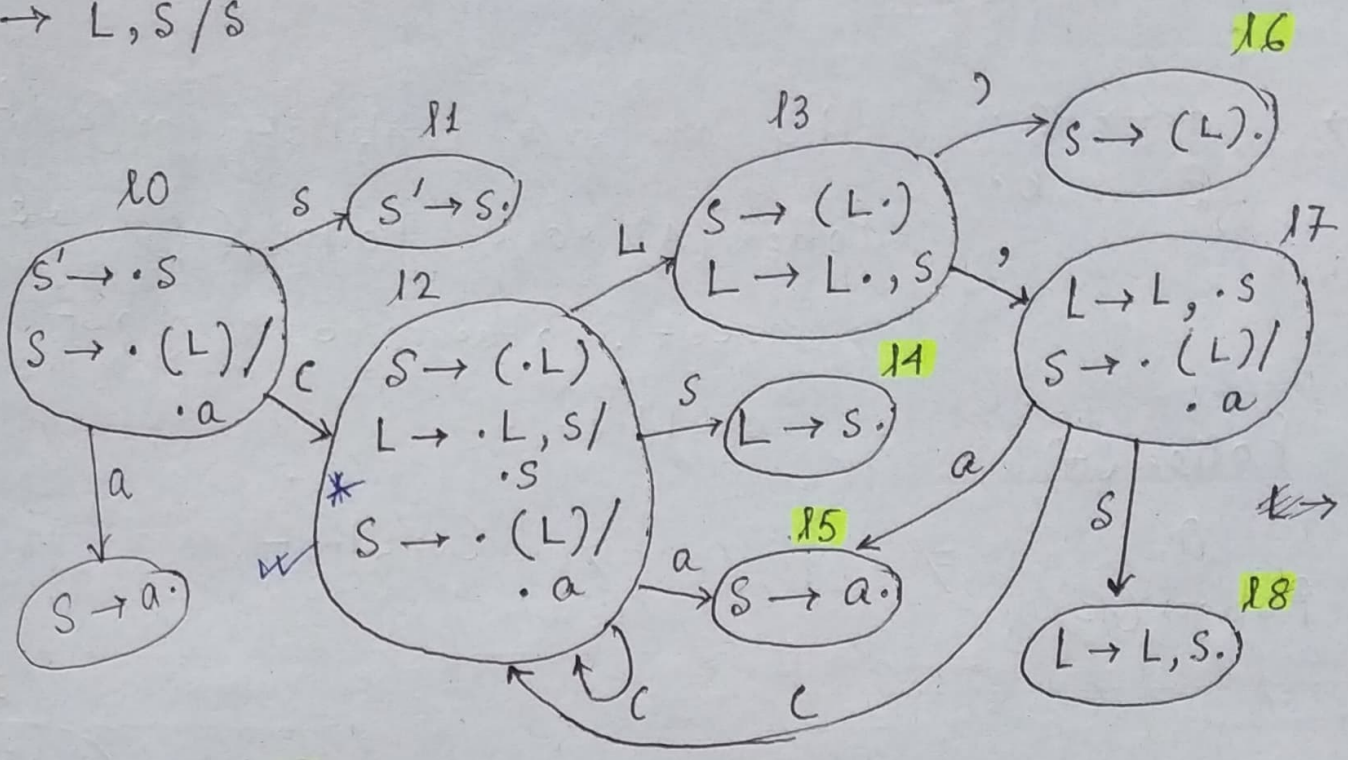
ϕ.

$\Rightarrow$ Not SLR(1).

eg.

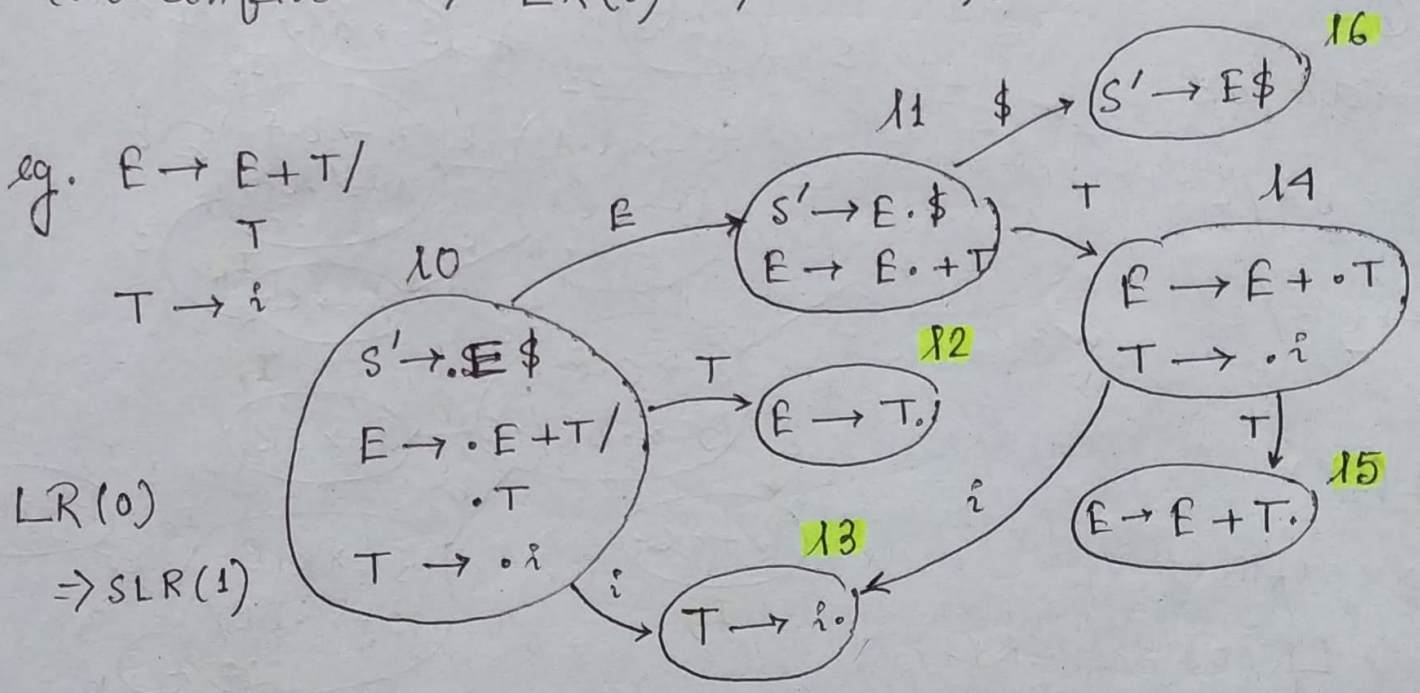
$S \to (L) / a$ Not LL(1) as left recursive.

$L \to L, S / S$



No conflict ⇒ LR(0) ⇒ SLR(1).

eg. $E \to E+T /$
T
$T \to i$

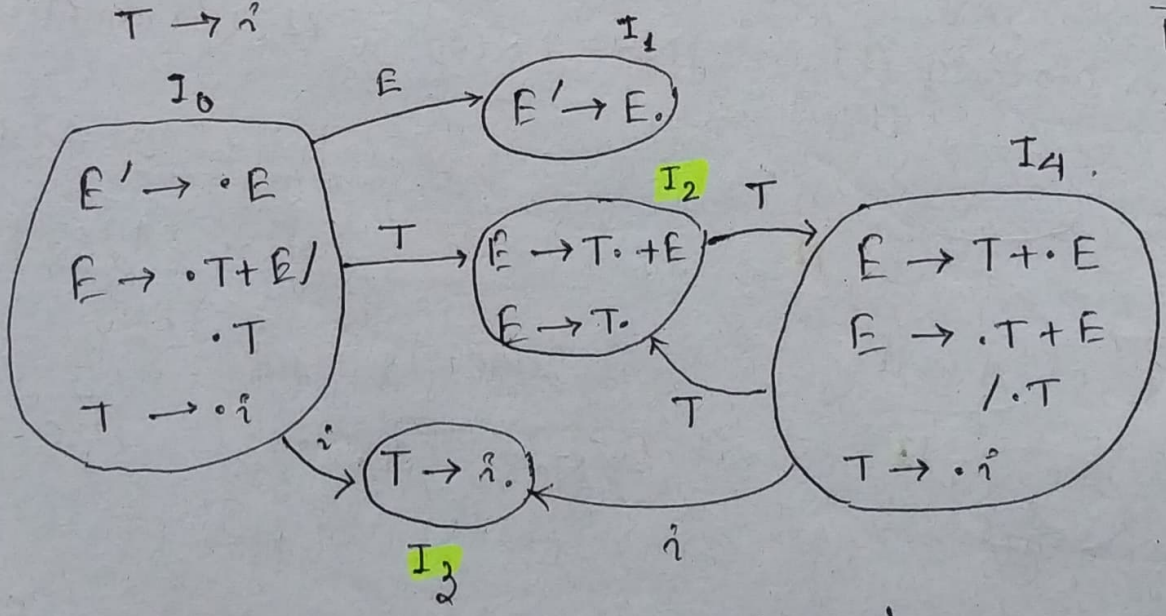


LR(0)
⇒ SLR(1)

eg. $E \to T+E / T$
$T \to i$

Not LL(1) as double entry

	+	i
E		{E → T+E, E → T}
T		T → i



s/r conflict in I_2

Not LR(0)

Follow(E) = {$}
Follow(T) = {+, $}

r_2 under $

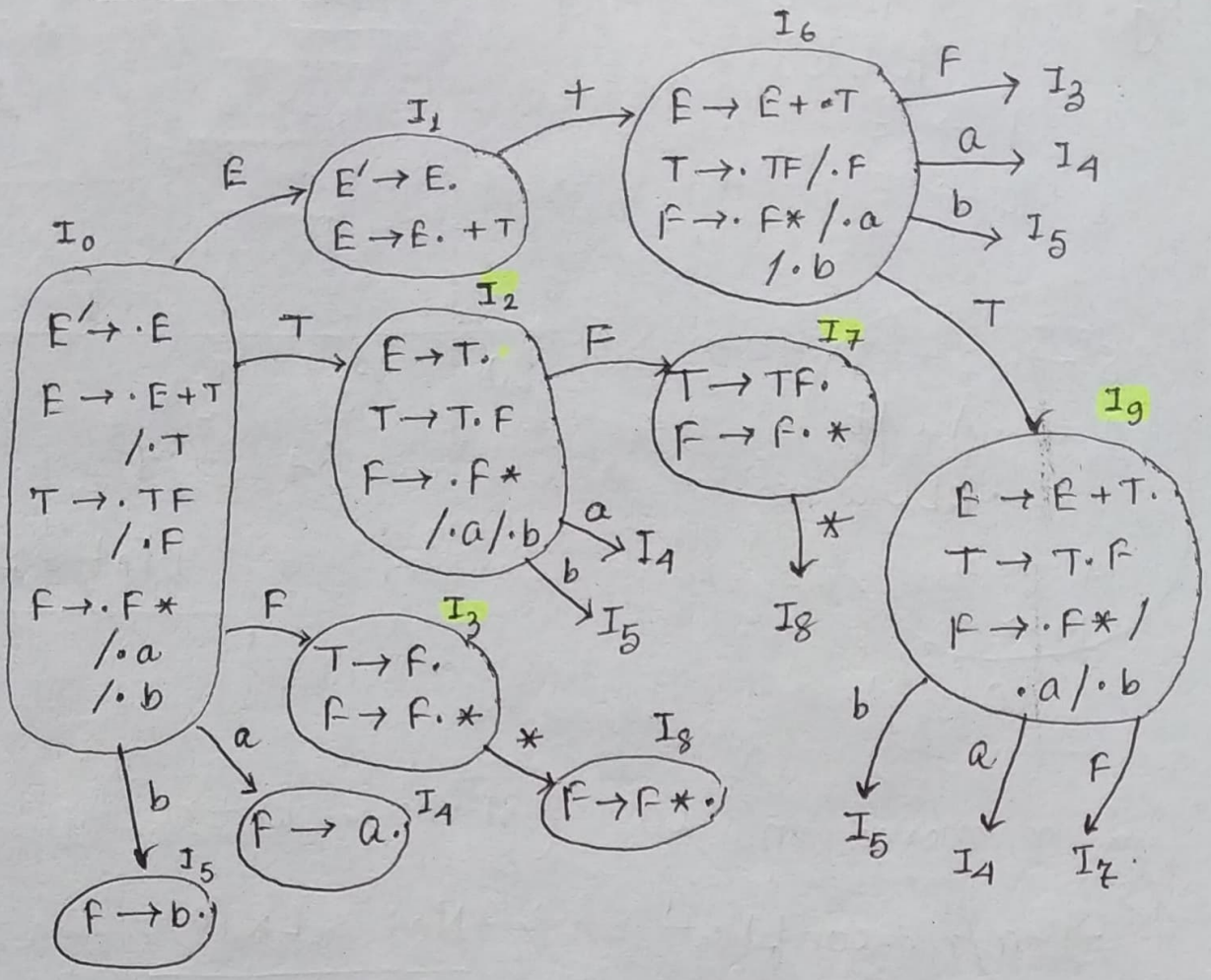
No s/r conflict.

SLR(1) ✓

eg.

Not LL(1) as left recursive.

$E \to E+T$ 1
$\quad / T$ 2
$T \to TF$ 3
$\quad / F$ 4
$F \to F*$ 5
$\quad / a$ 6
$\quad / b$ 7



In I_2, $E \to T.$ Final item.

$T \to T.F$ Shift move (not conflict, as in the Goto part)

$F \to \cdot F*$ Shift move (not conflict, in Goto)

$F \to \cdot a$ Conflict (in action part). S/R conflict

$\Rightarrow$ Not LR(0).

$\downarrow$

$\begin{cases} E \to T. \\ F \to \cdot a \end{cases}$

Follow(E) = {+, $}

Follow(T) = {+, $, a, b}

Follow(F) = {+, *, $, a, b}.

For SLR(1), conflict happens in action part.

Inadequate States	a	b	+	*	$	
I_2	S_4	S_5	r_2		r_2	No conflict.
I_3	r_4	r_4	r_4	S_8	r_4	"
I_7	r_3	r_3	r_3	S_8	r_3	"
I_9	S_4	S_5	r_1		r_1	". Beauty!

SLR(1) ✓

eg. $S \to AaAb / 1$
$BbBa$ 2
$A \to \epsilon$ 3
$B \to \epsilon$. 4

G is LL(1) ✓

$A \to \cdot\epsilon$ }
$A \to \epsilon\cdot$ } ↙
$A \to \cdot$

$S' \to \cdot S$
$S \to \cdot AaAb /$
$\cdot BbBa$
$A \to \cdot$
$B \to \cdot$
I0

	a	b	$
I0	r_3/r_4	r_3/r_4	

conflict

Not SLR(1)

2 r moves in I0 ($A \to \cdot$, $B \to \cdot$)

⇒ r/r conflict ⇒ Not LR(0).

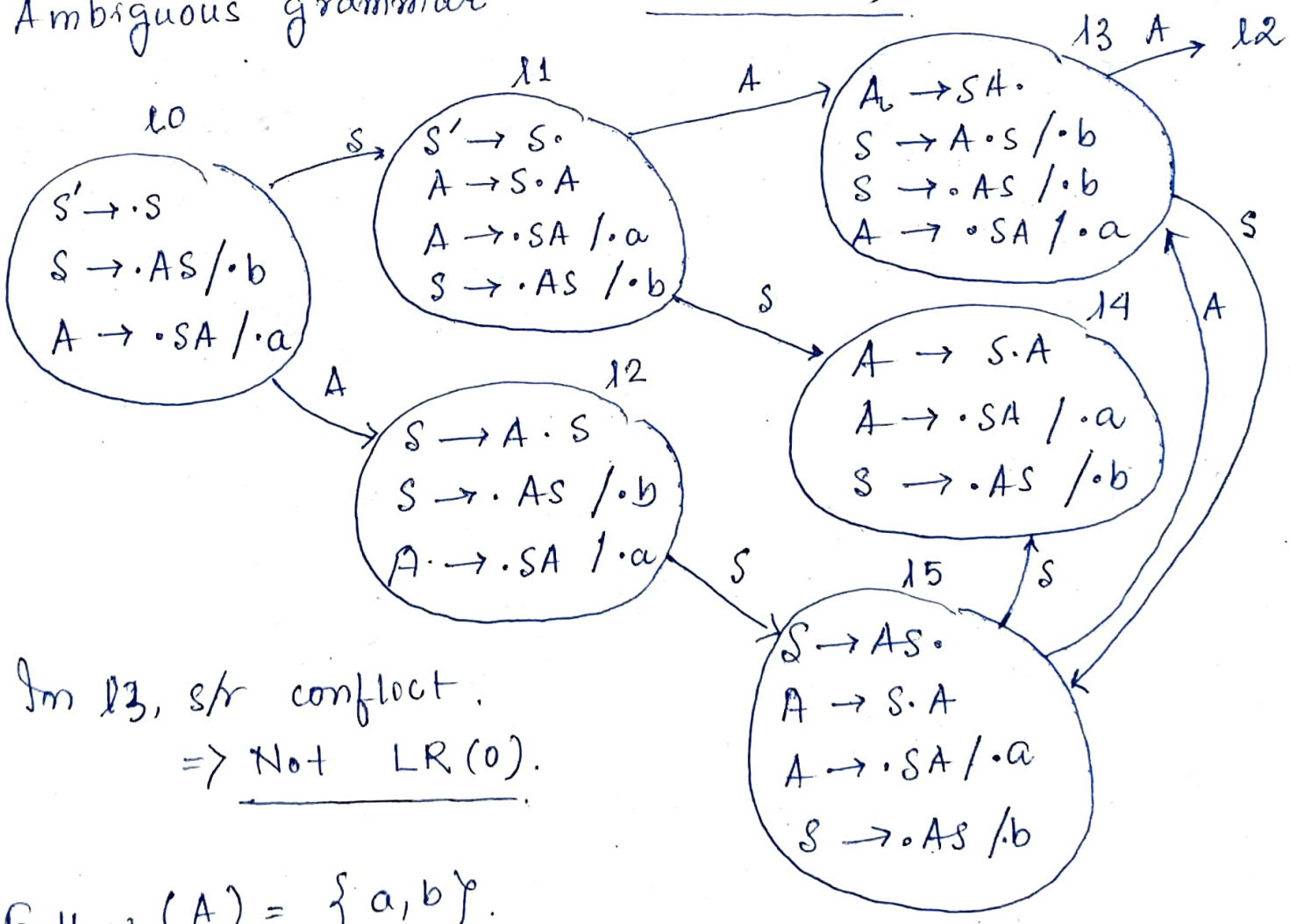
eg. $S \to As / b$
$A \to SA / a$.

Ambiguous grammar

	a	b
S	$S \to AS$	$S \to AS$ / $S \to b$

two ~~two~~ entries.

Not LL(1)



In I3, s/r conflict.
⇒ Not LR(0).

Follow(A) = {a, b}.

In I3, $A \to SA\cdot$ on a, b } conflict
$S \to \cdot b$ on b.

Not SLR(1).

eg. **Not LL(1)** — 1 and 3 prod^{n}s in same cell (S, d)

$S \to Aa$ 1
$/ bAC$ 2
$/ dc$ 3
$/ bda$ 1
$A \to d$ 5

I_0:
$S' \to .S$
$S \to .Aa$
$/ .bAc$
$/ .dc$
$/ .bda$
$A \to .d$

$\xrightarrow{d}$

I_1:
$S \to d.c$
$A \to d.$

$c \in Follow(A)$

	a	b	c	d	$
I1	r2			s/r3	

Not LR(0). conflict in I1.

Not SLR(1).

14

- **CLR(1) Parsing**

CLR refers to canonical lookahead. CLR parsing uses the canonical collection of LR(1) items to build the CLR(1) parsing table. CLR(1) parsing table produces more number of states as compared to the SLR(1) parsing. In the CLR(1), we place the reduce node only in the lookahead symbols.

LR(1) item

LR(1) item is a collection of LR(0) items & a look-ahead symbol.

LR(1) item = LR(0) item + lookahead ✓

Lookahead is used to determine that where we place the final item. The lookahead always add $ symbol for the augment production.

LR item : $A \to \alpha . \beta$, a Lookahead 'a' has no effect where β is not $\in$. But an item of the form $[A \to \alpha., a]$ calls for a reduction by $A \to \alpha$ only if next symbol (i/p) is a.

Set of such 'a's will be a subset of follow(A), but it could be a proper subset (less possibility of conflict).

LR(1) item :

If $A \longrightarrow \alpha \cdot B\beta, a$ is in closure(I) and $B \longrightarrow \gamma$ is a production rule of G ; then $B \longrightarrow \gamma, b$ will be in the closure(I) for each terminal b in first (βa).

$A \longrightarrow \cdot \underline{BC, a}$ $\quad C \longrightarrow \underline{+P}$

$B \longrightarrow \gamma, +$ ___ First (Ca)

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

$$\begin{cases} S' \to \cdot S, \$ \\ S \to \cdot AA, \$ \\ A \to \cdot aA, a/b \\ A \to \cdot b, a/b \end{cases}$$

I0

$I0 = \text{closure}(S' \to \cdot S)$

$= S' \to \cdot S, \$$
$S \to \cdot AA, \$$
$A \to \cdot aA, a/b$
$A \to \cdot b, a/b.$

I1 $I1 = \text{Goto}(I0, S) = \text{closure}(S' \to S\cdot, \$)$

$= S' \to S\cdot, \$$

I2 $I2 = \text{Goto}(I0, A) = \text{closure}(S \to A\cdot A, \$)$

$= S \to A\cdot A, \$$
$A \to \cdot aA, \$$
$A \to \cdot b, \$$

I3 $I3 = \text{Goto}(I0, a) = \text{closure}(A \to a\cdot A, a/b)$

$= A \to a\cdot A, a/b$
$A \to \cdot aA, a/b$
$A \to \cdot b, a/b$

$\text{Goto}(I3, a) = \text{closure}(A \to a\cdot A, a/b) = (\text{same as } I3)$

$\text{Goto}(I3, b) = \text{closure}(A \to b\cdot, a/b) = (\text{same as } I4)$

I4 $I4 = \text{Goto}(I0, b) = \text{closure}(A \to b\cdot, a/b)$

$= A \to b\cdot, a/b$

I5 $I5 = \text{Goto}(I2, A) = \text{closure}(S \to AA\cdot, \$)$

$= S \to AA\cdot, \$$

I6 $I6 = \text{Goto}(I2, a) = \text{closure}(A \to a\cdot A, \$)$

$= A \to a\cdot A, \$$
$A \to \cdot aA, \$$
$A \to \cdot b, \$$

$\text{Goto}(I6, a) = \text{closure}(A \to a\cdot A, \$) = (\text{same as } I6)$

$\text{Goto}(I6, b) = \text{closure}(A \to b\cdot, \$) = (\text{same as } I7)$

I_7 = Goto (I_2, b) = closure $(A \to b\cdot\ ,\ \$)$

= $A \to b\cdot\ ,\ \$$

I_8 = Goto (I_3, A) = closure $(A \to aA\cdot\ ,\ a/b)$

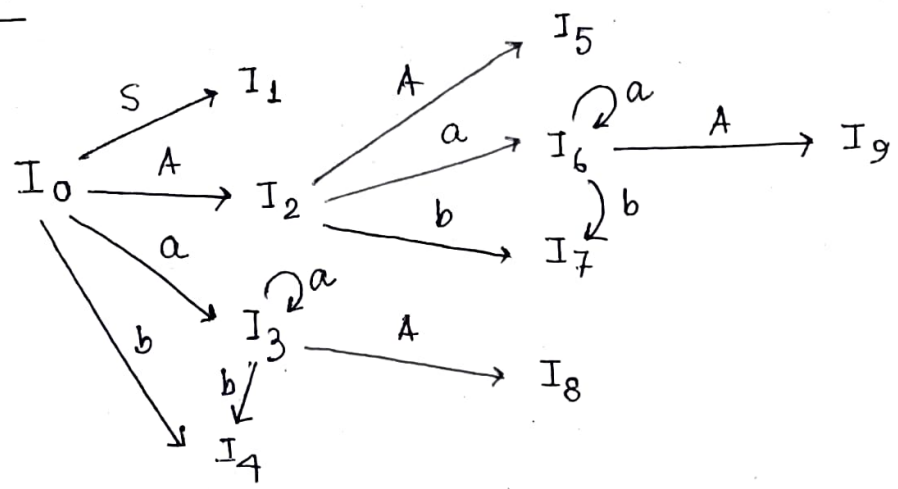
= $A \to aA\cdot\ ,\ a/b$

I_9 = Goto (I_6, A) = closure $(A \to aA\cdot\ ,\ \$)$

= $A \to aA\cdot\ ,\ \$$

DFA

Data flow diagram.



CLR(1) parsing table.

States	Action a	Action b	Action $	Goto S	Goto A
I_0	S_3	S_4		1	2
I_1			Accept		
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			9
✓ I_7			R_3		
✓ I_8	R_2	R_2			
✓ I_9			R_2		

action $(I_4, a) = R_3$
action $(I_4, b) = R_3$

action $(I_5, \$) = R_1$

action $(I_6, a) = S_6$

action $(I_7, \$) = R_3$

action $(I_8, a) = R_2$
action $(I_8, b) = R_3$

action $(I_9, \$) = R_2$.

- Placement of shift nodes in CLR(1) parsing table is same as the SLR(1) parsing table. Only difference is in the placement of reduce node.

- **LALR(1) Parsing**

LALR refers to lookahead LR.

The LR(1) items which have same prod^{n}s but different lookahead are combined to form a single set of items. LALR(1) and CLR(1) parsing are same, only difference in the parsing table.

eg. Previous. $S \rightarrow AA \quad A \rightarrow aA / b.$

I_3 and I_6 are same in their LR(0) items but differ in their lookahead. So, we combine.

$$I_{36} = \{ A \rightarrow a \cdot A, \ a/b/\$ \\ A \rightarrow \cdot aA, \ a/b/\$ \\ A \rightarrow \cdot b, \ a/b/\$ \}$$

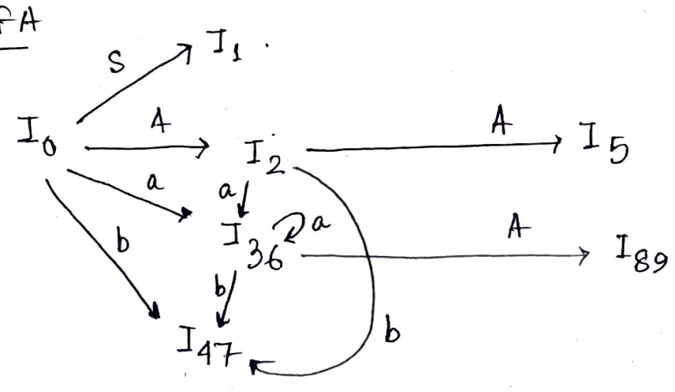
Same for I_4, I_7

$I_{47} = \{ A \rightarrow b \cdot, a/b/\$ \}$

Same for I_8, I_9

$I_{89} = \{ A \rightarrow aA \cdot, a/b/\$ \}$

DFA



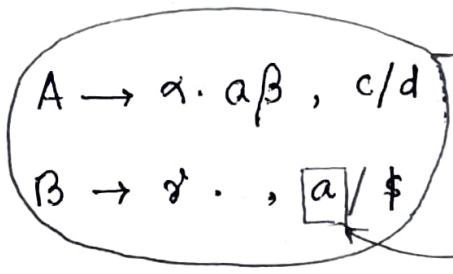
Reduced no. of states than CLR(1).

LALR(1) parsing table.

States	a	b	$	S	A
I_0	S_{36}	S_{47}		1	2
I_1			Accept		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	R_3	R_3	R_3		
I_5			R_1		
I_{89}	R_2	R_2	R_2		

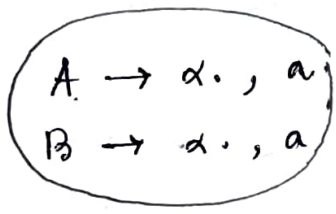
- Conflicts in CLR(1) and LALR(1).

LR(1) items



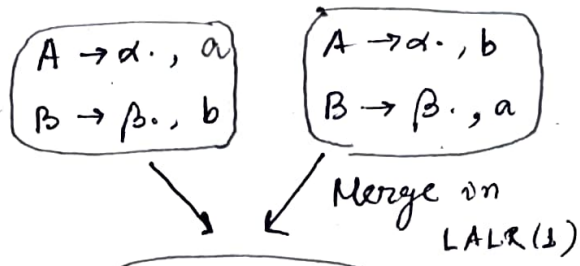
If grammar is not CLR(1) then it is not LALR(1). Because, we reduce the size of the table but not the conflicts in LALR(1) parser.

S/R conflict



R/R conflict

If grammar is CLR(1), then it may or may not be LALR(1).



eg. $S \to AaAb \mid BbBa$

$A \to \epsilon$

$B \to \epsilon$.

LL(1) ✓

LR(0) X

SLR(1) X

CLR(1) ✓

LALR(1) ✓

LR(0)

$S' \to \cdot S$

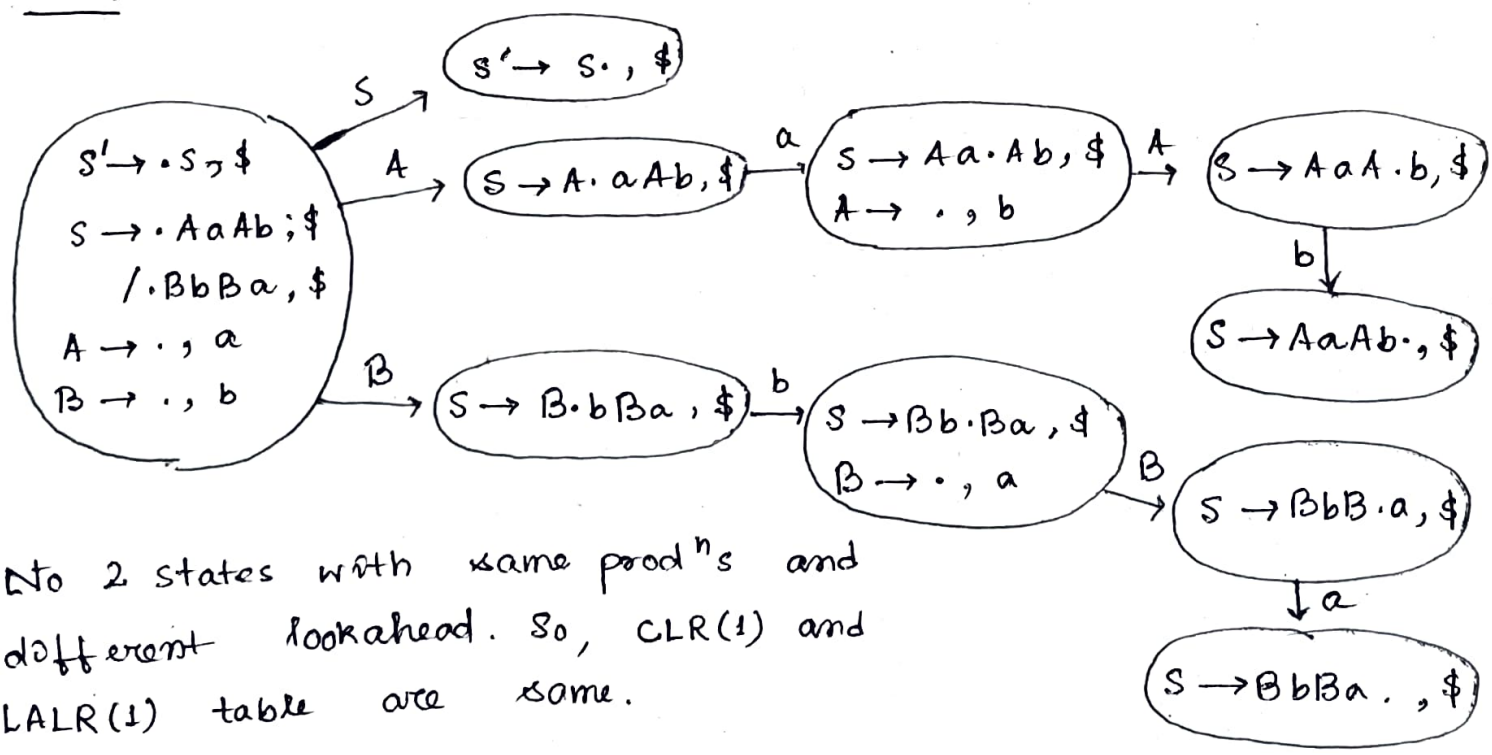
$S \to \cdot AaAb \mid \cdot BbBa$

$A \to \cdot$ placed under follow(A) = {a,b}

$B \to \cdot$ placed in follow(B) = {a,b}

Not SLR(1)

LR(1)

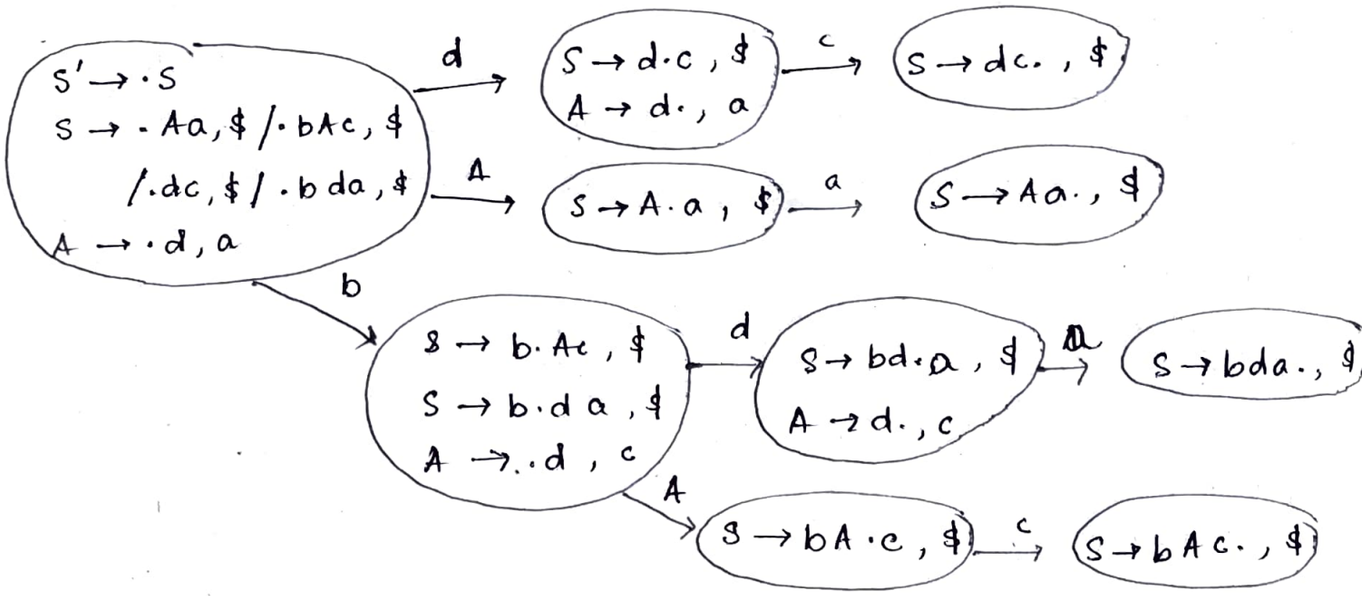


No 2 states with same prod^{n}s and different lookahead. So, CLR(1) and LALR(1) table are same.

No conflicts.

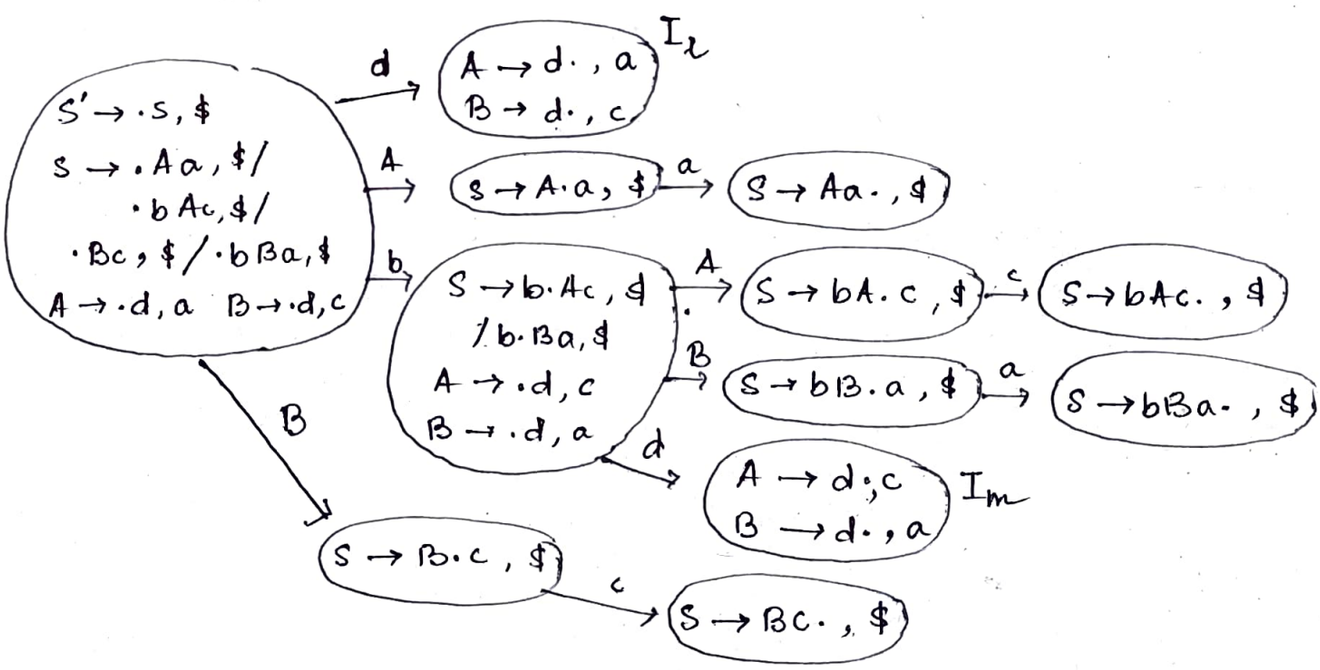
eg $S \to Aa/bAc/dc/bda$
$A \to d$

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



eg. $S \to Aa/bAc/Bc/bBa$
$A \to d \quad B \to d$

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



I_l : $A \to d\cdot, a$; $B \to d\cdot, c$

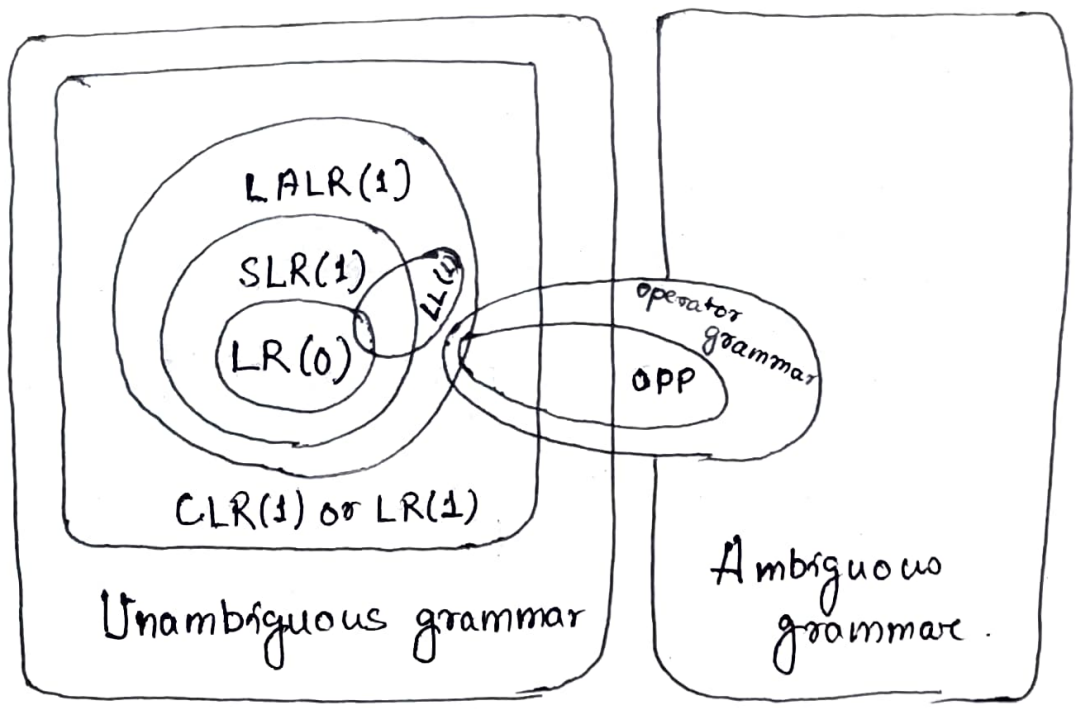
I_m : $A \to d\cdot, c$; $B \to d\cdot, a$

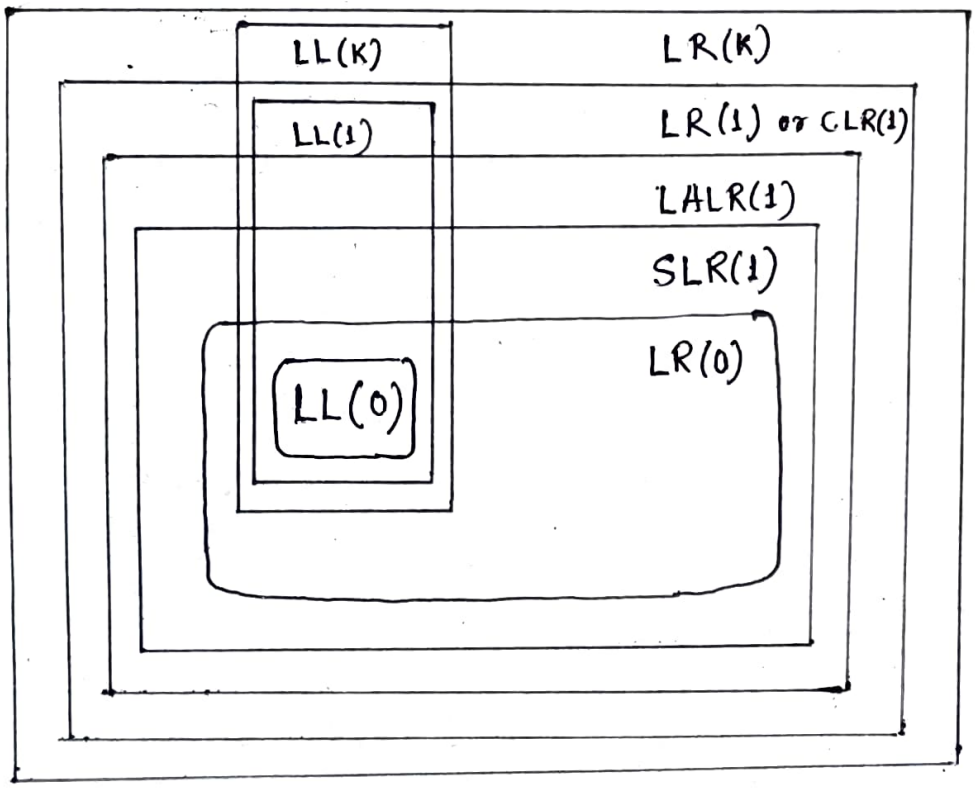
$A \to d\cdot, a/c$
$B \to d\cdot, a/c$

R/R conflict

Not LALR(1)

* Comparison of the parsers.





Unambiguous Grammar

Ambiguous grammar.

- LR(0) ⊂ SLR(1) ⊂ LALR(1) ⊂ CLR(1)
- LL(1) ⊂ LR(1) or CLR(1)
- If, for a grammar G, LALR(1) can be constructed then CLR(1) parser can also be constructed.
- If CLR(1) grammar parser can be constructed for a grammar, then we may or may not construct LALR(1) parser.
- No. of entries in the LALR(1) parser ≤ no. of entries in the CLR(1) parser.

- No. of entries in the LALR(1) parsing table is equal to the no. of entries in the SLR(1) parsing table.

- #states (SLR(1)) = #states (LALR(1))
  $\leq$ # states (CLR(1))

- CLR(1) parsers are more powerful, efficient than any other parser.

eg $S \to A$
$A \to AB/\epsilon$
$B \to aB/b$

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

Closure ($S \to \cdot A, \$$)
$S' \to \cdot S, \$$
$S \to \cdot A, \$$
$A \to \cdot AB, \$$
$/\cdot, \$$

$\Rightarrow$

$S' \to \cdot S, \$$
$S \to \cdot A, \$$
$A \to \cdot AB, \$/a/b$
$/\cdot, \$/a/b$

$A \to \cdot AB, a/b$
$/\cdot, a/b$

→ We have to add them too because there can be new lookaheads for the prodn $A \to \cdot AB, \$$ $/\cdot, \$$. So, we add new lookaheads a, b as these are

First ($B\$$) = First (B).

NOTE

LL(k): For a grammar to be LL(k), we must be able to recognise the use of a production by seeing only the first k symbols of what its RHS derives.

LR(k): For a grammar to be LR(k), we must be able to recognise the use of a production by having seen all of what is derived from its RHS with k more symbols of lookahead.

eg. $E \to E+T/T \quad T \to T*F/F \quad F \to i$

$E' \to \cdot E, \$$

$E \to \cdot E+T, \$$ — first $(+T\$) = +$

$/ \cdot T, \$$

$E \to \cdot E+T, +$

$/ \cdot T, +$

$T \to \cdot T*F, \$/+$ — first $(*F\$+) = *$

$/ \cdot F, \$/+$

$T \to \cdot T*F, *$

$/ \cdot F, *$

$F \to \cdot i, \$/+/*$

$\Rightarrow$

$E' \to \cdot E, \$$

$E \to \cdot E+T, \$/+$

$/ \cdot T, \$/+$

$T \to \cdot T*F, \$/+/*$

$/ \cdot F, \$/+/*$

$F \to \cdot i, \$/+/*$

. E → $E' \to E\cdot, \$$; $E \to E\cdot+T, \$/+$

$+$ → $E \to E+\cdot T, \$/+$; $T \to \cdot T*F, \$/+/*$; $/ \cdot F, \$/+/*$; $F \to \cdot i, \$/+/*$

T → $E \to T\cdot, \$/+$; $T \to T\cdot*F, \$/+/*$

F → $T \to F\cdot, \$/+/*$

i → $F \to i\cdot, \$/+/*$

T → $E \to E+T\cdot, \$/+$; $T \to T\cdot*F, \$/+/*$

OPP $<$ LL(1) $<$ LR(0) $<$ SLR(1) $\leq$ LALR $\leq$ CLR(1).

+(CD folder)
photo

Q. Consider SLR(1) and LALR(1) tables for a CFG. Then

(a) Goto of both tables may be different

✓(b) Shift entries are identical in both.

✓(c) Reduce entries in tables may be different

✓(d) Error entries in tables may be different.

Q. $S \to CC$
$C \to cC \mid d$

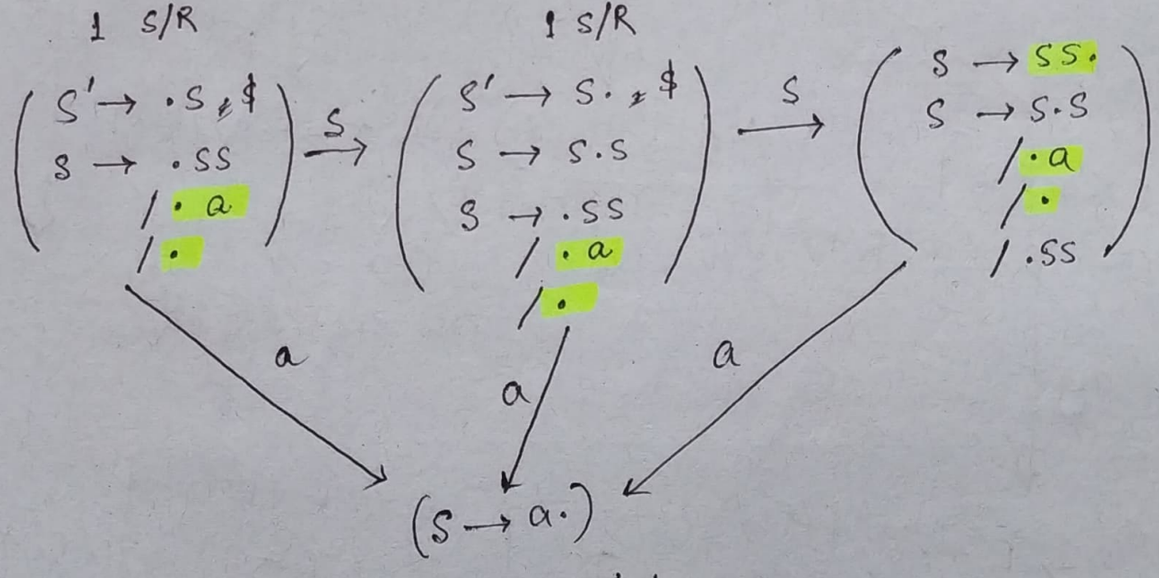
✓a) LL(1)
b) SLR(1) but not LL(1)
c) LALR(1) but not SLR(1)
d) CLR(1) but not LALR(1)

Q. Find # SR, RR conflicts in the DFA with LR(0) items.

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

or # inadequate states

1 S/R 1 RR



$(S \to a.)$

3 S/R & 1 R/R conflict

4 inadequate states.

8. $E \to E + n / E * n / n$

For the string $n + n * n$, the handles in right sentencial form of reduction are

$(n, E + n, E * n)$

$E \Rightarrow (E * n)$

$\Rightarrow (E + n) * n$

$\Rightarrow (n) + n * n$

$n + n * n$

$E + n * n$

$E * n$

E.

- A right sentencial form is a sentencial form that occurs in the rightmost derivation of some sentence.

$E \to E + n / E * n / n$

n terminal

$n + n * n$

① Deriving with RHD →

E

$E * n$

$E + n * n$

$n + n * n$

Right sentencial form is the reverse of RMD.

$n + n * n$

$E + n * n$

$E * n$

E

While pushing terminals to the stack, handles appear at the top of stack which are reduced with appropriate prod^n.

- A handle of a right sentencial form γ is a prod^n $A \to \beta$ & a position in γ where β may be found & replaced by A to produce the previous right-sentencial form in a RMD of γ. Because γ is a RSF, the substring to the right of a handle contains only terminal symbols.

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

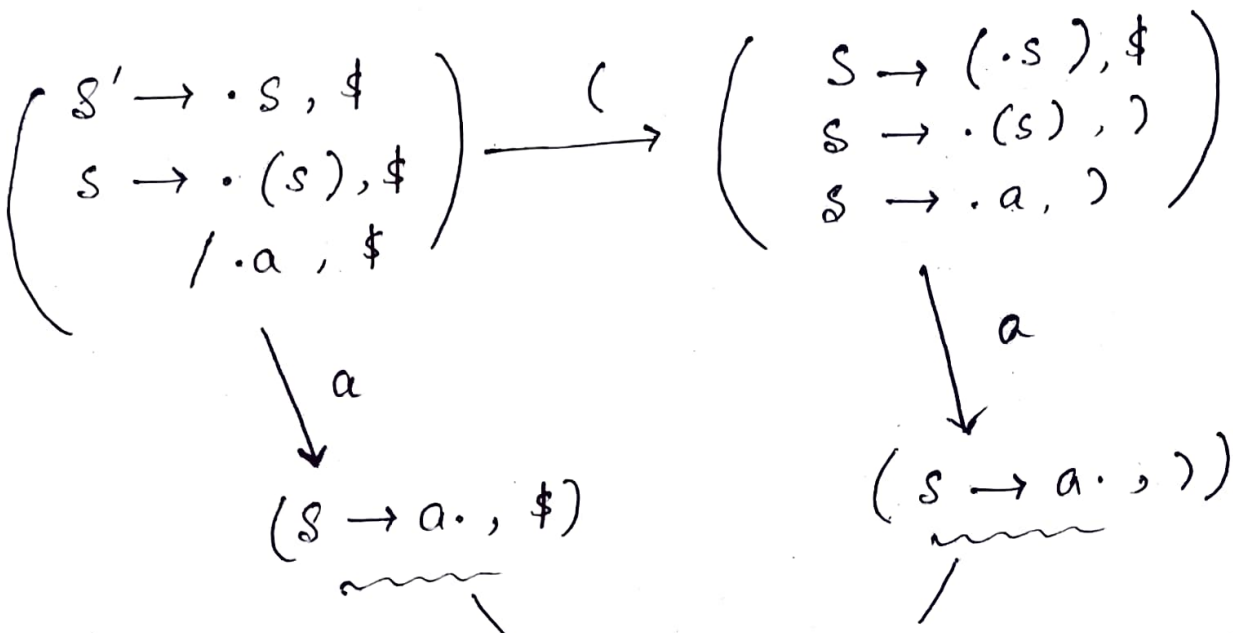
SLR(1)	LR(1) CLR(1)	LALR(1)
n_1	n_2	n_3

a) $n_1 < n_2 < n_3$ c) $n_1 = n_2 = n_3$

→ b) $n_1 = n_3 < n_2$ d) $n_1 \geq n_3 \geq n_2$

→ $n_1 = n_3 \leq n_2$

If same LR(0) items, but different lookaheads, then $n_2 > n_3 = n_1$



at LALR(1) ⇒

states (LALR(1)) < # states (CLR(1))

Syntax Directed Translation

* Parser uses a CFG to validate the input string and produce output for next phase of the compiler. Output could be either a parse tree or abstract syntax tree. Now to interleave semantic analysis with syntax analysis phase of the compiler, we use syntax directed translation.

* Syntax directed translation is augmented rules to the grammar that facilitates semantic analysis. SDT involves passing information bottom-up and/or top-down the parse tree in form of attributes attached to the nodes. SDT rules use 1) lexical value of nodes, 2) constants & 3) attributes associated to the non-terminals in their definitions.

→ In SDT, along with the grammar we associate some informal notations & these notations are called as semantic rules.

✓ Grammar + Semantic Rules = SDT

→ In SDT, every non-terminal can get one or more than one attribute or sometimes 0 attribute depending on the type of the attribute. The value of these attributes is evaluated by the semantic rules associated with the production rule.

In the semantic rule, attribute is VALU and an attribute may hold anything like a string, a number, a memory location & a complex record.

→ In SDT, whenever a construct encounters in the programming language then it is translated according to the semantic rules defined in that particular programming language.

The general approach to SDT is to construct a parse tree or syntax tree & compute the values of attributes at the nodes of the tree by visiting them in some order. In many cases, translation can be done during parsing without building an explicit tree.

eg. $E \rightarrow E+T / T$
$T \rightarrow T * F / F$
$F \rightarrow num$

Grammar to syntactically validate an expression having additions & multiplications in it.

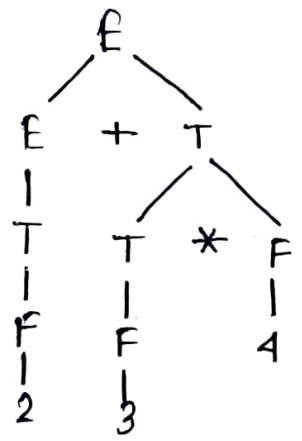
To carry out semantic analysis we will augment SDT rules to this grammar, in order to pass some information up the parse tree & check for semantic errors, if any.

$E \rightarrow E+T$	$\{E.val = E.val + T.val\}$	R1
$E \rightarrow T$	$\{E.val = T.val\}$	R2
$T \rightarrow T*F$	$\{T.val = T.val * F.val\}$	R3
$T \rightarrow F$	$\{T.val = F.val\}$	R4
$F \rightarrow num$	$\{F.val = num.lexval\}$	R5

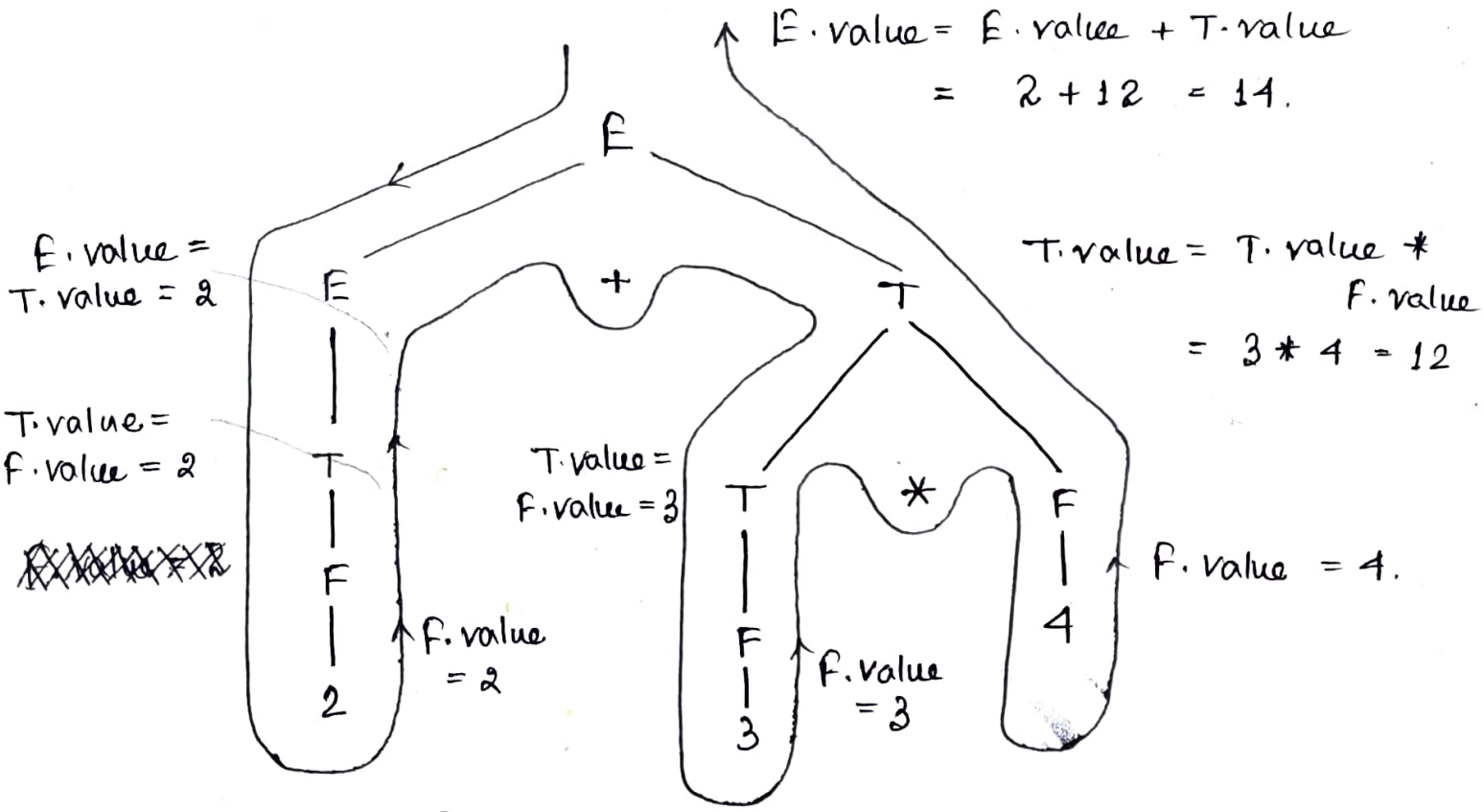
- SDT are augmented rules to a CFG that associate 1) set of attributes to every node of the grammar & 2) set of translation rules to every production rule using attributes, constants & lexical values.

Taking a string to see how semantic analysis happens - $2+3*4$.

Parse tree



We will move bottom up in left to right fashion for computing translation rules.



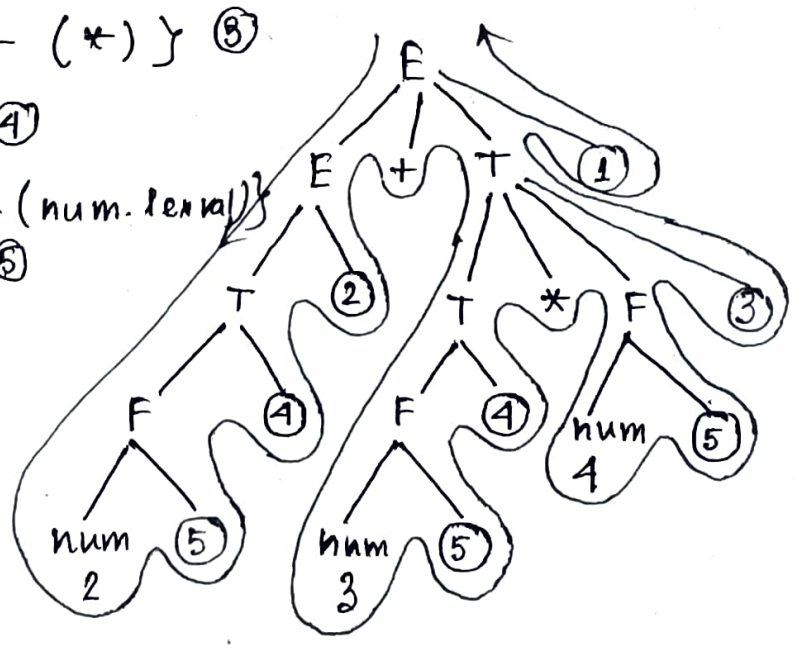
Semantic analysis.

Flow of information happens bottom up & all the children attributes are computed before parents.

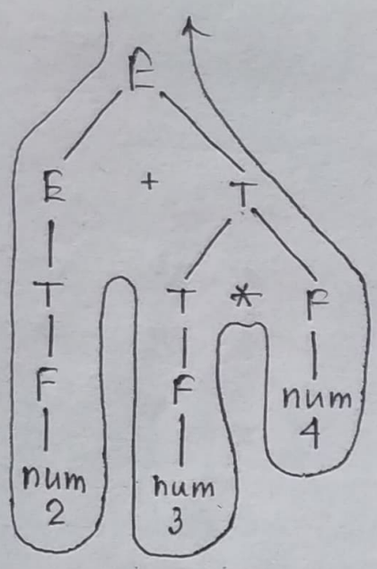
eg
$E \rightarrow E+T$ { print (+) } ①
$/T$ { } ②
$T \rightarrow T*F$ { print (*) } ③
$/F$ { } ④
$F \rightarrow num$ ⑤ { print (num.lexval) } ⑤

$2+3*4$

Postfix expression



$234*+$



2 3 4 * +

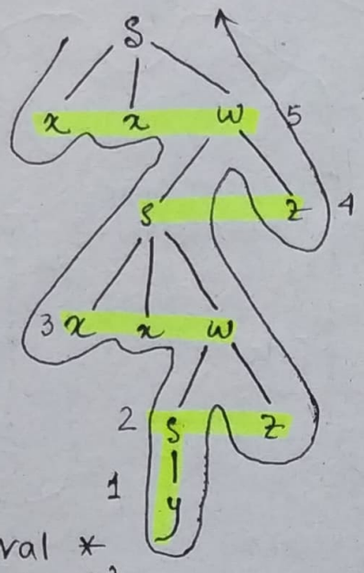
#reductions

eg $S \rightarrow xxw$ {print (1)}

$/y$ {print (2)}

$w \rightarrow sz$ {print (3)}

String x x x y z z.



Output

2 3 1 3 1

eg. $E \rightarrow E * T$ {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}

String, w = 4 - 2 - 4 * 2

E is left recursive ⇒ left associative *

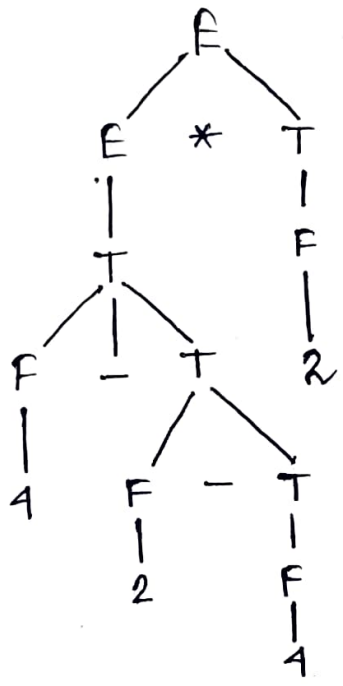
T is right ~~associative~~ ⇒ - right associative

* is defined at a higher level than -

- has higher precedence than *

Just by knowing this ((4-(2-4)) * 2)

= 12



# of reductions

= # non-leaves

= 10

eg

$E \to E \# T$	$\{E.val = E.val * T.val\}$
$/ T$	$\{E.val = T.val\}$
$T \to T \& F$	$\{T.val = T.val + F.val\}$
$/ F$	$\{T.val = F.val\}$
$F \to num$	$\{F.val = num.lexval\}$

2 # 3 & 5 # 6 & 4.

↓

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

↓

((2 * 8) * 10) = 160

→ *

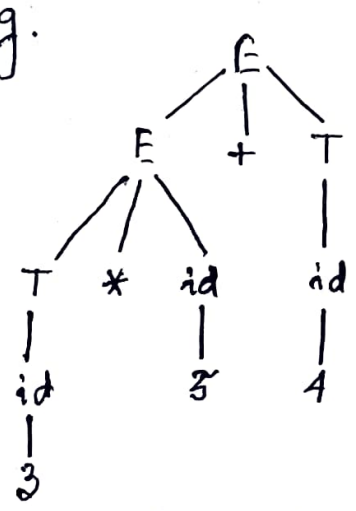
& → +

{ + has higher precedence than *

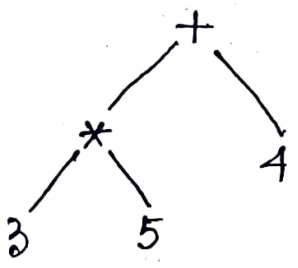
*, + left associative

- Syntax trees. Abstract or compact representation of parse trees.
(Also abstract syntax tree)

eg.



Parse tree



Syntax tree

→ Parse tree

Graphical representation of the replacement process in the derivation.

Interior node represents a grammar rule. Leaf nodes - terminal.

Provides every characteristic information from the real syntax.

Less dense than syntax trees.

Syntax tree.

Compact form of a parse tree.

Interior node represents an operator. Each leaf node represents an operand.

Don't provide every characteristic information from the real syntax.

More dense.

→ Syntax trees are called abstract syntax trees because -

they are abstract representation of the parse trees, they don't provide every characteristic information from the real syntax.

→ Constructing syntax tree

$(a+b) * (c-d) + ((e/f) * (a+b))$.

Step 1: Converting into postfix expression.

Postfix

$ab+cd-*ef/ab+**+$

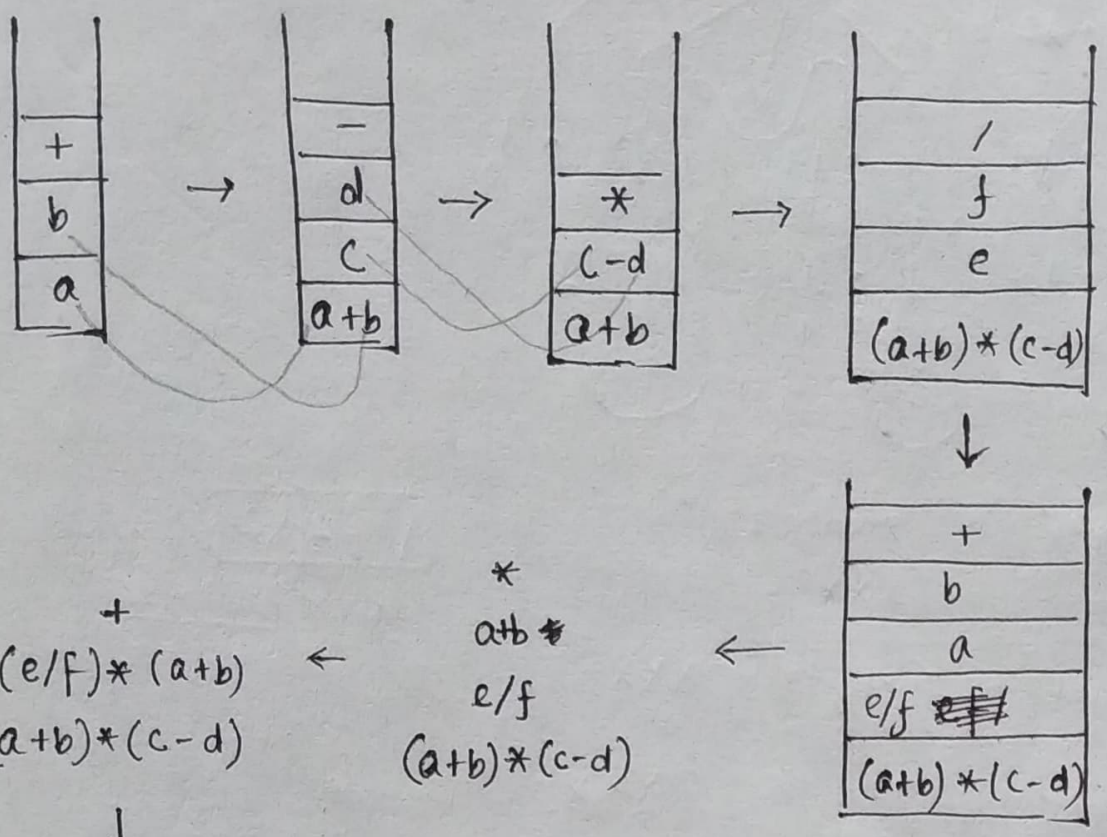
Step 2: Start pushing symbols of the postfix into stack one by one.

Use stack

When an operand is encountered, push into stack. When an operator is encountered, push into stack and pop the operator & the 2 symbols below it. Perform operation on 2

Symbols and build syntax tree accordingly.

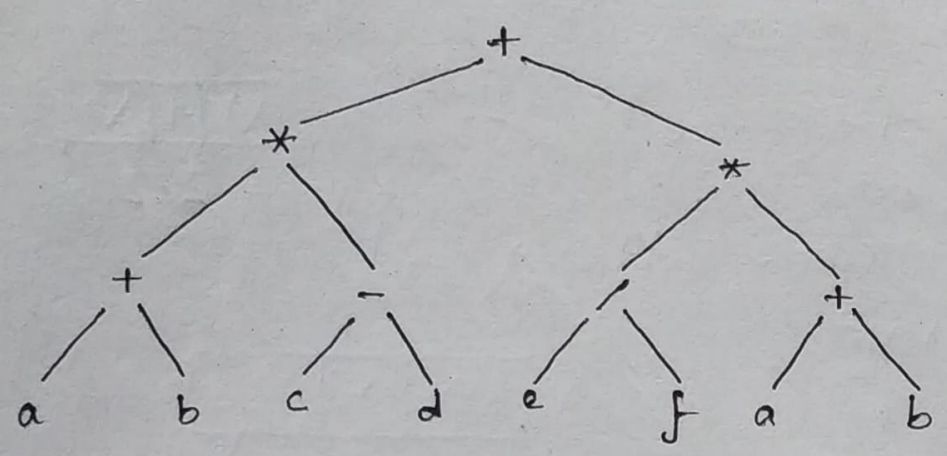
Push result back into stack. (Bottom up tree building)



↓

$(a+b)*(c-d)+(e/f)*(a+b)$

⇒ Postfix $ab+cd-*ef/ab+*+$



eg

$E \to E_1 + T$	$\{E.nptr = mknode(E_1.nptr, '+', T.nptr);\}$	A1
$/T$	$\{E.nptr = T.nptr;\}$	A2
$T \to T_1 * F$	$\{T.nptr = mknode(T_1.nptr, *, F.nptr);\}$	A3
$/F$	$\{T.nptr = F.nptr;\}$	A4
$f \to id$	$\{f.nptr = mknode(null, id\ name, null);\}$	A5

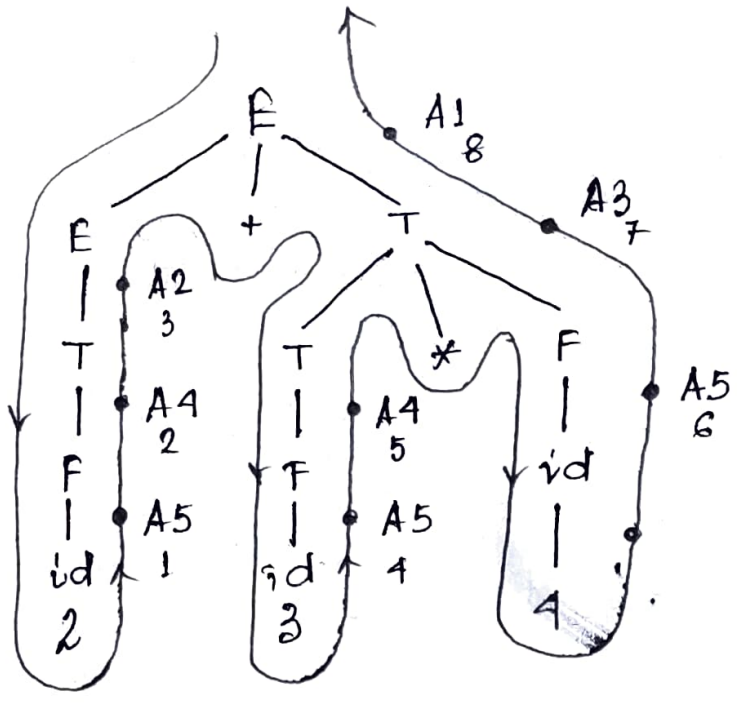
Building Syntax Tree

↑ SDT to build abstract syntax tree

Evaluate $2+3*4$.

E, E_1 ; T, T_1 used just to distinguish between left and right symbols.

2+3*4



1) f.nptr = Address (2)
= XXX

10	2	10

XXX address

2) T.nptr = f.nptr = XXX

3) E.nptr = T.nptr = XXX

4) f.nptr = Address (3) = YYY

10	3	10

YYY.

5) T.nptr = f.nptr = YYY.

6) f.nptr = Address (4) = ZZZ

10	4	10

ZZZ

7) T.nptr = mknode (YYY, *, ZZZ)
= MNN

YYY	*	ZZZ

MNN

10 3 10 10 4 10

8) E.nptr = mknode (XXX, +, MNN)
= NNN

XXX	+	MNN

10 2 10 YYY * ZZZ

10 3 10 10 4 10

eg. $E \rightarrow E_1 + E_2$ { if ((E_1.type == E_2.type) && (E_1.type = int) then E_1.type = int else error } A1

/ $E_1 == E_2$ { if ((E_1.type == E_2.type) && (E_1.type = int/ boolean)) then E.type = boolean else error } A2

Type checking

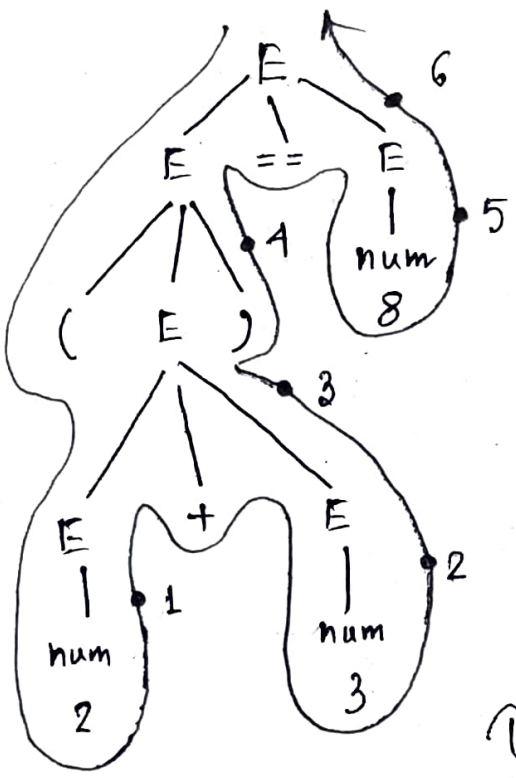
/ (E_1) { E.type = E_1.type } A3

/ num { E.type = int } A4

/ True { E.type = bool } A5

/ False { E.type = bool } A6

Expression - (2+3) == 8.



1) E.type = int A4

2) E.type = int A4

3) E.type = int A1

4) E.type = int A3

5) E.type = int A4

6) E.type = ~~bool~~ boolean A2

Type of expression is boolean.

eg.

N → L Binary number generation.

L → LB / B L - list of bits

B → 0 / 1 B - Bit

N - Number.

Semantic rules. for counting # 1's

N → L { N.count = L.count }

~~L~~ → LB { L.count = L.count + B.count }

/B { L.count = B.count }

B → 0 { B.count = 0 }

/1 { B.count = 1 }.

→ Semantic rules (Counting 0's)

All same except

$B \to 0$	$\{B.count = 1\}$
$/1$	$\{B.count = 0\}$

→ Semantic rules (counting bits)

All same except

$B \to 0$	$\{B.count = 1\}$
$/1$	$\{B.count = 1\}$

→ Binary to decimal.

dval – decimal value.

$N \to L$	$\{N.dval = L.dval\}$
$L \to LB$	$\{L.dval = L.dval * 2 + B.dval\}$ (base (2))
$/B$	$\{L.dval = B.dval\}$
$B \to 0$	$\{B.dval = 0\}$
$/1$	$\{B.dval = 1\}$.

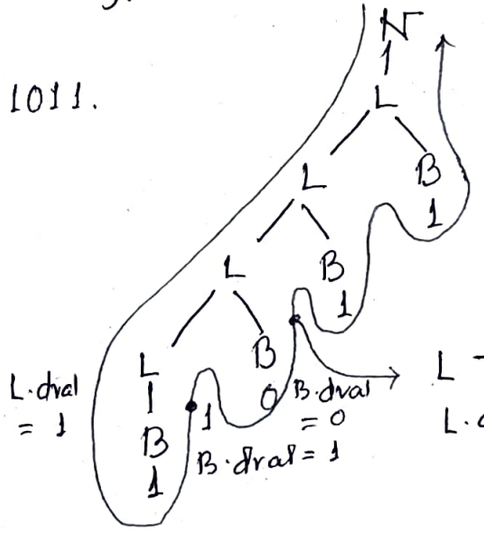
1101

1 x 2 + 1 = 3

3 x 2 + 0 = 6

6 x 2 + 1 = 13.

Ilus. $\omega = 1011$.



eg.

→ Dotted binary.

11.01.

$\hookrightarrow \frac{1}{2^2} = 0.25$

11

$\hookrightarrow \frac{3}{4} = 0.75$.

$N \to L_1.L_2$ $\{N.dval = L_1.dval + (L_2.dval)/2^{L_2.c}\}$

$L \to L_1B$ $\{L.count = L_1.count + B.count,\ L.dval = L_1.dval + B.dval\}$

$/B$ $\{L.count = B.count,\ L.dval = B.dval\}$

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

$/1$ $\{B.count = 1,\ B.dval = 1\}$

count = c

eg SDT to generate 3-address code.

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

$E \rightarrow E_1 + T$ { E.place = newTemp( ); gen(E.place = E_1.place + T.place) }

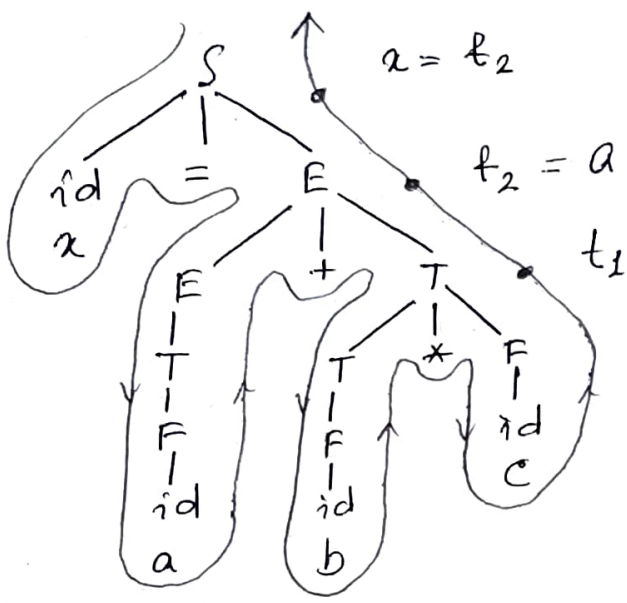
/ T { E.place = T.place }

$T \rightarrow T_1 * F$ { T.place = newTemp( ); gen (T.place = T_1.place * F.place); }

/ F { T.place = F.place }

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

$x = a + b * c$.



* Kinds of attributes.

a) Synthesized : They are computed from the values of the attributes of the children nodes.

b) Inherited : They are computed from the values of the the attributes of both the siblings & the parent nodes.

✓ eg. $A \rightarrow BC$ is a production.

A's attribute is dependent on B's or C's attributes. Then it is synthesized.

B's attribute is dependent on A's or C's attributes. Then it is inherited.

Synthesized

E.val ⇒ F.val.

Inherited.

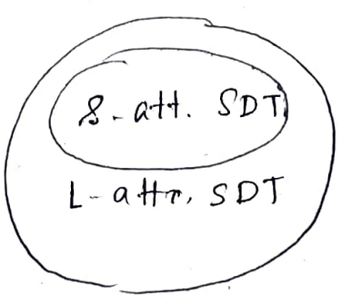
E.val = F.val

* S-attributed SDT and L-attributed SDT

S-Attributed.	L-attributed.
1. Uses only synthesized attributes. (that's why S-)	1. Uses both synthesized and inherited attributes with a restriction that inherited attribute can inherit values from left sibling only. eg.
$A \to XYZ \{Y.S = A.S,\ Y.S = X.S,\ Y.S = Z.S\}$ — Not L-attributed (inheritance from right sibling)	
2. Semantic actions are placed in rightmost place of RHS. $A \to BC\{\}$	2. Semantic actions are placed anywhere in RHS. $A \to \{\}BC / D\{\}E$
3. Evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.	3. Attributes are evaluated by depth-first & left-to-right parsing manner.

→ If a definition is S-attributed, then it is also L-attributed but not vice versa.

(as there's no condition checking for inherited attributes)



eg $A \rightarrow LM \{ L.i = f(A.i) ; M.i = f(L.s) ; A.s = f(M.s); \}$

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

→ Inherited attrib. Not S-att.

→ Not L. attributed. inheritance from right sibling

a) S. attrib.

b) L- attrib.

c) Both

d) None

eg. $A \rightarrow BC \{ B.s = A.s \}$ Inherited

Not S-attributed

a) S. att. b) L. att. c) Both d) None

eg. P1 : $S \rightarrow MN \{ S.val = M.val + N.val \}$

P2 : $M \rightarrow PQ \{ M.val = P.val * Q.val$ and $P.val = Q.val \}$.

a) P_1 and P_2 are S-att.

b) P_1 S-att. , P_2 L-att.

c) P_1 L-att. , P_2 – S-att.

d) None of the above

→ In P1, S is synthesized attribute and in L-attributed SDT synthesized att. is allowed.

In P2, P is depending on Q, that's on RHS.

P_2 not L-attributed.

* SDT to add type information into symbol table.

$D \rightarrow TL \quad \{L.in = T.type\}$ ⇒ Inherited attribute

$T \rightarrow int \quad \{T.type = int\}$ ⇒ Synthesized attribute

/char $\{T.type = char\}$

} L-attrib.

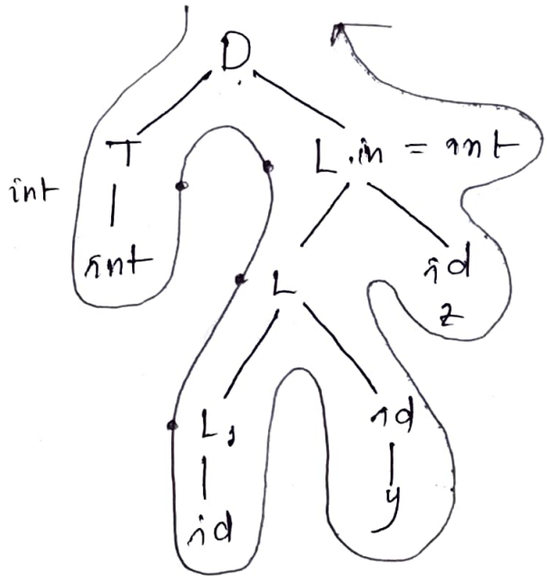
int x, y, z;

x	int
y	int
z	int

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

L → inherited

/id $\{addtype(id.name, L.in)\}$



→ Evaluate the synthesized attribute when you last visit it.

→ Evaluate the inherited attribute when you first visit it.

S-attributed SDT

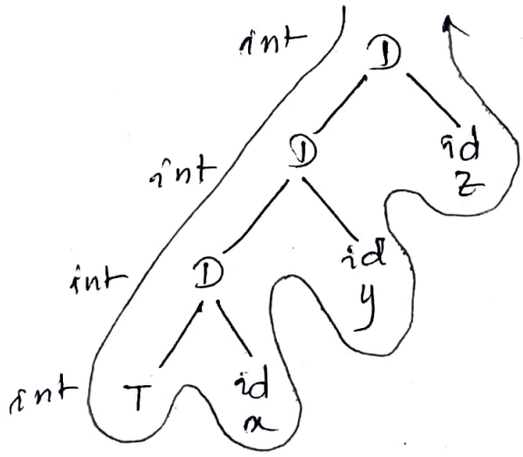
$D \rightarrow D_1, id \mid \{addtype(id.name, D_1.type)\}$

/T _ id $\mid \{addtype(id.name, T.type), D.type = T.type\}$

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

/char $\{T.type = char\}$

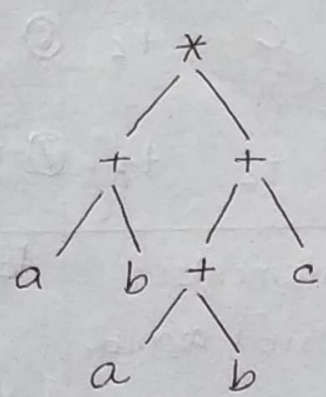
x	int
y	int
z	int



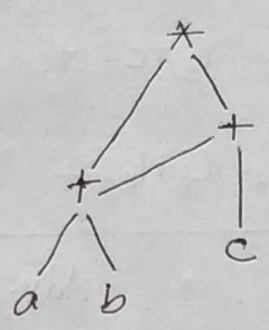
Intermediate Code Generation

Intermediate code. eg. $(a+b)*(a+b+c)$

- Linear form
  - Postfix: $ab+ab+c+*$
  - 3-address code:
    $t_1 = a+b$
    $t_2 = a+b$
    $t_3 = t_2+c$
    $t_4 = t_1 * t_3$
- Tree form
  - Syntax tree



  - DAG



* Types of 3 address code.

1. $x = y$ op z — $x = a+b$ binary operation
2. $x =$ op z — unary operation.
3. $x = y$ — assignment
4. if x (rel op) y goto L
5. goto L — unconditional
6. $A[i] = x$ — Array indexing
   $y = A[i]$
7. $x = *p$ — pointer
   $y = \&z$ — address of variable.

* Various representations of 3-address code:

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

	Quadruple				Triple			Indirect Triple
	opr	op1	op2	res	opr	op1	op2	pointer to instructi
1. $t_1 = a+b$	+	a	b	t_1	① +	a	b	i) (1)
2. $t_2 = -t_1$	−	t_1	NULL	t_2	② −	(1)		ii) (2)
3. $t_3 = c+d$	+	c	d	t_3	③ +	c	d	iii) (3)
4. $t_4 = t_2 * t_3$	*	t_2	t_3	t_4	④ *	(2)	(3)	iv) (4)
5. $t_5 = a+b$	+	a	b	t_5	⑤ +	a	b	v) (5)
6. $t_6 = t_5 + c$	+	t_5	c	t_6	⑥ +	(5)	c	vi) (6)
7. $t_7 = t_4 + t_6$	+	t_4	t_6	t_7	⑦ +	(4)	(6)	vii) (7)
	Adv.: Statements can be moved around. Dadv: Space wasted.				Adv.: Space not wasted. D.Adv.: Statements can't be moved around.			Adv.: Statements can be moved around. D.Adv.: 2 accesse to memory.

* Backpatching & conversions to 3-address code:

Backpatching is leaving the labels as empty & filling them later.

- if $(a<b)$ then $t=1$ else $t=0$.

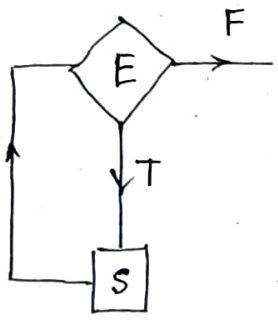
i) if $(a<b)$ goto $i+3$
$i+1$) $t=0$
$i+2$) goto $i+4$
$i+3$) $t=1$
$i+4$)

t_1 t_2 t_3

- $a < b$ and $c < d$ or $e < f$

100) if $(a<b)$ goto 103
101) $t_1 = 0$
102) goto 104
103) $t_1 = 1$
104) if $(c<d)$ goto 107
105) $t_2 = 0$
106) goto 108
107) $t_2 = 1$
108) if $(e<f)$ goto 111
109) $t_3 = 0$
110) goto 112
111) $t_3 = 1$
112) $t_4 = [t_1$ and $t_2]$
113) $t_5 = [t_4$ or $t_3]$

- While E do S.



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

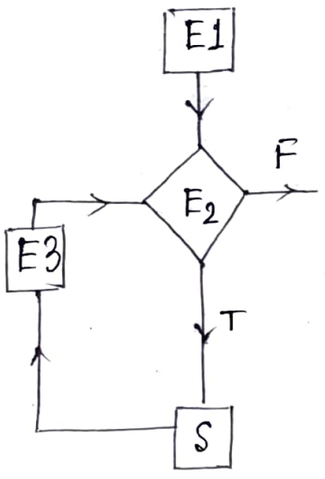
OR

L: if (E) goto L1
goto Last
L1: S
goto L
Last:

- While $(a<b)$ do
$x = y+z$.

L: if $(a<b)$ 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 L1
    goto Last
L1: t1 = b + c
    a = t1
    t = i + 1
    i = t
    goto L
Last:
```

- Switch (i + j)
  {
  case (1) :
    a = b + c;
    break;
  case (2) :
    p = q + r;
    break ;
  default :
    x = y + z;
    break;
  }

```
    t = 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
test: if (t == 1) goto L1
      if (t == 2) goto L2
      goto L3
Last:
```

* 2D array to 3 address code.

eg. A[4][4]

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

A[2][3]

row major order :

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

We need to cross $2 \times \left(\begin{matrix}\text{no. of}\\ \text{col.}\end{matrix}\right) + 3$

$= 2 \times 4 + 3$

$= 11$ elems to reach A[2][3]

→ $x = A[y][z]$

$t_1 = y * \left(\begin{matrix}\text{no. of}\\ \text{columns}\end{matrix}\right)$

$t_2 = t_1 + z$

$t_3 = t_2 * \left(\begin{matrix}\text{size of}\\ \text{each word}\\ \text{in Bytes}\end{matrix}\right)$

$t_4 =$ Base address of A

$x = t_4[t_3]$ ~ Base offset addressing

(Base address + offset)

$t_4 + t_3$

G'07 OP R_j, R_i Perform R_j OP R_i and store in R_j

OP m, R_i (m) OP $R_i \rightarrow R_i$ (m) - value in mem loc^n m

MOV m, R_i (m) $\rightarrow R_i$

MOV R_i, m $R_i \rightarrow$ (m).

Computer has only 2 registers of OP is either ADD or SUB. Consider following :

$t_1 = a + b$ $t_3 = e - t_2$

$t_2 = c + d$ $t_4 = t_1 - t_3$

Assuming all operands are initially in memory, and the final value of computation should be in memory. What is the min. no. of MOV inst^{n}s in the code generated for this basic block ?

→ During ADD or SUB, we already access the memory to ~~fetch~~ fetch operands. So, we use these operators to fetch one operand and another operand is fetched by MOV.

```
MOV a, R1        SUB e, R2
ADD b, R1        SUB R1, R2      => (3) Ans
MOV c, R2        MOV R2, m
ADD d, R2
```

Q. G'04. Consider grammar rule $E \rightarrow E_1 - E_2$ for arithmetic expressions. The code generated is targeted to a CPU having a single user register. Subtraction requires the first operand to be in the register. If E_1 & E_2 do not have any common subexpression, in order to get the shortest possible code —

a) E_1 should be evaluated first.

✓ b) E_2 should be n n.

c) Evaluation of E_1 & E_2 should necessarily

be interleaved.

d) Order of evaluation of E_1 & E_2 is of no consequence.

→ E_1 evaluated first

R: E_1

R: E_1 → m

R: E_2 E_1 m

R: E_1 E_2 m.

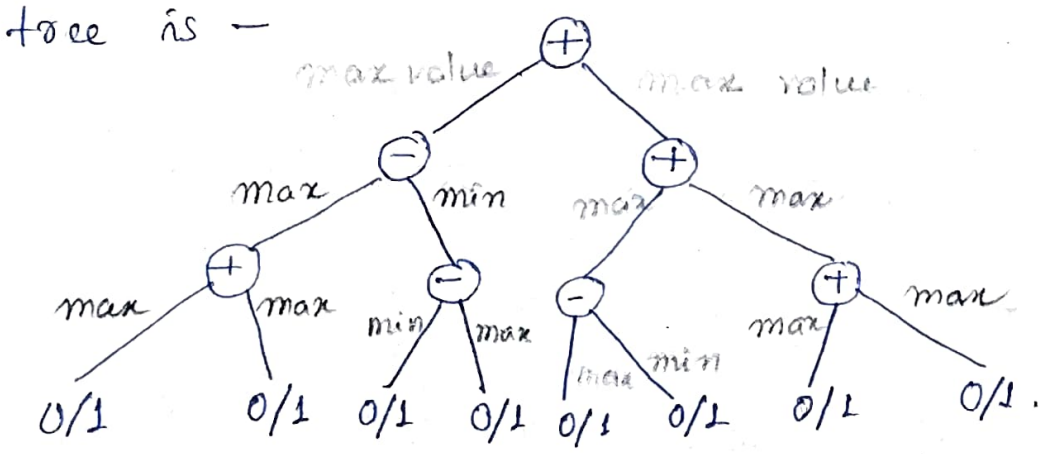
E_2 evaluated first

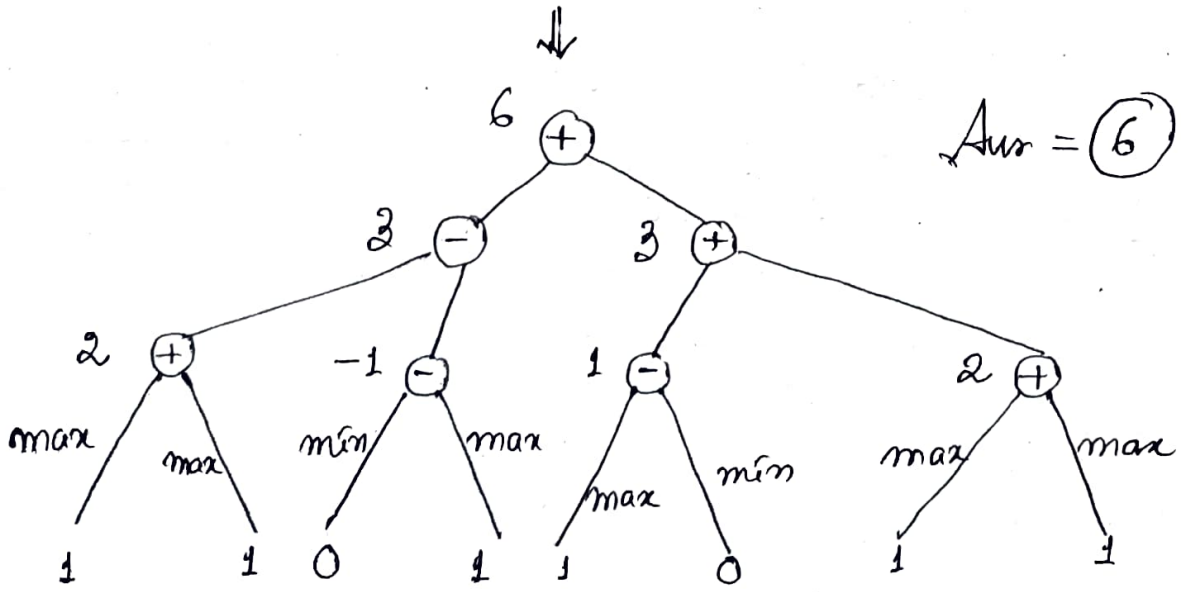
R: E_2

R: E_2 → m

R: E_1 E_2 m

Q. G'14. Each leaf represents a numerical value (0/1) in the expression tree. Max. possible value of the expression represented by the tree is —





Ans = (6)

Runtime Environment.

* Storage Allocation Strategies.

1. static.

- Allocation is done at ~~runtime~~ compile time.
- Bindings do not change at runtime.
- One activation record per procedure.

disadv.

- Recursion is not supported.
- Size of data objects must be known at compile time.
- Data structures cannot be created dynamically.

2. Stack

- Whenever a new activation begins, activation record is pushed on to the stack & whenever activation ends, activation record is popped off.

disadv: Local variables cannot be retained once activation ends.

- Local variables are bound to fresh storage.

3. Heaps

- Allocation & deallocation can be done in any order.

disadv. Heap management is overhead.

(OS - Memory mgmt → create holes in memory.).

Summary

Activations can have

i) permanent lifetime in case of static allocation.

ii) nested life time in case of stack allocation.

iii) Arbitrary life-time in case of heap allocation.

Code optimisation

* Optimisation.

Machine independent

1. Loop optimisations
   a) Code motion or frequency ~~reduction~~ reduction
   b) Loop unrolling
   c) Loop jamming
2. Folding
   - Constant propagation
3. Redundancy elimination
4. Strength reduction
5. Common subexpression elimination

Machine dependent.

1. Register allocation
2. Use of addressing modes
3. Peephole optimisation
   a) Redundant load/store
   b) Flow of control optimisation.
   c) Strength reduction
   d) Use of machine idioms.

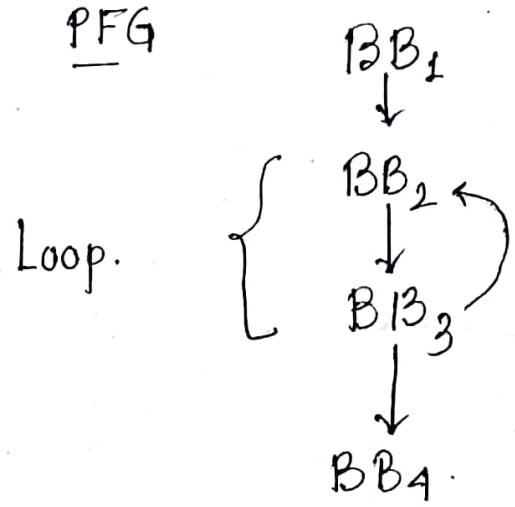
* Loop optimisations.

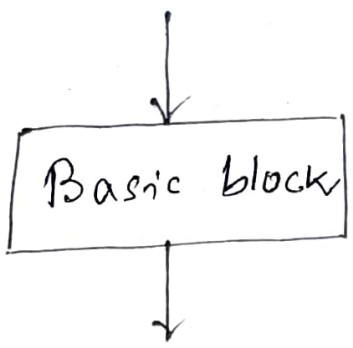
- To apply loop optimisation, we must first detect loops.
- For detecting loops, we use control flow analysis (CFA) using program flow graph (PFG).
- To find PFG, we need to find basic blocks.

A basic block is a sequence of

3-address statements where control enters at the beginning & leaves only at the end without any jumps or halts.

PFG





→ Finding the basic blocks.

In order to find the basic blocks, we need to find the leaders in the program. Then a basic block will start from one leader to the next leader excluding next leader.

Identifying leaders.

- 1. 1 statement is a leader.
- 2. Statement that is target of conditional or unconditional statement is a leader.
- 3. Statement that follows immediately a conditional or unconditional statement is a leader.

eg. fact(x) {

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

3 address code —

B_1
1. f = 1
2. i = 2

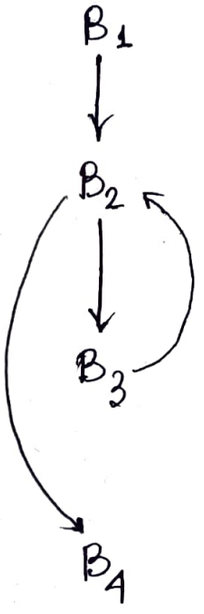
B_2
3. if (i > x) goto 9

B_3
4. t_1 = f * i
5. f = t_1
6. t_2 = i + 1
7. i = t_2
8. goto (3)

B_4
9. goto calling program.

Leaders are 1, 3, 4, 9
acc. to rules 1, 2, 3, 2

PFG.



Loop. detected.

* Loop optimisations.

i) Frequency reduction. : Moving the code from high frequency region to low frequency region.

eg. while (i < 5000) {
    A = (sinx / cosx) * i;
    i++;
}

⟹

t = sinx/cosx;
while (i < 5000)
    A = t * i;

ii) Loop unrolling

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

⇓

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

iii) Loop jamming / fusion

combines 2 loops.

```
for (i = 0 ; i < 10 ; i++) {
    for (j = 0 ; j < 10 ; j++)
        a[i][j] = 0 ; }
```

```
for (i = 0 ; i < 10 ; i++)
    a[i][i] = 0.
```

⇓

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

* Machine independent optimisation.

- Folding : Replacing an expression that can be computed at compile time by its values.

eg. $2+3+b+c \Rightarrow 5+b+c$

- Redundancy elimination. (DAG)

eg. $A = B + C$

$D = 2 + B + 3 + C$

$D = 2 + 3 + A.$

- Strength reduction.

Replacing a costly operation by a cheaper one.

eg. $B = A * 2 \Rightarrow B = A \ll 1$

- Algebraic simplification

$A = A + 0$ } Eliminate

$x = x * 1$ }

* Machine dependent optimisations

1. Register allocation
   - Local allocation
   - Global allocation.

   (COA)

2. Use of addressing modes. (COA)

3. Peephole optimisation.

   a) Redundant Load & Store elimination

$x = y + z$

```
MOV y, R0
ADD z, R0
MOV R0, x
```

```
a = b+c        MOV b1, R0
d = a+e        ADD c, R0
               MOV R0, a
               MOV a, R0
               ADD e, R0
               MOV R0, d
```

} x Redundant

   b) Flow control optimisation.

Avoid jumps on jumps

```
L1 : Jump L2 ← L4
L2 : Jump L3
L3 : Jump L4
```

Eliminate dead code

```
#define x 0
if (x) {
  ---
}
```

} dead code

c) Use of machine idioms.

$i = i + 1$ |

```
MOV R0, i
ADD R0, 1
MOV i, R0
```

} **inc i** if available in the instruction set.

- Common subexp^n elimination

```
a = b*c + g          t = b*c
                ⇒    a = t + g
d = b*c * K          d = t * K
```

- Data flow analysis. (Global optimiz^n)
  - Available exp^n
  - Reaching def^n

| Def^n pt.
| Reference pt.
| Eval^n pt.

- CO
  - M/C dependent
  - Machine independent
    - Loop invariant
    - Loop jamming
    - Loop unrolling
    - Induction variable elimination
    - Strength red^n
    - Algebraic simplification
    - Constant folding
    - Copy propagation
    - Deadcode elim^n
    - Common subexp^n elim^n
    - Constant prop^n

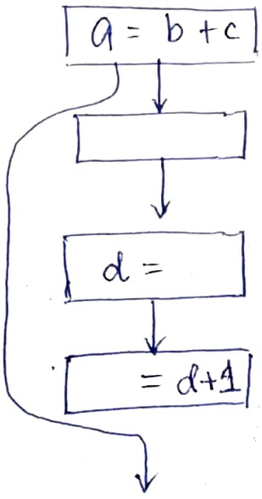
Liveness Analysis.

> A variable v is live at a statement n if there exists a path in the CFG (control flow graph) from this statement to a statement m such that v is used in m & for each $n \le k < m$, $v \notin def[k]$.

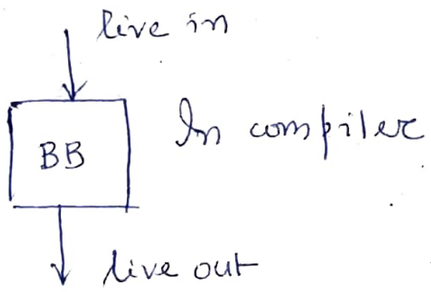
or, A variable is live at a statement if it is used at that statement.

> When there's def^n of a variable, that var. is dead at that statement.

$a = b + c$ a dead here

> 

b, c, d



In compiler

Eg:

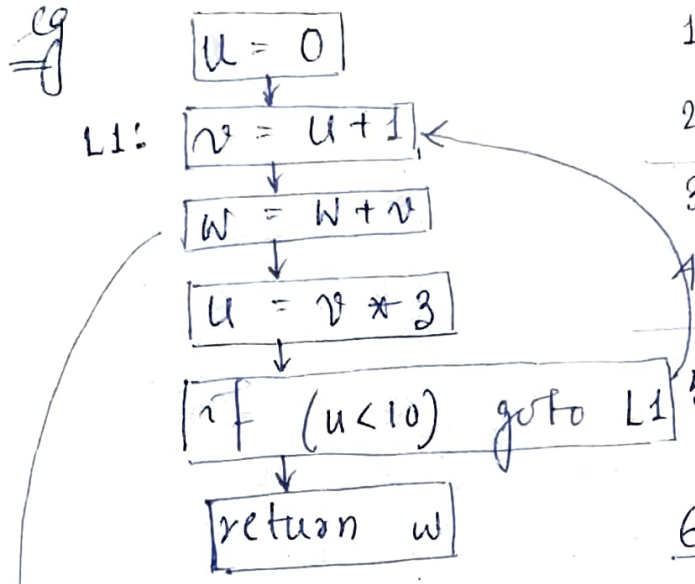
1	$v = 1$	x	y	z	u	v	w
2	$z = v + 1$	dead	dead	dead	dead	dead	dead
3	$x = 2 * v$	dead	dead	dead	dead	live	dead
4	$y = x * 2$	d	d	l	d	l	d
5	$w = x + z * y$	l	d	l	d	d	d
6	$u = z + 2$	l	l	l	d	d	d
7	$v = u + w + y$	d	l	l	d	d	l
8	return $v * u$	d	l	d	l	d	l
		d	d	d	l	l	d

life (x) = 4, 5 life (y) = 5, 6, 7

variables live at 4 = x, z

variables live at both 3, 4 = z

eg



	u	v	w
1	d	d	l ← **
2	l	d	l
3	d	l	l
4	d	l	l
5	l	d	l
6	d	d	l

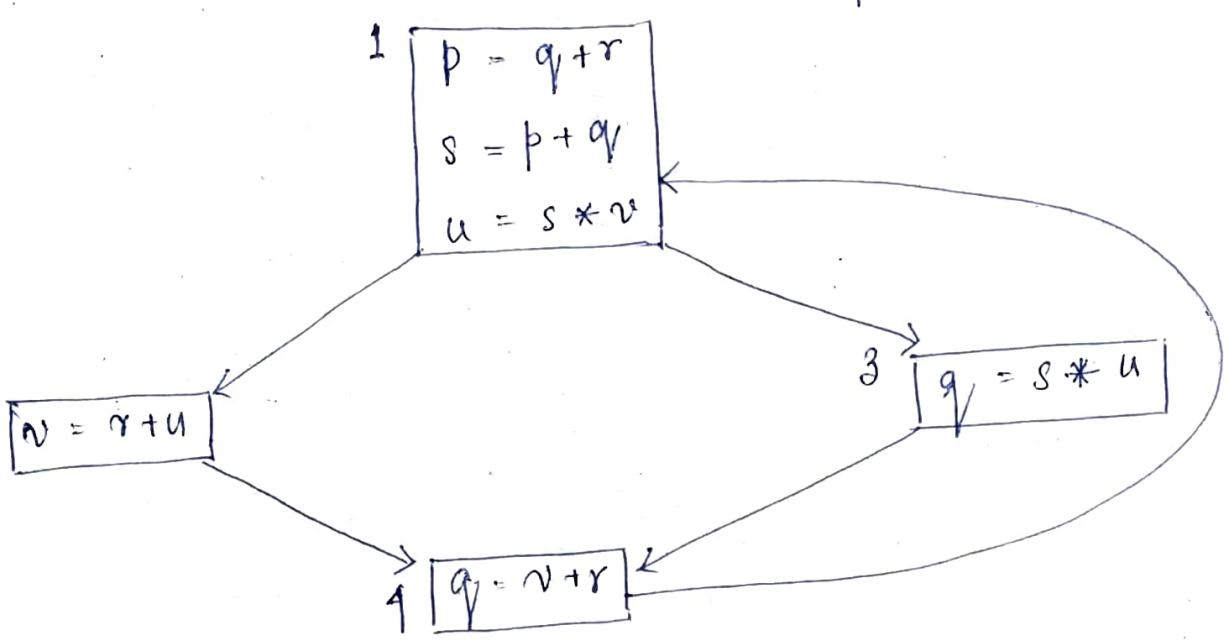
it's same as

$t = w + v$

$w_1 = t$

eg. G.

p, q, r, s, u, v



	p	q	r	s	u	v
(2)	d	d	l	d	l	d
(3)	d	d	l	l	l	l

live both in 2, 3 ⇒ r, u

> IN[B] = Set of variables live at beginning of Block B

OUT[B] = " " " just after B

USE/GEN[B] = Variables that are used in B. before any assignment

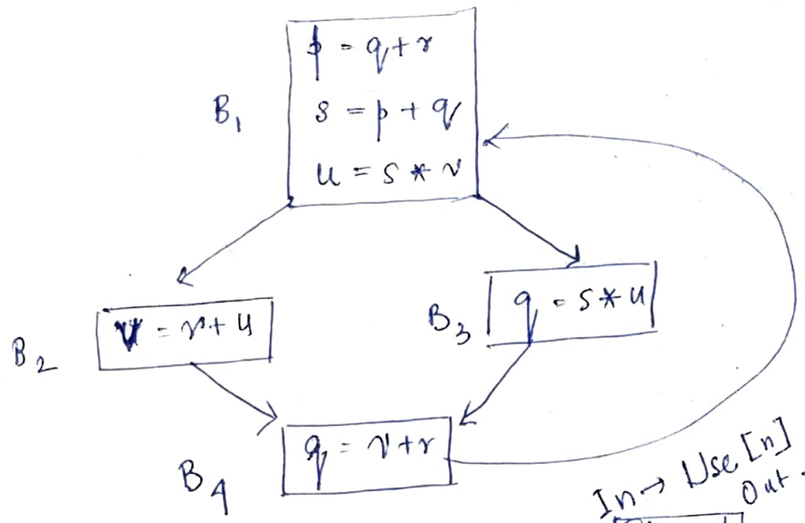
(Var.s in RHS but not in LHS of prior statement in B)

DEF/KILL[B] = Var.s that are assigned a value in B (Var. in LHS)

$$IN[n] = USE[n] \cup (OUT[n] - DEF[n])$$

$$OUT[n] = \bigcup_{s \in succ[n]} IN[s]$$

p, q, r, s, u, v



In → Use[n]
Out → ⋃ In[s]
succ[n]

Step 1

In → Use ∪ (Out − DEF)
Out → ⋃ In

Step 2

node/ Block	USE	DEF	IN	OUT	IN	OUT
1	q, r, v	p, s, u	q, r, v	r, u, s	q, r, v,	s, u, v, r
2	r, u	v	r, u	v, r	r, u	v, r
3	s, u	q	s, u	v, r	s, u, v, r	v, r
4	v, r	q	v, r	q, r, v	v, r	q, r, v

live (B_2) = r, u

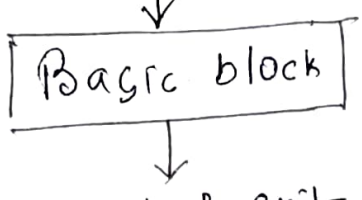
live (B_3) = s, u, v, r

Control flow graph.

> Finding leader :

1) First statement always leader
2) Addr. of cond^{l}, uncond^{l} goto are leaders
3) Next line of cond^{l}, uncond^{l} goto are leaders.

> First line of basic block is leader.



Eg. Intermediate code.

Find # nodes, # edges in CFG.

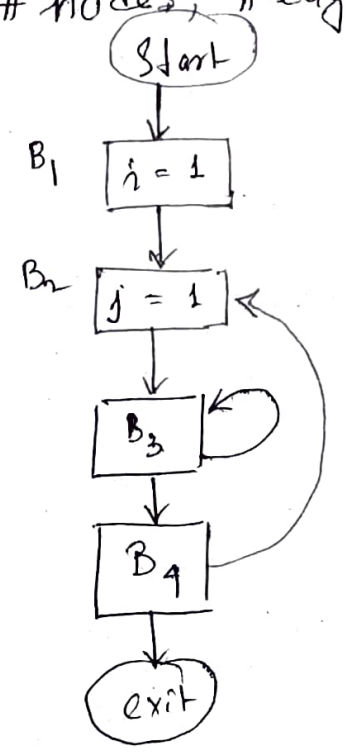
Leaders - L

1 L 1 $i = 1$

2 L 2 $j = 1$

2 L
3	$t_1 = 5 * i$	
4	$t_2 = t_1 + j$	
5	$t_3 = 4 * t_2$	
6	$t_4 = t_3$	B_3
7	$a[t_4] = -$	
8	$j = j + 1$	
9	if $j \leq 5$	goto 3

3 L
10	$i = i + 1$	B_4
11	if $i < 5$	goto 2



n = 4 + 2

e = 5

• Static single assignment (ICG).

```
x = u - t ;
y = x * v ;
x = y + w ;
y = t - z ;
y = x * y ;
```

min # total vars. needed to convert this 3A code into SSA form is — 10

```
x1 = u - t
y1 = x1 * v
x2 = y1 + w
y2 = t - z
y3 = x2 * y2
```

⇒ x_1, x_2, y_1, y_2, y_3, u, t, v, w, z

(Write exp" → Best 3AC → Best SSA)

eg. $t_1 = a * a$
$t_2 = a * b$
$t_3 = t_1 * t_2$
$t_4 = t_3 + t_2$.

Min. no. of vars in eqv. 3AC. or (eqv. SSA).

$t_4 = t_3 + t_2$
$= (t_1 * t_2) + (a * b)$
$= ((a * a) * (a * b)) + (a * b)$.

(min 2 var in 3AC.)

```
b = a * b
a = a * a
a = a * b
a = a + b
```

eqv. 3AC

eqv. SSA

```
b1 = a * b
a1 = a * a
a2 = a1 * b1
a3 = a2 * b1
```

5 vars min in SSA

---

eg $t_1 = b * c$
$t_2 = a + t_1$
$t_3 = b * c$
$t_4 = d * t_3$
$t_5 = t_2 + t_4$

$t_5 = (a + t_1) + (d * t_3)$
$= (a + (b * c)) + (d * (b * c))$

(b * c): b, a — (b * c): b

```
b = b * c
a = a + b
b = d * b
a = a + b
```

eqv. 3AC. 4 vars.

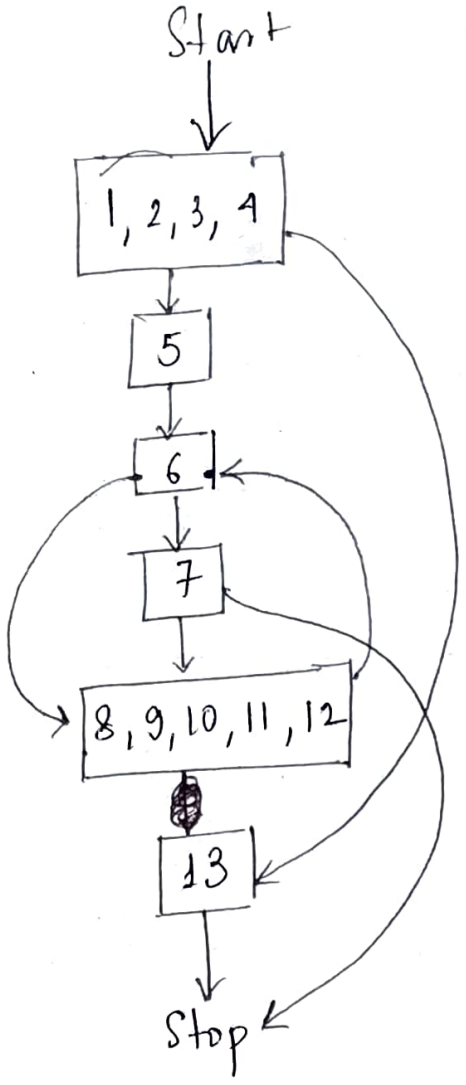
SSA

```
b1 = b * c
a1 = a + b1
b2 = d * b1
a2 = a1 + b2
```

8 for SSA ($a_1, b_1, a_2, b, c, a,$ b_2)

eg CFG

```
→   receive m (val)          1
    f0 ← 0                   2
    f1 ← 1                   3
    if m <= 1 goto L3        4
→   i ← 2                    5
→ L1: if i <= m goto L2      6
→   return f2                7
→ L2: f2 ← f0 + f1           8
    f0 ← f1                  9
    f1 ← f2                  10
    i ← i + 1                11
    goto L1                  12
→ L3 : return m              13
```

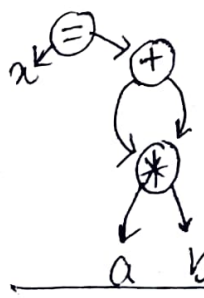


6 BBs, 8 nodes,
7 controls, 10 edges

Intermediate code repⁿ

- Linear : Postfix, 3AC, SSA
- Non-linear : Syntax tree, DAG.

→ DAG used to eliminate common subexpⁿ elimination.



$x = (a*b) + (a * b)$

eg. 3-address code → Syntax tree → DAG

```
1   t1 = b * c
2   t2 = a + t1
3   t3 = b * c
4   t4 = d / t3
5   t5 = t2 - t4 .
```

* Optimisation

- Constant folding $3*2=6$
- Copy propagation
  - Constant prop^n
  - Variable ^{n}
- Algebraic simplification

$y = x+a$
$z = y-a$
$w = z*b$
$\rightarrow$ $w = (y-a)*b = ((x+a)-a)*b = x*b$
$\Downarrow$
$w = a*b$

- Strength reduction
- Dead code elimination
- Common subexp^n elimination.
- Loop optimization
  - Code motion
  - Induction var. elimination
  - Loop merge
  - Loop unroll

* Data flow analysis.

- Available exp^n analysis (Common subexp^n elim^n) — forward
- Liveness analysis (Dead code elim^n) — backward
- Reaching def^n's (Copy propagation) — forward

* Live variable

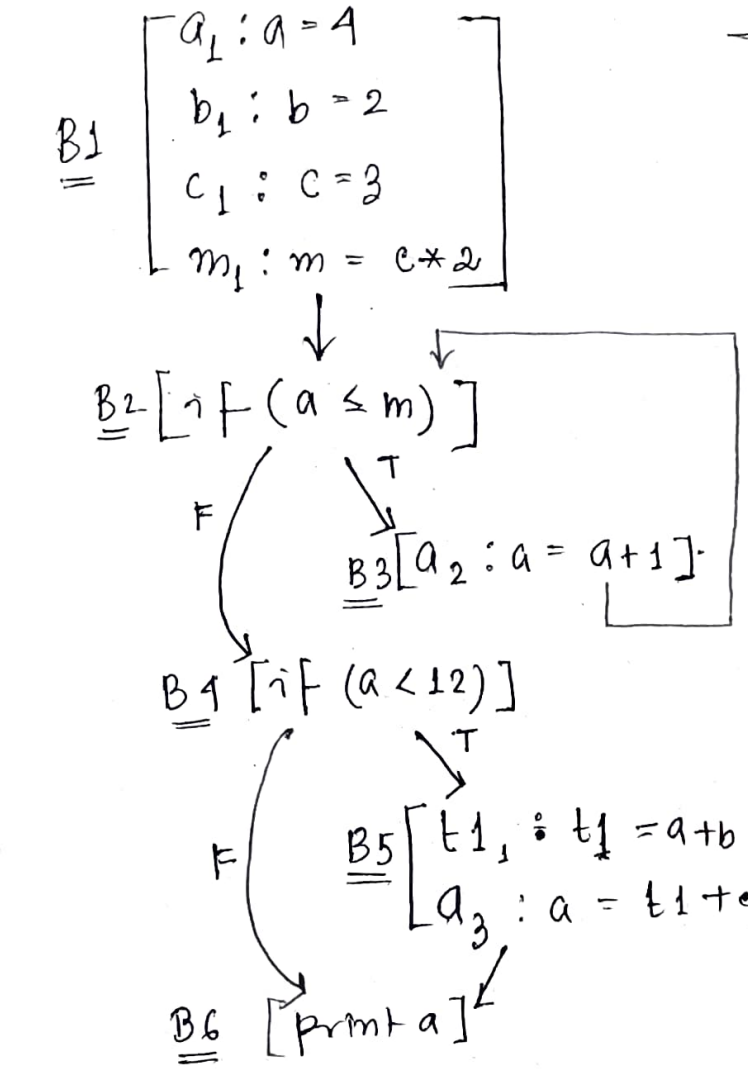
x live at S_i iff-

a) S_j that reads x

b) path S_i to S_j

c) No write into x before S_j

eg



a, b, c, m, t_1

$In[n] = Use[n] \cup (Out[n] - Def[n])$

$Out[n] = \bigcup_{s \in succ(n)} In[s]$

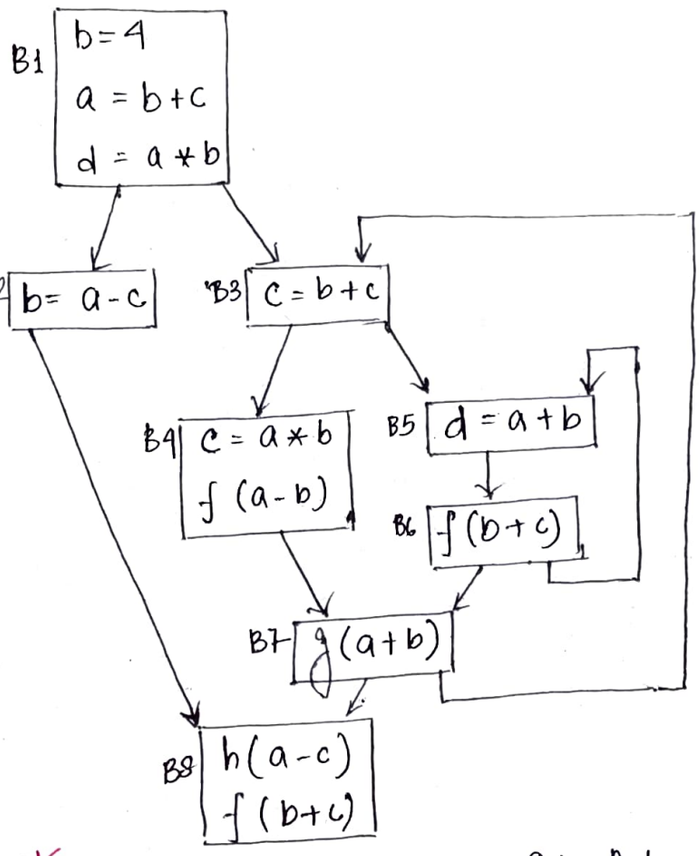
Use ~ Gen
Def ~ Kill
} While calculating don't look outside block

Block	Use (Gen)	Def (Kill)	1 In	1 Out	2 In	2 Out
1	{ }	{a, b, c, m}	ϕ	a, b, c, m	a, b, c, m	a, b, c, m
2	{a, m}	{ }	a, m, b, c	a, b, c, m	a, b, c, m	a, b, c, m
3	{a}	{a}	a, b, c, m	a, b, c, m	a, b, c, m	a, b, c, m
4	{a}	{ }	a, b, c	a, b, c	a, b, c	a, b, c
5	{a, b, c}	{t_1, a}	a, b, c	a	a, b, c	a
6	{a}	{ }	a	ϕ	a	ϕ

start

Block	Use	Def	Succ^r	Out	In	Out	In.
6	a	ϕ	-	ϕ	a		
5	a, b, c	t_1, a	6	a	a, b, c		
4	a	ϕ	5, 6	a, b, c	a, b, c	Same as iter^n 1.	
3	a	a	2	a, b, c, m	a, b, c, m		
2	a, m	ϕ	3, 4	a, b, c, m	a, m, b, c		
1	ϕ	a, b, c, m	2	a, b, c, m	ϕ		

eg.



$$\begin{cases} Out[n] = \bigcup_{s \in succ[n]} In[s] \\ In[n] = Use[n] \cup (Out[n] - Def[n]) \end{cases}$$

***

	Use	Def	Iter^n 1 Out	Iter^n 1 In	Iter^n 2 Out	Iter^n 2 In
8	a, b, c	ϕ	ϕ	a, b, c		
7	a, b	ϕ	a, b, c	a, b, c		
6	b, c	ϕ	a, b, c	b, c, a		
5	a, b	d	a, b, c	a, b, c	Same as iter^n 1	
4	a, b	c	a, b, c	a, b		
3	b, c	c	a, b, c	b, c, a		
2	a, c	b	a, b, c	a, c		
1	c	a, b, d	a, b, c	c.		

Liveness paths. —

- for c :

Start from block where c is live

$B_1 \rightarrow B_7 \rightarrow B_8$

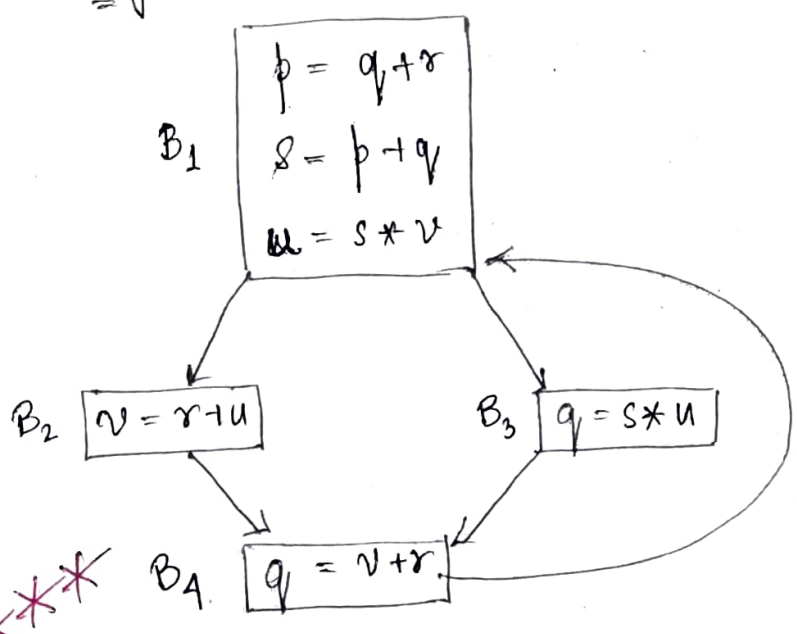
$B_1 \rightarrow B_2 \rightarrow B_8$

$B_3 \rightarrow B_5 \rightarrow B_6 \rightarrow B_7 \rightarrow B_8$

$B_3 \rightarrow B_5 \rightarrow B_4 \rightarrow B_5 \rightarrow B_6 \rightarrow B_7 \rightarrow B_8$

Length of greatest liveness path for c = 6

eg



$In[n] = Use[n] \cup (Out[n] - Def[n])$

$Out[n] = \bigcup_{s \in succ[n]} In[s]$

p, q, r, s, u, v

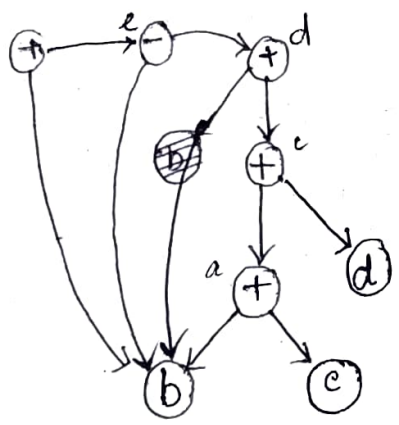
Block	Use	Def	Succ^r	Out	In	Out	In
4	v, r	q	1	q, r, v	v, r		
3	s, u	q	4	v, r	s, u, v, r		
2	r, u	v	4	v, r	r, u	Same as iter^n 1	
1	q, r, v	p, s, u	2, 3	s, u, v, r	q, r, v		

ME Examples QRC.

eg. Intermediate code

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

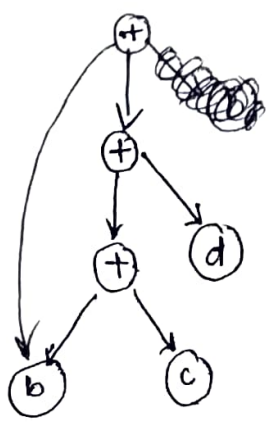
Min #nodes & edges present in DAG.



not minimum ✗

a = e + b = (d - b) + b = ((b + c) - b) + b = (a + d) + b
= ((b + c) + d) + b.

minimum ✓

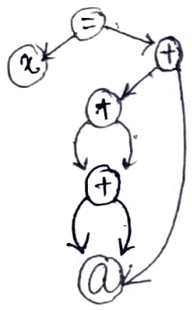


✓

#n = 6
#e = 6

eg $x = a+a+a+a+a$

min # nodes, # edges in DAG.



$\#n = 6$

$\#e = 8$

eg Least # temp var. reqd to create 3AC in SSA form.

$q + r/s + s - t*5 + u*v/w$.

*, /

+, -

3AC.

$r = r/s$ — reuse as nowhere r is there afterwards.

$t = t*5$

$u = u*v$

$w = u/w$

$q = q+r$

$s = q+s$

$t = s-t$

$w = t+w$

For 3AC, 0 new temp variables.

Best 3AC

SSA

$r_1 = r/s$

$t_1 = t*5$

$u_1 = u*v$

$w_1 = u_1/w$

$q_1 = q+r_1$

$s_1 = q_1+s$

$t_2 = s_1 - t_1$

$w_2 = t_2 + w_1$

Best SSA

For SSA, 8 new temp var.s ($r_1, t_1, w_1, u_1, q_1, s_1, t_2, w_2$)

eg $t_1 = a*a$

$t_2 = a*b$

$t_3 = t_1 * t_2$

$t_4 = t_3 + t_2$

$exp^n - t_4 = t_3 + t_2$

$= (t_1 * t_2) + t_2$

$= (t_1 * (a*b)) + (a*b)$

$= ((a*a) * (a*b)) + (a*b)$

min # temp for 3AC/SSA

3AC.

$b = a*b$ — b not used further, so reuse

$a = a*a$

$a = a*b$

$a = a+b$

temp = 0

best

SSA

$b_1 = a*b$

$a_1 = a*a$

$a_2 = a_1 * b_1$

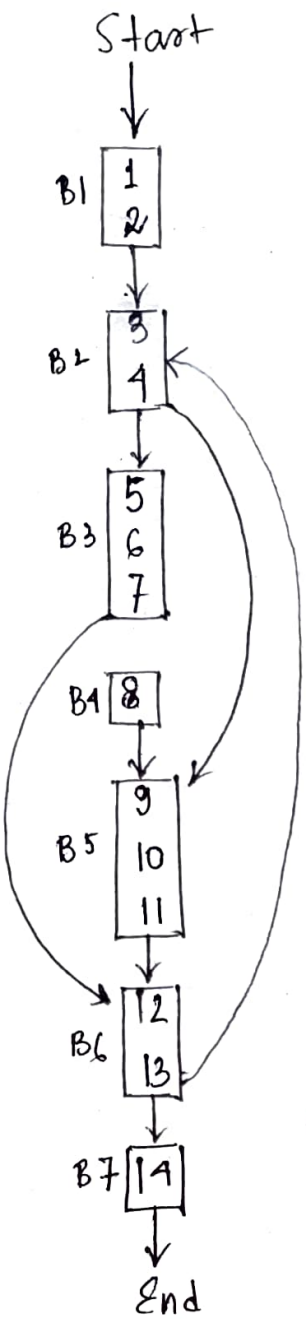
$a_3 = a_2 + b_1$

temp = 1

Total = 6

29 CFG

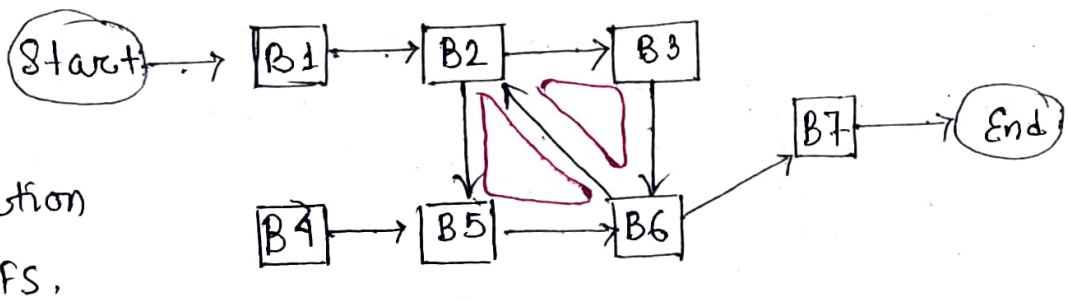
```
→ 1   x = a+b
  2   t1 = x+c
→ 3   t2 = a+t1
  4   if t2 > t1 goto 9
→ 5   t3 = t1+t2
  6   t5 = t3-t2
  7   goto 12
→ 8   t6 = t5/t2
→ 9   t3 = t1-t2
  10  a = b+c
  11  d = b-c
→ 12  t6 = t4+t9
  13  if t6 > t7 goto 3
→ 14  t4 = t7+t5
```



cond^l goto
→ 2 edges

uncond^l goto
→ 1 edge

Total
7+2 = 9 nodes
10 edges



cycle detection using DFS.

Cycle found in 3AC.

eg

Liveness

$x = a + b$.

$y = d + c$

$a = x + b$

$a = b + d$

$a = a + b + c$

a	b	c	d	x	y
l	l	l	l	d	d
d	l	l	l	l *	d
d	l	l	l	l	d
d	l	l	l	d	d
l	l	l	d	d	d